操作系统Lab4说明文档

个人信息

191250012 陈益超

实验说明

1 添加系统调用

1.1 my_sleep

以系统调用my_sleep为例，添加一个系统调用需要以下几个步骤：

1、在proto.h中声明两个函数

```
PUBLIC void my_sleep(int time);
```

```
PUBLIC void sys_my_sleep(int milli_seconds);
```

2、在syscall.asm中实现汇编部分代码

```
_NR_my_sleep        equ 1;
```

```
my_sleep:
    mov eax, _NR_my_sleep
    mov ecx, [esp+4]
    int INT_VECTOR_SYS_CALL
    ret
```

3、在kernel.asm中更改系统调用，进行参数的压栈和出栈

```
sys_call:
        call    save

        sti

        push    ecx ; 压栈
        call    [sys_call_table + eax * 4]
        add     esp, 4
        mov     [esi + EAXREG - P_STACKBASE], eax
        cli

        ret
```

4、在proc.c中实现c部分代码

```
PUBLIC void sys_my_sleep(int milli_seconds){
    int sleep_ticks = milli_seconds/1000*HZ;
    p_proc_ready->wakeup_tick = get_ticks()+sleep_ticks;
    schedule();
}
```

5、在global.c的sys_call_tabal中添加函数指针

```
PUBLIC  system_call  sys_call_table[NR_SYS_CALL] = {sys_get_ticks,
sys_my_sleep, sys_my_sprint, sys_P, sys_V};
```

汇编部分的代码实际上是通过函数列表，来调用c语言的具体函数实现。_NR_my_sleep指向global.c的sys_call_tabal中的第二个元素。system_call实际上是void*类型，而第二个元素是自己添加的sys_my_sleep函数指针，指向proc.c的具体函数实现。

sys_my_sleep实现的功能是让当前进程在接下来的milli_seconds毫秒内不参与进程调度，或者说不会被调度。将毫秒转换成tick，计算出醒来的时刻，调度算法通过判断当前ticks是否小于进程的wakeup_tick，来决定是否将其纳入调度的范围。具体的调度算法实现在后面再详细描述。

1.2 my_print

将打印字符串封装成了系统调用

```
PUBLIC void sys_my_sprint(void *str){
    disp_str(str);
}
```

1.3 信号量PV操作

信号量的数据结构:

```
typedef struct semaphore{
    int sem_count;
    PROCESS* waiting_proc[6];
    int wait_count;
    char name[32];
}SEM;
```

P操作:

```
PUBLIC void sys_P(void* mutex){
    disable_irq(CLOCK_IRQ); //
    SEM* sem = mutex;
    if (sem->sem_count==0){
        p_proc_ready->isWait = 1;
        sem->waiting_proc[sem->wait_count]=p_proc_ready;
        sem->wait_count++;
        schedule();
        enable_irq(CLOCK_IRQ);
        return;
    }
    sem->sem_count--;
    enable_irq(CLOCK_IRQ);
}
```

信号量结构体维护了一个等待进程的指针数组waiting_proc和等待进程数wait_count，数组一开始为空，wait_count初始为0，当有进程P操作这个信号量时，检查信号量的值，如果大于0则分配信号量给进程，然后自减；如果等于0，则将进程加入这个信号量的等待队列，将进程状态改为等待，调用调度算法切换进程。

V操作：

```
PUBLIC void sys_V(void* mutex){
    disable_irq(CLOCK_IRQ);
    SEM* sem = mutex;
    sem->sem_count++;
    if(sem->wait_count>0){
        // 出队
        p_proc_ready = sem->waiting_proc[0];
        p_proc_ready->isWait = 0;
        sem->sem_count--;
        for (int i = 0; i < sem->wait_count-1; i++){
            sem->waiting_proc[i] = sem->waiting_proc[i+1];
        }
        sem->wait_count--;
    }
    enable_irq(CLOCK_IRQ);
}
```

当一个进程释放一个信号量时，信号量的值加一，然后检查做这个信号量的等待队列中是否有进程在等待，如果有，则将信号量分配给等待队列的第一个进程，将其唤醒。

2 读写者问题

2.1 添加新的task

这里的task和进程是同一个概念

进程包括：进程表、进程体、GDT、TSS

添加进程包括：

1. task_table中增加一项task（global.c）
2. 让NR_TASKS加一(proc.h)
3. 定义任务堆栈（proc.h）
4. 修改STAVK_SIZE_TOTAL（proc.h）
5. 添加新任务执行体的函数声明（proto.h）
6. 添加新任务执行体的函数定义（这里我都写在了main.c中）

2.2 读者进程

```
void reader(int time)
{
    int sleep_time = time;
   while (1) {
      /* disp_str("A."); */
        switch (MODE) {
            case READER_FIRST:
                my_p(change_read);
                if (read_count == 0 && wait_read_count == 0) { // 说明当前并不在读，所以要等写进程释放信号量
                    my_p(write_sem);
```

```
                }
                wait_read_count++;
                my_v(change_read);

                my_p(read_sem);//开始读

                my_p(change_read);
                wait_read_count--;
                read_count++;
                my_v(change_read);
                //读
                read(time);

                my_p(change_read);
                read_count--;
                if (read_count == 0 && wait_read_count == 0) {  // 没有读进程了，包括等待中和进行中的。这才释放信号量给写进程
                    my_v(write_sem);
                }
                p_proc_ready->isFinish = 1;
                my_v(change_read);

                my_v(read_sem);//读完成
                break;
            case WRITER_FIRST:
                my_p(read_sem);//开始读

                my_p(change_read);
                read_count++;
                my_v(change_read);
                //读
                read(time);

                my_p(change_read);
                read_count--;
                p_proc_ready->isFinish = 1;
                my_v(change_read);

                my_v(read_sem);//读完成

                break;
            default:
                break;
        }
      milli_delay(10);
        switch (H_MODE) {
            case ACCEPT:
                my_sleep(1);
                break;
            case AVOID:
                my_sleep(sleep_time);
                break;
            default: ;
        }

    }
}
```

2.3 写者进程

```c
void writer(int time){
    int i = 0x3000;
    int sleep_time = time;
    while (1){
        switch (MODE) {
            case READER_FIRST:
                my_p(write_sem);
                my_p(change_write);
                write_count++;
                // 写
                write(time);
                write_count--;
                p_proc_ready->isFinish = 1;
                my_v(change_write);
                my_v(write_sem);
                break;
            case WRITER_FIRST:
                my_p(change_write);
                if (write_count == 0 && wait_write_count == 0) {// 说明当前并不在写，所以要等读进程释放信号量
                    for (int j = 0; j < MAX_READERS; ++j) {
                        my_p(read_sem);

                    }
                }
                wait_write_count++;
                my_v(change_write);

                my_p(write_sem);//开始写

                my_p(change_write);
                wait_write_count--;
                write_count++;
                my_v(change_write);
                //写
                write(time);

                my_p(change_write);
                write_count--;
                if (write_count == 0 && wait_write_count == 0) {  // 没有写进程了，包括等待中和进行中的。这才释放信号量给读进程
                    for (int j = 0; j < MAX_READERS; ++j) {
                        my_v(read_sem);
                    }
                }
                p_proc_ready->isFinish = 1;
                my_v(change_write);
                my_v(write_sem);//写完成
                break;
            default:
                break;
        }
        milli_delay(10);
        switch (H_MODE) {
            case ACCEPT:
```

```
                my_sleep(1);
                break;
            case AVOID:
                my_sleep(sleep_time);
                break;
            default: ;
        }
    }

}
```

2.4 说明

读写者进程涉及四个信号量的操作

```
SEM sem_table[] = {
        {MAX_READERS,{},0,"read_sem"},
        {1,{},0,"write_sem"},
        {1,{},0,"write_block"},
        {1,{},0,"change_read"},
        {1,{},0,"read_block"},
        {1,{},0,"change_write"}

};
SEM *read_sem = &sem_table[0];
SEM *write_sem = &sem_table[1];
SEM *change_read = &sem_table[3];
SEM  *change_write = &sem_table[5];
```

MAX_READERS表示允许同时读的人数。MODE表示是读者优先还是写者优先，H_MODE表示是否要解决饿死。read_sem控制读操作，write_sem控制写操作，change类信号量控制对对应count计数器的修改。

读者优先：

读者进程有三种状态，一种是正在读的状态，这个状态下的读者已经拿到了read_sem这个信号量；一种是等待读的状态，还没有去拿或者没有拿到read_sem，但是有读的意愿；最后一种是既不在读，也不在等待读。写者也有类似的三种状态。

读进程在读前，还需要保证有一个读进程拿到了write_sem，这样在读时不会有写操作。

在读者优先的模式中，只要有进程在读，或者在等待读，那么就不会释放写信号量，这样写进程就不会执行。只有当这两种状态的读者数量为0时，才会释放写信号量。

写者优先：

与读者优先类似，写者在执行前需要去抢占读信号量，当有写者在写或在等待写时，不释放读信号量。不同的是，写信号量只有一个，而读信号可能有多个，所以写者在抢占和释放读信号量时需要循环操作。

解决饿死：

无论是读者优先还是写者优先，都会出现饿死的情况。这是因为无论是读进程还是写进程，在完成相应的读写操作后，又立马回到等待队列当中，这样就导致了读信号或写信号总是不能释放。

解决的办法是让读过或写过的进程暂时休息一段时间，不参与接下来的进程调度，即不去等待，这样就使得信号量得以释放。

```
        case AVOID:
            my_sleep(sleep_time);
            break;
```

2.5 进程调度算法

```
PUBLIC void schedule()
{
   PROCESS* p;
   int     greatest_ticks = 0;

   while (!greatest_ticks) {
      for (p = proc_table; p < proc_table+NR_TASKS; p++) {
         if (p->isWait == 0 && p->wakeup_tick<=ticks && p->ticks >
greatest_ticks){
            greatest_ticks = p->ticks;
            p_proc_ready = p;

         }
            // 选择进程的条件 :没有被阻塞（即没有因为拿不到信号量被挂起）, 进程仍需要的
时间片, 进程没有睡眠
      }

      if (!greatest_ticks) {
         for (p = proc_table; p < proc_table+NR_TASKS; p++) {
            p->ticks = p->priority;
         }
      }
   }
}
```

一个tick其实是非常短的时间，如果将一个tick当做一个时间片，并且依据p->ticks判断进程是否执行结束，得到的结果会比较奇怪。期望的结果是各个进程所占用时间比例是明确的，如果读写进程执行milli_delay时间较短（几个ticks），因为进程的其他语句也会占用tick，最后每个进程占用的时间比例就不一定是期望的比例。只有当milli_delay的时间远大于其他语句执行的时间，进程的时间比例才符合预期。因此，进程调度中的p->ticks的作用其实就十分有限了。

初始化时将各个进程的p->ticks和p->priority设为相同的值，按照ABCDEF的顺序（FIFO）调度。

2.6 F进程

```
void TestF(){
    int i = 0x5000;
    int time = 10000;
    // my_sprint("start_F");
    while (1){
        if(read_count!=0){
            my_sprint("reading:");
            disp_int(read_count);
            my_sprint(" ");
        } else{
            my_sprint("writing:");
            disp_int(write_count);
            my_sprint(" ");
        }
        my_sleep(time);
```

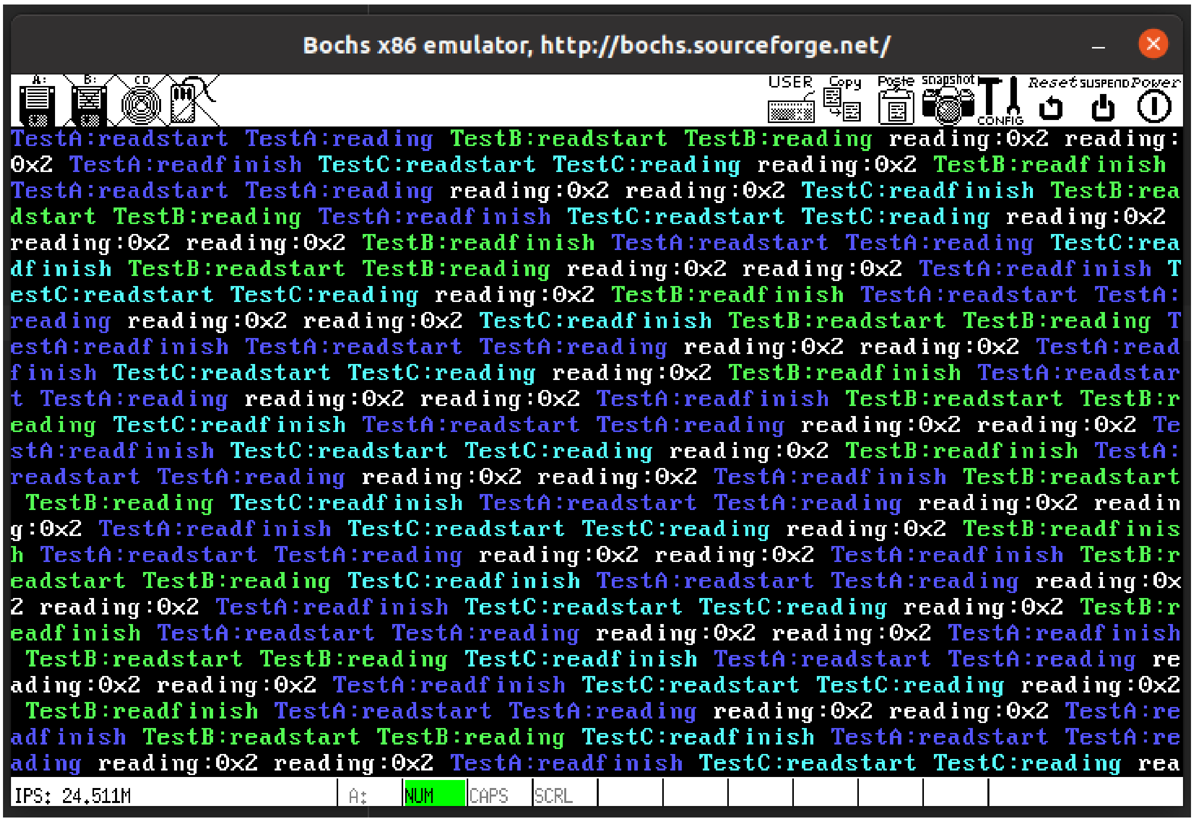
```
    }
    my_sleep(1);
}
```

F进程每隔一个"时间片"打印一次，因此通过F进程可以知道过了多久的时间，值得注意的是F进程用的是my_sleep而不是milli_delay，不占用"时间片"。

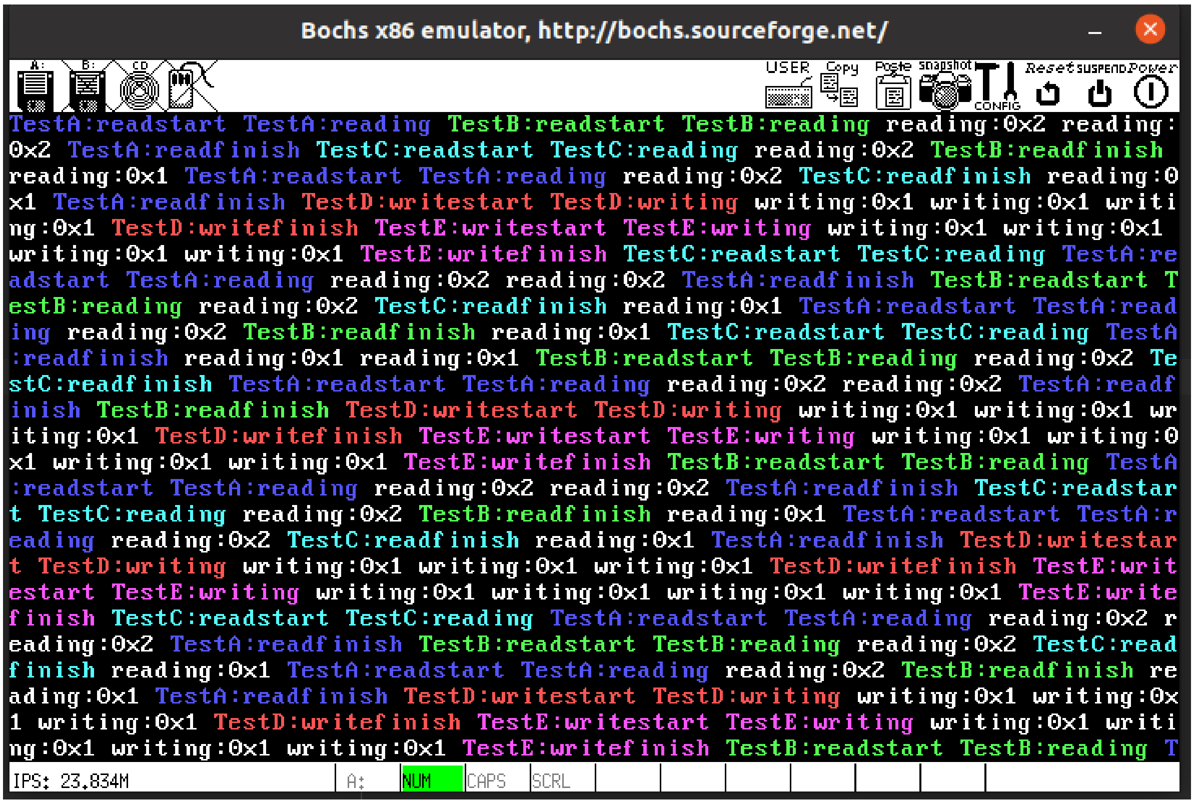
实验截图

最终结果实现的是"并发"的执行。以同时允许两个读者读为例

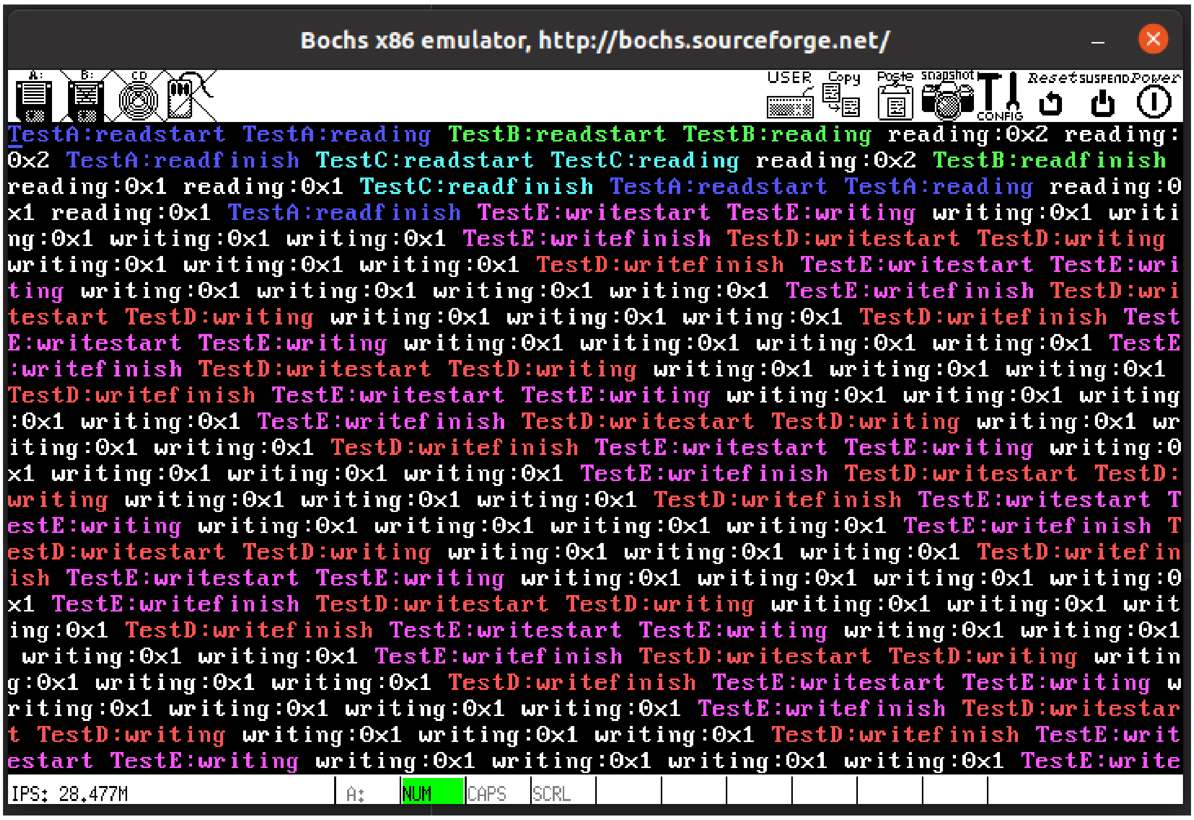
读者优先、允许饿死：



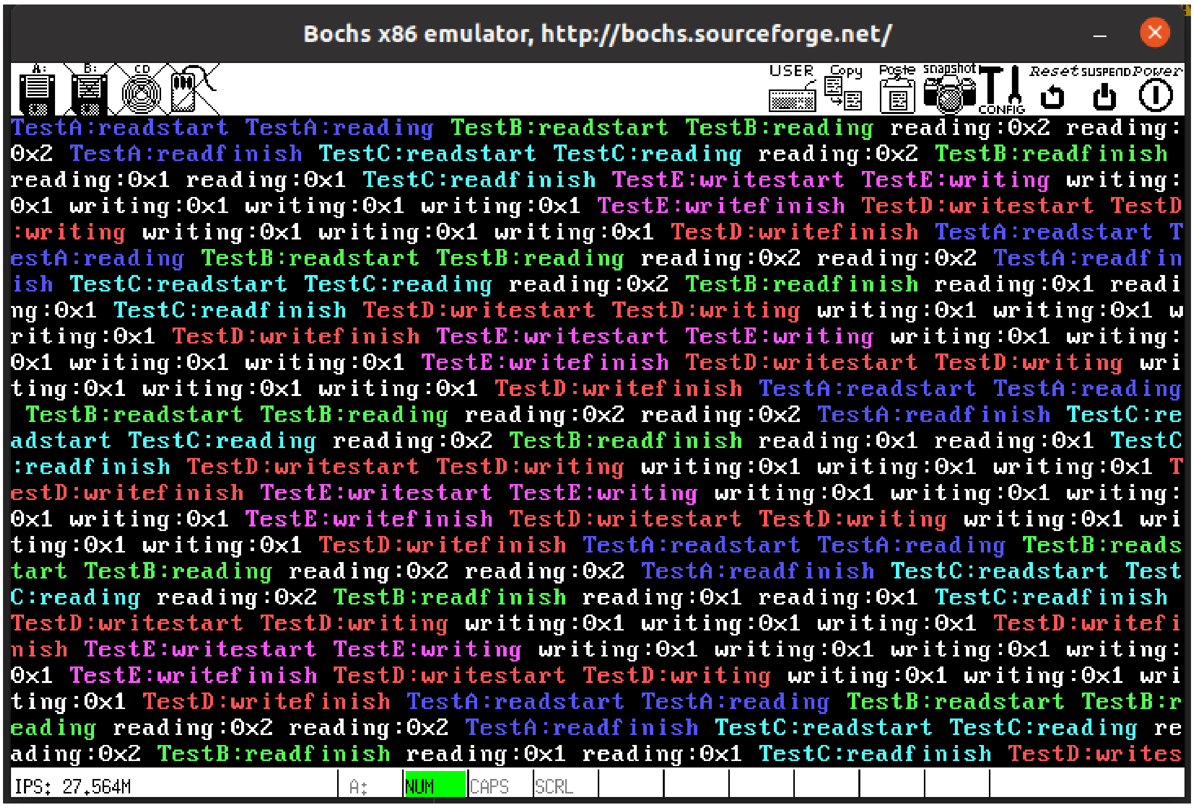
读者优先、解决饿死：



写者优先、允许饿死：



写者优先、解决饿死：


Bochs x86 emulator, http://bochs.sourceforge.net/
USER Copy Paste snapshot CONFIG Reset SUSPEND Power
TestA:readstart TestA:reading TestB:readstart TestB:reading reading:0x2 reading:
0x2 TestA:readfinish TestC:readstart TestC:reading reading:0x2 TestB:readfinish
reading:0x1 reading:0x1 TestC:readfinish TestE:writestart TestE:writing writing:
0x1 writing:0x1 writing:0x1 writing:0x1 TestE:writefinish TestD:writestart TestD
:writing writing:0x1 writing:0x1 writing:0x1 TestD:writefinish TestA:readstart T
estA:reading TestB:readstart TestB:reading reading:0x2 reading:0x2 TestA:readfin
ish TestC:readstart TestC:reading reading:0x2 TestB:readfinish reading:0x1 readi
ng:0x1 TestC:readfinish TestD:writestart TestD:writing writing:0x1 writing:0x1 w
riting:0x1 TestD:writefinish TestE:writestart TestE:writing writing:0x1 writing:
0x1 writing:0x1 writing:0x1 TestE:writefinish TestD:writestart TestD:writing wri
ting:0x1 writing:0x1 writing:0x1 TestD:writefinish TestA:readstart TestA:reading
TestB:readstart TestB:reading reading:0x2 reading:0x2 TestA:readfinish TestC:re
adstart TestC:reading reading:0x2 TestB:readfinish reading:0x1 reading:0x1 TestC
:readfinish TestD:writestart TestD:writing writing:0x1 writing:0x1 writing:0x1 T
estD:writefinish TestE:writestart TestE:writing writing:0x1 writing:0x1 writing:
0x1 writing:0x1 TestE:writefinish TestD:writestart TestD:writing writing:0x1 wri
ting:0x1 writing:0x1 TestD:writefinish TestA:readstart TestA:reading TestB:reads
tart TestB:reading reading:0x2 reading:0x2 TestA:readfinish TestC:readstart Test
C:reading reading:0x2 TestB:readfinish reading:0x1 reading:0x1 TestC:readfinish
TestD:writestart TestD:writing writing:0x1 writing:0x1 writing:0x1 TestD:writefi
nish TestE:writestart TestE:writing writing:0x1 writing:0x1 writing:0x1 writing:
0x1 TestE:writefinish TestD:writestart TestD:writing writing:0x1 writing:0x1 wri
ting:0x1 TestD:writefinish TestA:readstart TestA:reading TestB:readstart TestB:r
eading reading:0x2 reading:0x2 TestA:readfinish TestC:readstart TestC:reading re
ading:0x2 TestB:readfinish reading:0x1 reading:0x1 TestC:readfinish TestD:writes
IPS: 27.564M
A:
NUM
CAPS
SCRL