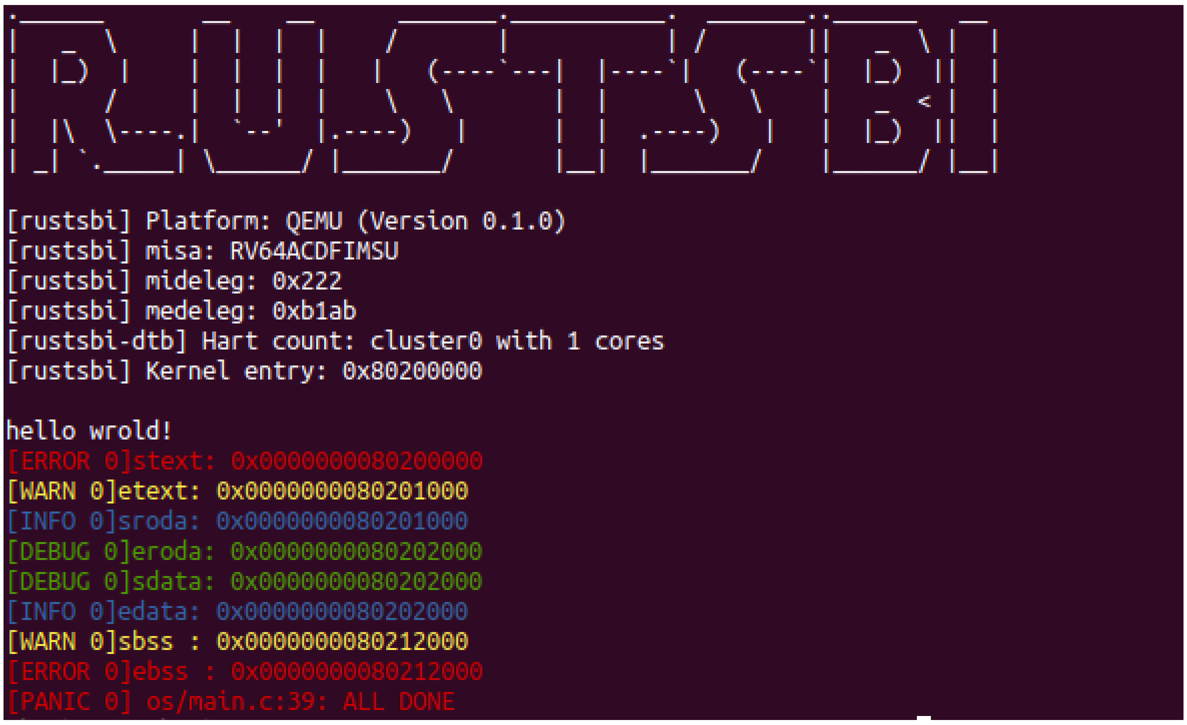
ch1

计83李天勤2018080106

实验目的

1. Understand framework of lab
2. 读懂os/kernel.ld和Makefile
3. make debug,自学gdb调试的方式

实验结果



```
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000

hello wrold!
[ERROR 0]stext: 0x0000000080200000
[WARN 0]etext: 0x0000000080201000
[INFO 0]sroda: 0x0000000080201000
[DEBUG 0]eroda: 0x0000000080202000
[DEBUG 0]sdata: 0x0000000080202000
[INFO 0]edata: 0x0000000080202000
[WARN 0]sbss : 0x0000000080212000
[ERROR 0]ebss : 0x0000000080212000
[PANIC 0] os/main.c:39: ALL DONE
```

问答作业

1. 请学习gdb调用工具的使用（这对后续调试很重要），并通过gdb简单跟踪从机器加电到跳转到0x80200000的简单过程，就需要描述重要跳转即可，就需要描述上qemu的情况。

首先我们用gdb看汇编，最开头0x1000那四五个执行. 0x80000000以后可直接看rustbi的源代码，不需要看gdb汇编了，但是我用break和continue看了这几个执行

```
$ make debug
0x0000000000001000 in ?? ()
Breakpoint 1 at 0x1000
(gdb) ni
0x0000000000001004 in ?? ()
1: x/12i $pc-8
   0xffc:   unimp
   0xffe:   unimp
   0x1000:  auipc   t0,0x0          # add upper immediate to pc
=> 0x1004:  addi    a1,t0,32        # add immediate
   0x1008:  csrr    a0,mhartid      # control status register read machine
hardware thread id to a0
   0x100c:  ld  t0,24(t0)
   0x1010:  jr  t0                      # jump to RustSBI_start(0x80000000)
   0x1014:  unimp
   0x1016:  unimp
   0x1018:  unimp
   0x101a:  0x8000
   0x101c:  unimp
(gdb) break *0x80000000
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000080000000 in ?? ()
1: x/12i $pc-8
   0x7ffffff8:  unimp
   0x7ffffffa:  unimp
   0x7ffffffc:  unimp
   0x7ffffffe:  unimp
=> 0x80000000:  csrr    a2,mhartid              # RustSBI_start
   0x80000004:  lui     t0,0x0
   0x80000008:  addi    t0,t0,7
   0x8000000c:  bltu    t0,a2,0x8000003a
   0x80000010:  auipc   sp,0x200
   0x80000014:  addi    sp,sp,-16
   0x80000018:  lui     t0,0x10
   0x8000001c:  mv      t0,t0
(gdb) ni
...
 0x0000000080000036 in ?? () # jump to RustSBI main
1: x/12i $pc-8
   0x8000002e:  sub     sp,sp,t0
   0x80000032:  csrw    mscratch,zero
=> 0x80000036:  j       0x80002572
   0x8000003a:  wfi
   0x8000003e:  j       0x8000003a
   0x80000040:  unimp
   0x80000042:  addi    sp,sp,-16
   0x80000044:  mv      a1,a0
   0x80000046:  sw      a0,8(sp)
   0x80000048:  sw      a0,12(sp)
   0x8000004a:  slli    a0,a0,0x20
   0x8000004c:  srli    a0,a0,0x20
(gdb) ni
[rustsbi] RustSBI version 0.1.1
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000

Breakpoint 4, 0x0000000080002cce in ?? () # jump to enter+priveleged
(0x80001504)
```

```
1: x/12i $pc-8
   0x80002cc6:  ld      a0,16(sp)
   0x80002cc8:  ld      a1,320(sp)
   0x80002cca:  auipc   ra,0xfffff
=> 0x80002cce:  jalr    -1990(ra)
   0x80002cd2:  unimp
   0x80002cd4:  addi    sp,sp,-464
   0x80002cd6:  sd      ra,456(sp)
   0x80002cd8:  sd      a0,400(sp)
   0x80002cda:  li      a1,0
   0x80002cdc:  sb      a1,399(sp)
   0x80002ce0:  mv      a1,a0
   0x80002ce2:  sd      a0,408(sp)
(gdb) ni
...
0x000000008000150e in ?? () # jump to RustSBI's mode_start (0x800023da)
1: x/12i $pc-8
   0x80001506:  sd      a0,0(sp)
   0x80001508:  sd      a1,8(sp)
   0x8000150a:  csrrw   sp,mscratch,sp
=> 0x8000150e:  mret
   0x80001512:  unimp
   0x80001514:  addi    sp,sp,-48
   0x80001516:  sd      ra,40(sp)
   0x80001518:  sd      a0,24(sp)
   0x8000151a:  sd      a1,32(sp)
   0x8000151c:  li      a2,0
   0x8000151e:  sd      a1,0(sp)
   0x80001520:  beq     a0,a2,0x80001534

0x00000000800023e2 in ?? () # jump to 0x802000000, the memory location where
$KERNEL_BIN is located, and execute the first instruction of the operating
system
hello wrold!
[ERROR 0]stext: 0x0000000080200000
[WARN 0]etext: 0x0000000080201000
[INFO 0]sroda: 0x0000000080201000
[DEBUG 0]eroda: 0x0000000080202000
[DEBUG 0]sdata: 0x0000000080202000
[INFO 0]edata: 0x0000000080202000
[WARN 0]sbss : 0x0000000080212000
[ERROR 0]ebss : 0x0000000080212000
[PANIC 0] os/main.c:39: ALL DONE
Remote connection closed
(gdb) quit
```

ch2

实验目的

The purpose of this chapter is to complete these two functionalities

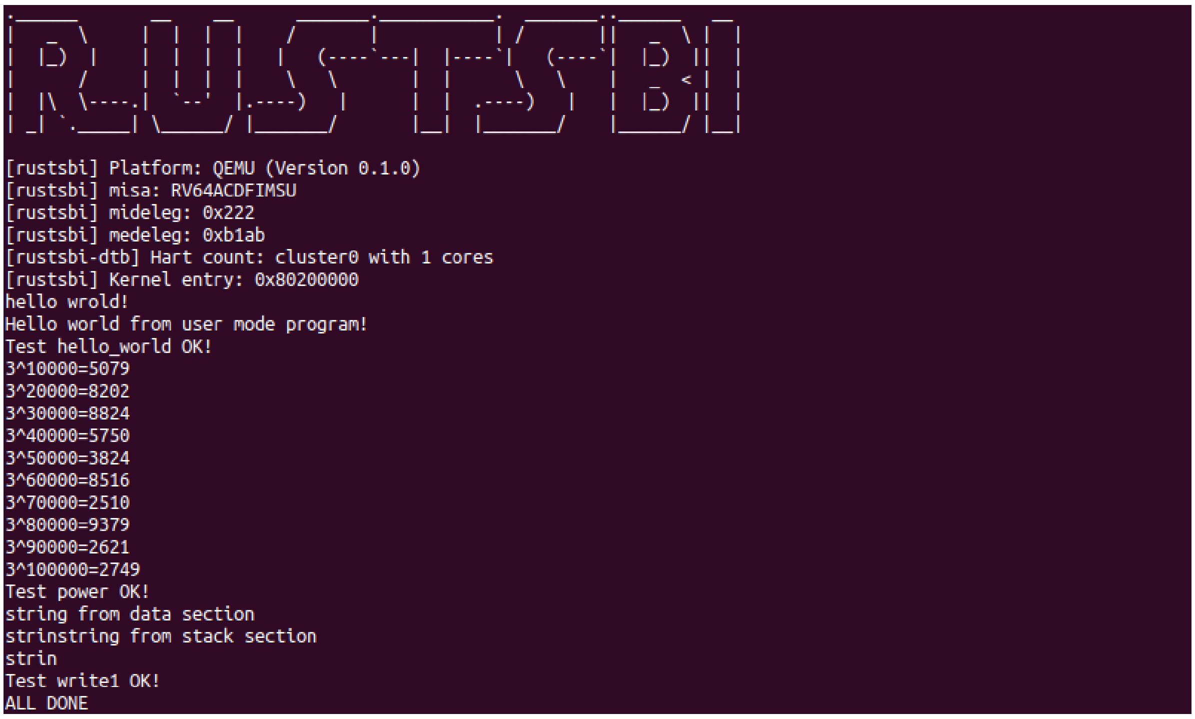
1. Support automatic loading and running of multiple programs through batch processing
2. Protect the operating system using hardware privilege level mechanism
3. Read and understand scheduler and syscall

实验原理

The core idea of the Batch System is to pack multiple programs into the computer. When a program is finished, the computer will automatically load the next program into memory and start execution. This system has to make sure that errors in a program does not affect the operating system itself, instead, it only terminates the errored program and continues with the same. This is known as the Privilege mechanism, realizing the isolation between user mode and kernel mode and requires joint efforts between software and hardware.

实验结果

```
git checkout ch2
make test BASE=1
```



```
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
hello wrold!
Hello world from user mode program!
Test hello_world OK!
3^10000=5079
3^20000=8202
3^30000=8824
3^40000=5750
3^50000=3824
3^60000=8516
3^70000=2510
3^80000=9379
3^90000=2621
3^100000=2749
Test power OK!
string from data section
strinstring from stack section
strin
Test write1 OK!
ALL DONE
```

We can see that the first four basic tests run correctly.

The biggest difference between ch1 and ch2 is that this chapter supports applications running in user mode and can complete system calls issued by the application, and run different applications one after another.

Here, the relatively loose application program execution code and operating system execution code can be connected together, so that the qemu-system-riscv64 emulator can load the two into the memory at one time and allow the operating system to find the location of the application. But first, the generated application program needs to be modified.

1. App program execution file is changed from ELF to BInary
2. `scripts/pack.py` generate assembly file `os/link_app.S` and generate auxilary information so its location can be found by the operating system.
3. `scripts/kernelld.py` will generate new `kernell_app.ld`
4. Compiler will combine the os with `os/link_app.S` to compile ELF of OS + Binary application, and further transform into Binary format.
5. Then OS will complete search of Binary application, copy it to physical memory, allocate series of resources (user stack/kernel stack)

问道作业

1. 正确进入 U 态后，程序的特征还应有：使用 S 态特权指令，访问 S 态寄存器后会报错。请同学们可以自行测试这些内容（参考 前三个测例，描述程序出错行为，同时注意注明你使用的 sbi 及其版本。

   【解答】

   ```
   [rustbi] RustSBI version 0.1.1

   __ch2_bad_address.c, writing 0s at 0x0 causes error, bad_address
   __ch2_bad_instruction.c # error using SRET, bad_instruction
   __ch2_bad_register.c # error accessing sstatus using csrr, bad_register
   ```

2. 请结合用例理解 trampoline.S 中两个函数 userret 和 uservec 的作用，并回答如下几个问题:
   1. L79: 刚进入 userret 时，a0、a1 分别代表了什么值。

      ```
      a0 represents trapframe's initial address
      a1 represents user page initial address
      ```

   2. L87-L88: sfence 指令有何作用？为什么要执行该指令，当前章节中，删掉该指令会导致错误吗？

      ```
      csrw satp, a1
      sfence.vma zero, zero

      # sfence.vma instruction refreshes/flushes TLB entries, it is done so
      because userret needs to complete conversion from S state to U state.
      Because U state uses page tables and TLB, we need to refresh the cached
      content.  Page table has not been implemented, so deletion will have no
      effect.
      ```

   3. L96-L125: 为何注释中说要除去 a0？哪一个地址代表 a0？现在 a0 的值存在何处？

      ```
      # restore all but a0 from TRAPFRAME
      ld ra, 40(a0)
      ld sp, 48(a0)
      ld t5, 272(a0)
      ld t6, 280(a0)

      #
      1. a0 is used to obtain the values of other registers saved in
      trapframe, thus must be saved after the other registers are saved.
      2. the address 112(a0) represents a0.
      3. a0 is eventually saved in the sscratch register
      ```

   4. userret：中发生状态切换在哪一条指令？为何执行之后会进入用户态？

      ```
      csrrw a0, sscratch, a0

      # state switch occurs at the last sret instruction of userret.
      # Since sstatus and sepc are set in usertrapret, and the sret
      instruction will cause the pc to jump to the next instruction of the
      interrupt instruction
      ```

   5. L29： 执行之后，a0 和 sscratch 中各是什么值，为什么？

```
csrrw a0, sscratch, a0

# After execution, a0 is the starting address of trapframe, and sscratch
is the value in the original U state a0 register. This is because before
execution, sscratch is the starting address of the trapframe, and the
value in a0 is set in the U state. The above statement is to exchange a0
and sscratch.
```

6. L32-L61: 从 trapframe 第几项开始保存？为什么？是否从该项开始保存了所有的值，如果不是，为什么？

```
sd ra, 40(a0)
sd sp, 48(a0)
...
sd t5, 272(a0)
sd t6, 280(a0)

# Start saving from the 6th item, because the first five items are all
values of S state, you don't need to save it when you enter the S state
from the U state.
# Do not save from a0 to 112(a0), since at this time a0 stores the
starting address of the trapframe, and the real U state a0 register
value is stored in sscratch.
```

7. 进入 S 态是哪一条指令发生的？ `enter the S state when the ecall instruction is executed`

8. L75-L76: `ld t0, 16(a0)` 执行之后，`t0` 中的值是什么，解释该值的由来？

```
ld t0, 16(a0)
jr t0

# value of t0 is the start address of usertrap.
16(a0) refers to the third item in trapframe, kernel_trap, and this item
is assigned to the start address of usertrap in usertrapret.
```

3. 请依次简要回答如下问题：

- 程序陷入内核的原因有中断和异常（系统调用），请问 RISC-V 64 支持哪些中断 / 异常？ 如何判断进入内核是由于中断还是异常？请描述陷入内核时的几个重要寄存器及其值。

Interrupt	Exception Code	Description
1	0	user software interrupt
1	1	supervisor software interrupt
1	2	hyper visor software interrupt
1	3	machine software interrupt
1	4	user timer interrupt
1	5	supervisor timer interrupt
1	6	hpervisor timer interrupt
1	7	machine timer interrupt
1	8	user external interrupt
1	9	supervisor external interrupt
1	10	hypervisor external interrupt
1	11	machin external interupt
1	≥ 12	reserved
0	0	Instruction address misaligned
0	1	instruction access fault
0	2	illegal instruction
0	3	breakpoint
0	4	load address misaligned
0	5	load access fault
0	6	store/AMO address misaligned
0	7	store/AMO access fault
0	8	environment call from U-mode
0	9	environment call from S-mode
0	10	environment call from H-mode
0	11	environment call from M-mode
0	$\geq$12	reserved

We can use the first bit of the `scause` register to determine whether the reason for entering the kernel is an interrupt or an exception

There are 8 important registers

- `stvec`: supervisor trap handler base address
- `sepc`: supervisor exception program counter.
- `scause`: supervisor trap cause.

    - `sie`: Supervisor interrupt-enable register.
    - `sip`: Supervisor interrupt pending.
    - `stval`: additional information: the address of the error in the address exception, the instruction of the illegal instruction exception, etc.;
    - `sscratch`: scratch register for supervisor trap handlers.
    - `sstatus`: supervisor status register.
- 为了方便 os 处理，M 态软件会将 S 态异常/中断委托给 S 态软件，请指出有哪些寄存器记录了委托信息。RustSBI 委托了哪些异常/中断？（提示：看看 RustSBI 在启动时输出了什么？）
    - We can look at `trap.c` to see which interrupts that rustbi supports. We see that after interrupt is saved, different processing procedures will occur depending on the type of cause.

ch3

实验目的

1. Realize a safer sys_write
2. Realize stride algorithm

实验原理

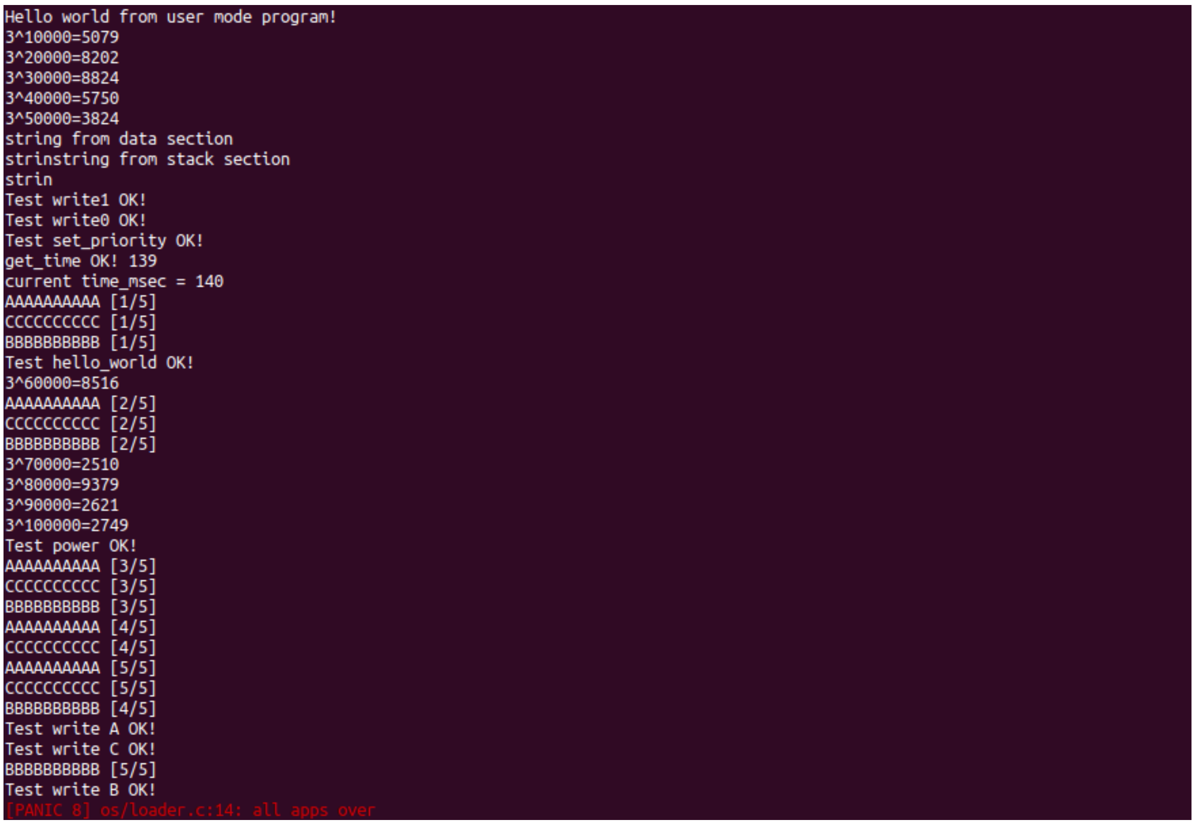
Since we have not implemented virtual memory, we have to modify `sys_write` to check whether the incoming buffer is located at the user's address. If not, `sys_write` should return -1.

Secondly, we also have to implement the concept of task scheduling, which can switch between different tasks. Currently, there is no priority scheduling algorithm for our operating system. The algorithm we will implement is known as the Stride Scheduler.

This includes adding new fields to the proc structure and realizing `set_priority`, as well as updating the scheduler located in `proc.c`

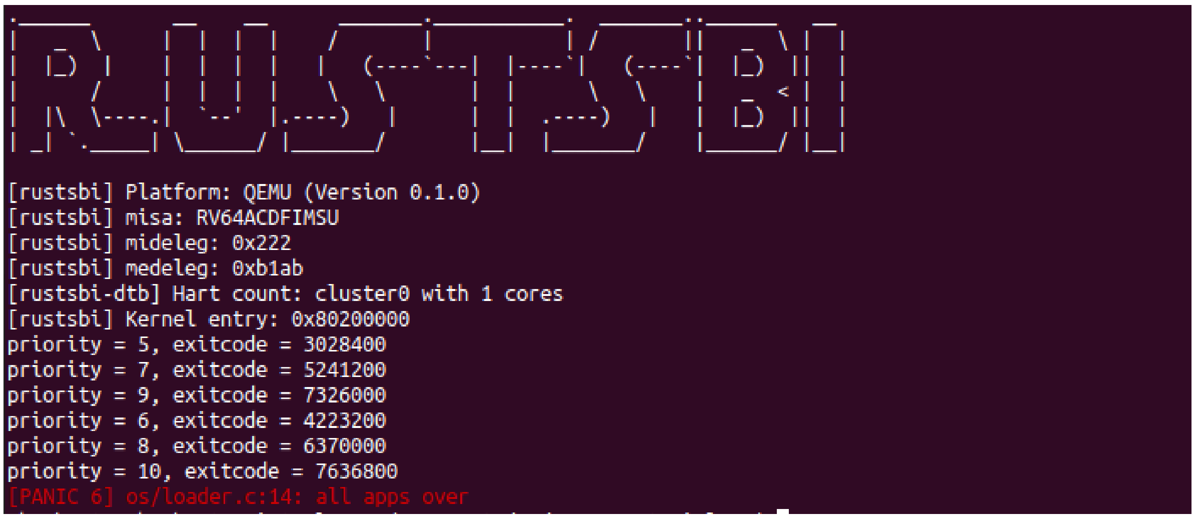
实验结果

run `make test CHAPTER=3`, from this we keen see that the `ch2b_write0` and `ch2b_write1` both pass, making our `sys_write` implementation correct as well.



```
Hello world from user mode program!
3^10000=5079
3^20000=8202
3^30000=8824
3^40000=5750
3^50000=3824
string from data section
strinstring from stack section
strin
Test write1 OK!
Test write0 OK!
Test set_priority OK!
get_time OK! 139
current time_msec = 140
AAAAAAAAAA [1/5]
CCCCCCCCCC [1/5]
BBBBBBBBBB [1/5]
Test hello_world OK!
3^60000=8516
AAAAAAAAAA [2/5]
CCCCCCCCCC [2/5]
BBBBBBBBBB [2/5]
3^70000=2510
3^80000=9379
3^90000=2621
3^100000=2749
Test power OK!
AAAAAAAAAA [3/5]
CCCCCCCCCC [3/5]
BBBBBBBBBB [3/5]
AAAAAAAAAA [4/5]
CCCCCCCCCC [4/5]
AAAAAAAAAA [5/5]
CCCCCCCCCC [5/5]
BBBBBBBBBB [4/5]
Test write A OK!
Test write C OK!
BBBBBBBBBB [5/5]
Test write B OK!
[PANIC 8] os/loader.c:14: all apps over
```

run `make test CHAPTER=3t`



```
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
priority = 5, exitcode = 3028400
priority = 7, exitcode = 5241200
priority = 9, exitcode = 7326000
priority = 6, exitcode = 4223200
priority = 8, exitcode = 6370000
priority = 10, exitcode = 7636800
[PANIC 6] os/loader.c:14: all apps over
```

实验问答

1、stride 算法深入

stride 算法原理非常简单，但是有一个比较大的问题。例如两个 pass = 10 的进程，使用 8bit 无符号整形储存 stride， p1.stride = 255, p2.stride = 250，在 p2 执行一个时间片后，理论上下一次应该 p1 执行。

- 实际情况是轮到 p1 执行吗？为什么？

```
Since a 8-bit unsigned is used to store stride, after P2 executes for a time
slice, p2.stride = 250 + 10 = 4 < p1.stride = 255, so p2 is still executed
next.
```

我们之前要求进程优先级 >= 2 其实就是为了解决这个问题。可以证明，**在不考虑溢出的情况下**, 在进程优先级全部 >= 2 的情况下，如果严格按照算法执行，那么 STRIDE_MAX – STRIDE_MIN <= BigStride / 2。

- 为什么？尝试简单说明（传达思想即可，不要求严格证明）。

  ```
  if you meet the required 'stride' at a certain moment, STRIDE_MAX -
  STRIDE_MIN <= BigStride /2, the corresponding process STIDE_MIN should be
  executed at the next time period. At this moment, MAX_STIDE = max(MAX_STIDE,
  MIN_STRIDE + BigStride / priority) <= max(MAX_STRIDE, MIN_STRIDE + BigStride
  / 2), which fits the above comparison.
  ```

已知以上结论，**在考虑溢出的情况下**，假设我们通过逐个比较得到 Stride 最小的进程，请设计一个合适的比较函数，用来正确比较两个 Stride 的真正大小：

```
typedef unsigned long long Stride_t;
const Stride_t BIG_STRIDE = 0xffffffffffffffffULL;
int cmp(Stride_t a, Stride_t b) {
   if (a < b)  return b - a <= BIG_STRIDE / 2;
   else return a - b > BIG_STRIDE / 2;
}
```