Lab 3

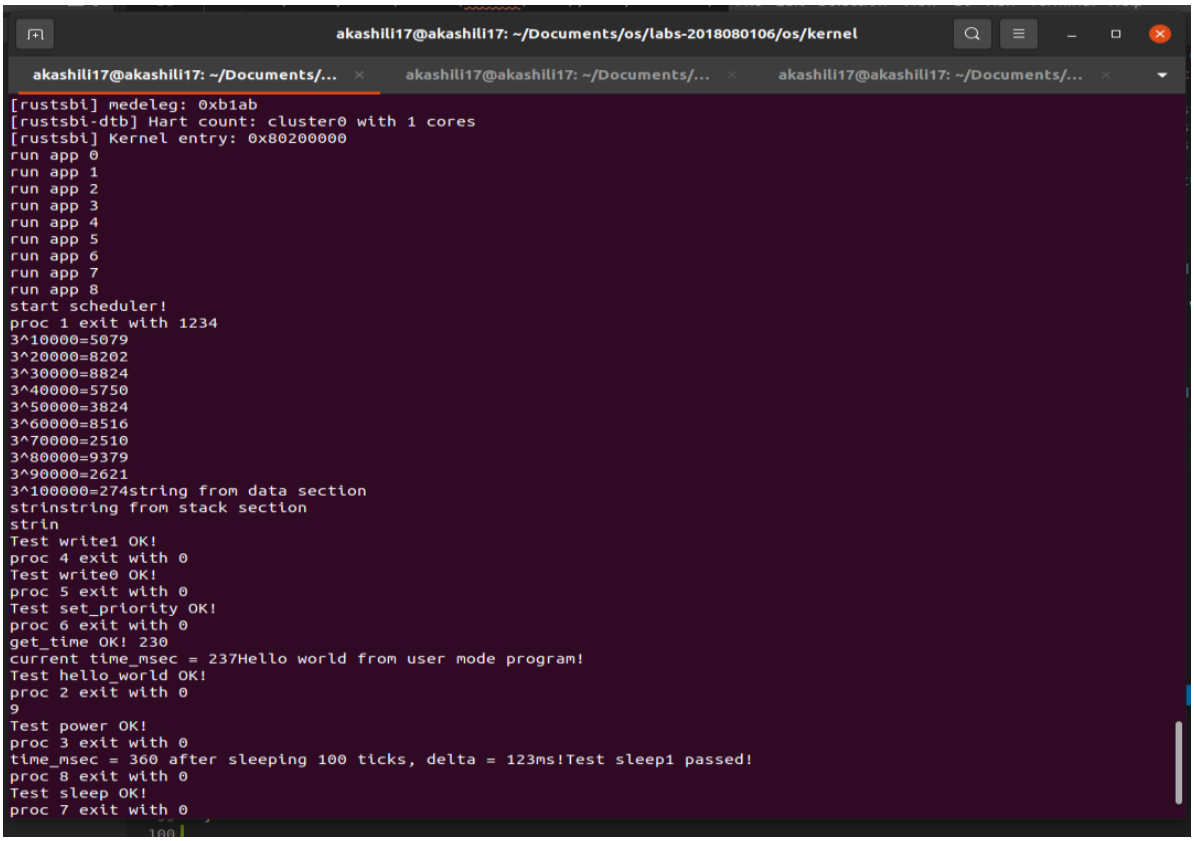
计 83 李天勤 201808106

实验目的

1. 实现分ch3
2. 实现sys_yeild,实现写作试喝抢占式的调度
3. 实现stride带哦都算法，实现sys_gettime,sys_set_priority
4. 通过riscvos-c-tests测试

实验结果

checking testcases `make all CHAPTER=3_0` in user



checking testcases `make all CHAPTER=3_1` in user

```
akashili17@akashili17: ~/Documents/os/labs-2018080106/os/kernel
qemu-system-riscv64 -nographic -smp 1 -machine virt -bios ../bootloader/rustsbi-qemu.bin -kernel kernel
[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
run app 0
run app 1
run app 2
start scheduler!
AAAAAAAAAA [1/5]
CCCCCCCCCC [1/5]
BBBBBBBBBB AAAAAAAAAA [2/5]
CCCCCCCCCC [2/5]
[1/5]
AAAAAAAAAA [3/5]
CCCCCCCCCC [3/5]
BBBBBBBBBB [2/5]
AAAAAAAAAA [4/5]
CCCCCCCCCC [4/5]
BBBBBBBBBB [3/5]
AAAAAAAAAA [5/5]
CCCCCCCCCC [BBBBBBBBBB [4/5]
Test write A OK!
proc 1 exit with 0
5/5]
BBBBBBBBBB [5/5]
Test write C OK!
proc 2 exit with 0
Test write B OK!
proc 3 exit with 0
panic: all apps over

akashili17@akashili17:~/Documents/os/labs-2018080106/os/kernel$ s
```

checking testcases make all `CHAPTER=3_2` in user

```
akashili17@akashili17: ~/Documents/os/labs-2018080106/os/kernel
 -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o timer.o timer.c
riscv64-unknown-elf-gcc    -c -o link_app.o link_app.S
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel_app.ld -o kernel switch.o entry.o trampoline.o panic.o syscall
.o trap.o string.o batch.o console.o sbi.o main.o proc.o printf.o timer.o link_app.o
riscv64-unknown-elf-objdump -S kernel > kernel.asm
riscv64-unknown-elf-objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
qemu-system-riscv64 -nographic -smp 1 -machine virt -bios ../bootloader/rustsbi-qemu.bin -kernel kernel
[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
run app 0
run app 1
run app 2
run app 3
run app 4
run app 5
start scheduler!
priority = 7, exitcode = 3514000
proc 3 exit with 0
priority = 6, exitcode = 3251600
proc 2 exit with 0
priority = 9, exitcode = 4618400
proc 5 exit with 0
priority = 5, exitcode = 2280800
proc 1 exit with 0
priority = 8, exitcode = 4106000
proc 4 exit with 0
priority = 10, exitcode = 5204800
proc 6 exit with 0
panic: all apps over

akashili17@akashili17:~/Documents/os/labs-2018080106/os/kernel$
```

问答

考虑在一个完整的 os 中，随时可能有新进程产生，新进程在调度池中的位置见chapter5相关代码。

1. 请简要描述**chapter3示例代码**调度策略，尽可能多的指出该策略存在的缺点。

【解】The `idle` process continuously iterates through the process pool. Any process that is currently in the state of "runnable" is found, then starts to execute. After the execution, the process becomes idle and then the process continues to traverse the process pool. This strategy cannot execute multiple process in parallel, and can only move onto the next process when the process being run gives up, such as if the execution is too long (deadloop case). It also can not set different priority for the process.

2. 调度策略在公平性上存在比较大的问题，请找到一个进程产生和结束的时间序列，使得在该调度算法下发生：先创建的进程后执行的现象。你需要给出类似下面例子的信息（有更详细的分析描述更好，但尽量精简）。同时指出该序列在你实现的 **stride** 调度算法下顺序是怎样的？

时间	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
运行进程	-	p1	p2	p3	p4	p1	-	-
事件	p1、p2，p3产生	p1 结束	p2 结束	p3结束	p4产生	p4 结束	-	-

产生顺序：p1、p2、p3、p4。第一次执行顺序: p1、p2、p3、p4。

3. **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。

- 【解】Since 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，**如果不考虑溢出**，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。

- If you meet the required 'stride' at a certain moment, `STRIDE_MAX - STRIDE_MIN <= BigStride / 2`, then the corresponding process `STRIDE_MIN` should be executed at the next time slice. At his moment, `MAX_STRIDE' = max(MAX_STRIDE, 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 BIGSTRIDE = 0xffffffffffffffffULL;
bool Less(Stride_t, Stride_t) {
    if (x < y)
        return y - x <= BIGSTRIDE / 2;
    else
        return x - y > BIGSTRIDE / 2;
}
```