

## os-assignment1 实验报告

18364067 罗淦元

### 一. 实验要求

在 github 上创建 os-assignment1 项目，项目包括三个题目的结果：

1. 生产者消费者 (producer-consumer) 问题: prod.c 和 cons.c
2. 哲学家就餐 (dinning philosophers) 问题: dph.c
3. MIT S.081 实验: xv6 阅读和实验过程

该项目目录下，手工编写 Makefile，通过命令行下的 make all 命令，可以同时产生上述文件对应的可执行文件: prod, cons, dph; 通过单独的 make 目标，可以产生单独的 bin 文件。比如 make dph 可以单独生成 dph。提交之前，保证这三个程序均可正确执行，评分的时候将考察运行过程。

该目录下，同时存放 1 个 word 文件，按上述顺序包含实验内容报告，每个题目安排一大节。

### 1. (30 分) 生产者消费者问题

1. 需要创建生产者和消费者两个进程（注意：不是线程），一个 prod，一个 cons，每个进程有 3 个线程。两个进程之间的缓冲最多容纳 20 个数据。
2. 每个生产者线程随机产生一个数据，打印出来自己的 id（进程、线程）以及该数据；每个消费者线程取出一个数据，然后打印自己的 id 和数据。
3. 生产者和消费者这两个进程之间通过共享内存来通信，通过信号量来同步。
4. 生产者生成数据的间隔和消费者消费数据的间隔，按照负指数分布来控制，各有一个控制参数  $\lambda_p$ ,  $\lambda_c$
5. 运行的时候，开两个窗口，一个 ./prod  $\lambda_p$ ，另一个 ./cons  $\lambda_c$ ，要求测试不同的参数组合，打印结果，截屏放到作业报告里。

#### 1.1 实验结果

以下展示不同参数组合下，生产者进程（左）和消费者进程（右）的输出结果

- 1.1.1  $\lambda_p = 1$ ,  $\lambda_c = 1$  情况 ( $\lambda_p = \lambda_c$ )

```
10月17日 星期六, 17:20
dasquar@dasquar-VirtualBox: ~/os-assignment1

Thread 3315459840 of Process 209019 write data 27
Thread 3298674432 of Process 209019 write data 59
Thread 3315459840 of Process 209019 write data 26
Thread 3307607136 of Process 209019 write data 26
Thread 3307607136 of Process 209019 write data 36
Thread 3307607136 of Process 209019 write data 68
Thread 3307607136 of Process 209019 write data 29
Thread 3307607136 of Process 209019 write data 30
Thread 3307607136 of Process 209019 write data 23
Thread 3315459840 of Process 209019 write data 35
Thread 3315459840 of Process 209019 write data 2
Thread 3315459840 of Process 209019 write data 58
Thread 3298674432 of Process 209019 write data 67
Thread 3315459840 of Process 209019 write data 56
Thread 3298674432 of Process 209019 write data 42
Thread 3315459840 of Process 209019 write data 73
Thread 3298674432 of Process 209019 write data 19
Thread 3315459840 of Process 209019 write data 37
Thread 3298674432 of Process 209019 write data 24
Thread 3298674432 of Process 209019 write data 70
Thread 3298674432 of Process 209019 write data 26
Thread 3298674432 of Process 209019 write data 88
Thread 3315459840 of Process 209019 write data 73
Thread 3315459840 of Process 209019 write data 70
Thread 3315459840 of Process 209019 write data 81
Thread 3315459840 of Process 209019 write data 25
Thread 3307607136 of Process 209019 write data 27
Thread 3298674432 of Process 209019 write data 5
Thread 3307607136 of Process 209019 write data 29
Thread 3298674432 of Process 209019 write data 57
Thread 3307607136 of Process 209019 write data 95
Thread 3298674432 of Process 209019 write data 45
Thread 3307607136 of Process 209019 write data 67
Thread 3315459840 of Process 209019 write data 64
Thread 3307607136 of Process 209019 write data 84
Thread 3315459840 of Process 209019 write data 8
Thread 3307607136 of Process 209019 write data 78
Thread 3315459840 of Process 209019 write data 84
Thread 3315459840 of Process 209019 write data 51
Thread 3315459840 of Process 209019 write data 99
Thread 3298674432 of Process 209019 write data 68
Thread 3307607136 of Process 209019 write data 12
Thread 3298674432 of Process 209019 write data 86
Thread 3298674432 of Process 209019 write data 39
Thread 3298674432 of Process 209019 write data 78
Thread 3307607136 of Process 209019 write data 1
Thread 3315459840 of Process 209019 write data 2
Thread 1674237696 of Process 209707 get data 27
Thread 1682636400 of Process 209707 get data 59
Thread 1674237696 of Process 209707 get data 26
Thread 1682636400 of Process 209707 get data 26
Thread 1682636400 of Process 209707 get data 36
Thread 1674237696 of Process 209707 get data 68
Thread 1674237696 of Process 209707 get data 29
Thread 1682636400 of Process 209707 get data 30
Thread 1674237696 of Process 209707 get data 23
Thread 1682636400 of Process 209707 get data 35
Thread 1682636400 of Process 209707 get data 2
Thread 1674237696 of Process 209707 get data 58
Thread 1691823184 of Process 209707 get data 67
Thread 1682636400 of Process 209707 get data 56
Thread 1674237696 of Process 209707 get data 42
Thread 1691823184 of Process 209707 get data 73
Thread 1682636400 of Process 209707 get data 19
Thread 1674237696 of Process 209707 get data 37
Thread 1691823184 of Process 209707 get data 24
Thread 1682636400 of Process 209707 get data 79
Thread 1674237696 of Process 209707 get data 26
Thread 1682636400 of Process 209707 get data 73
Thread 1674237696 of Process 209707 get data 78
Thread 1691823184 of Process 209707 get data 81
Thread 1682636400 of Process 209707 get data 25
Thread 1674237696 of Process 209707 get data 27
Thread 1691823184 of Process 209707 get data 5
Thread 1682636400 of Process 209707 get data 29
Thread 1674237696 of Process 209707 get data 57
Thread 1691823184 of Process 209707 get data 95
Thread 1682636400 of Process 209707 get data 45
Thread 1674237696 of Process 209707 get data 67
Thread 1691823184 of Process 209707 get data 64
Thread 1682636400 of Process 209707 get data 59
Thread 1674237696 of Process 209707 get data 8
Thread 1691823184 of Process 209707 get data 78
Thread 1682636400 of Process 209707 get data 84
Thread 1674237696 of Process 209707 get data 51
Thread 1691823184 of Process 209707 get data 99
Thread 1682636400 of Process 209707 get data 68
Thread 1674237696 of Process 209707 get data 12
Thread 1682636400 of Process 209707 get data 86
Thread 1674237696 of Process 209707 get data 39
Thread 1691823184 of Process 209707 get data 78
Thread 1682636400 of Process 209707 get data 1
Thread 1674237696 of Process 209707 get data 2
Thread 1691823184 of Process 209707 get data 2
```

1.1.2  $\lambda_p = 1.234$ ,  $\lambda_c = 5.678$  情况 ( $\lambda_p < \lambda_c$ )

```
10月17日 星期六, 19:36
dasquar@dasquar-VirtualBox: ~/os-assignment1

sleep 0.756743 secondThread 501511936 of Process 212466 write data 21
sleep 0.362839 secondThread 518297344 of Process 212466 write data 95
sleep 0.860315 secondThread 501511936 of Process 212466 write data 37
sleep 1.377570 secondThread 518297344 of Process 212466 write data 93
sleep 0.036603 secondThread 518297344 of Process 212466 write data 28
sleep 1.791191 secondThread 501511936 of Process 212466 write data 11
sleep 2.865767 secondThread 518297344 of Process 212466 write data 29
sleep 3.884122 secondThread 509984640 of Process 212466 write data 4
sleep 2.336828 secondThread 501511936 of Process 212466 write data 63
sleep 1.892933 secondThread 501511936 of Process 212466 write data 38
sleep 0.179888 secondThread 501511936 of Process 212466 write data 40
sleep 2.157763 secondThread 509984640 of Process 212466 write data 18
sleep 1.848985 secondThread 509984640 of Process 212466 write data 88
sleep 0.992847 secondThread 518297344 of Process 212466 write data 17
sleep 1.286041 secondThread 501511936 of Process 212466 write data 96
sleep 0.396354 secondThread 509984640 of Process 212466 write data 43
sleep 0.077055 secondThread 509984640 of Process 212466 write data 83
sleep 0.253169 secondThread 509984640 of Process 212466 write data 99
sleep 0.744721 secondThread 501511936 of Process 212466 write data 25
sleep 0.328562 secondThread 518297344 of Process 212466 write data 98
sleep 1.270357 secondThread 501511936 of Process 212466 write data 39
sleep 0.637681 secondThread 509984640 of Process 212466 write data 86
sleep 2.888109 secondThread 501511936 of Process 212466 write data 82
sleep 0.418002 secondThread 501511936 of Process 212466 write data 64
sleep 0.578688 secondThread 518297344 of Process 212466 write data 7
sleep 0.423031 secondThread 518297344 of Process 212466 write data 4
sleep 1.791219 secondThread 501511936 of Process 212466 write data 11
sleep 1.125396 secondThread 501511936 of Process 212466 write data 28
sleep 0.057659 secondThread 518297344 of Process 212466 write data 43
sleep 1.245998 secondThread 509984640 of Process 212466 write data 68
sleep 0.972392 secondThread 501511936 of Process 212466 write data 22
sleep 0.534408 secondThread 501511936 of Process 212466 write data 10
sleep 1.128663 secondThread 509984640 of Process 212466 write data 1
sleep 0.426356 secondThread 518297344 of Process 212466 write data 30
sleep 0.920319 secondThread 509984640 of Process 212466 write data 5
sleep 1.312638 secondThread 501511936 of Process 212466 write data 36
sleep 1.348761 secondThread 518297344 of Process 212466 write data 26
sleep 2.537254 secondThread 509984640 of Process 212466 write data 65
sleep 1.058806 secondThread 501511936 of Process 212466 write data 16
sleep 0.760239 secondThread 509984640 of Process 212466 write data 30
sleep 0.361777 secondThread 501511936 of Process 212466 write data 37
sleep 1.158769 secondThread 509984640 of Process 212466 write data 24
sleep 0.117958 secondThread 509984640 of Process 212466 write data 36
sleep 0.830652 secondThread 518297344 of Process 212466 write data 99
sleep 1.998282 secondThread 509984640 of Process 212466 write data 59
sleep 1.495871 secondThread 501511936 of Process 212466 write data 71
sleep 0.399943 secondThread 501511936 of Process 212466 write data 31
sleep 4.158128 secondThread 518297344 of Process 212466 write data 30
sleep 1.756727 secondThread 509984640 of Process 212466 write data 94
sleep 0.115887 secondThread 2077165312 of Process 212476 get data 21
sleep 0.515325 secondThread 2066379984 of Process 212476 get data 95
sleep 0.101645 secondThread 2068772688 of Process 212476 get data 37
sleep 0.337228 secondThread 2066379984 of Process 212476 get data 93
sleep 0.230541 secondThread 2077165312 of Process 212476 get data 28
sleep 0.483761 secondThread 2068772688 of Process 212476 get data 11
sleep 0.331746 secondThread 2066379984 of Process 212476 get data 29
sleep 0.048625 secondThread 2077165312 of Process 212476 get data 4
sleep 0.680159 secondThread 2068772688 of Process 212476 get data 63
sleep 0.545467 secondThread 2066379984 of Process 212476 get data 38
sleep 0.031954 secondThread 2077165312 of Process 212476 get data 40
sleep 0.425831 secondThread 2068772688 of Process 212476 get data 18
sleep 0.260931 secondThread 2066379984 of Process 212476 get data 88
sleep 1.13110 secondThread 2068772688 of Process 212476 get data 17
sleep 0.193385 secondThread 2077165312 of Process 212476 get data 96
sleep 0.645996 secondThread 2066379984 of Process 212476 get data 43
sleep 0.388358 secondThread 2068772688 of Process 212476 get data 83
sleep 0.073477 secondThread 2077165312 of Process 212476 get data 99
sleep 1.104492 secondThread 2068772688 of Process 212476 get data 25
sleep 0.851264 secondThread 2066379984 of Process 212476 get data 98
sleep 0.041536 secondThread 2068772688 of Process 212476 get data 39
sleep 0.464658 secondThread 2077165312 of Process 212476 get data 66
sleep 1.62586 secondThread 2066379984 of Process 212476 get data 82
sleep 0.107743 secondThread 2068772688 of Process 212476 get data 64
sleep 0.637771 secondThread 2077165312 of Process 212476 get data 7
sleep 0.474610 secondThread 2066379984 of Process 212476 get data 28
sleep 0.621778 secondThread 2068772688 of Process 212476 get data 11
sleep 0.496638 secondThread 2068772688 of Process 212476 get data 30
sleep 1.16815 secondThread 2077165312 of Process 212476 get data 43
sleep 0.285820 secondThread 2068772688 of Process 212476 get data 68
sleep 0.836657 secondThread 2066379984 of Process 212476 get data 22
sleep 0.546686 secondThread 2077165312 of Process 212476 get data 10
sleep 0.236884 secondThread 2066379984 of Process 212476 get data 1
sleep 0.060973 secondThread 2077165312 of Process 212476 get data 26
sleep 0.277076 secondThread 2077165312 of Process 212476 get data 5
sleep 0.491983 secondThread 2066379984 of Process 212476 get data 36
sleep 0.060973 secondThread 2077165312 of Process 212476 get data 38
sleep 0.452381 secondThread 2068772688 of Process 212476 get data 65
sleep 0.175649 secondThread 2066379984 of Process 212476 get data 16
sleep 0.050652 secondThread 2077165312 of Process 212476 get data 37
sleep 0.633367 secondThread 2068772688 of Process 212476 get data 37
sleep 0.263465 secondThread 2066379984 of Process 212476 get data 24
sleep 0.026815 secondThread 2077165312 of Process 212476 get data 36
sleep 0.315462 secondThread 2066379984 of Process 212476 get data 99
sleep 0.635850 secondThread 2068772688 of Process 212476 get data 79
sleep 0.273354 secondThread 2077165312 of Process 212476 get data 51
sleep 0.815401 secondThread 2066379984 of Process 212476 get data 31
sleep 0.599563 secondThread 2077165312 of Process 212476 get data 30
sleep 0.010792 secondThread 2068772688 of Process 212476 get data 94
```

1.1.3  $\lambda_p = 5.678$ ,  $\lambda_c = 1.234$  情况 ( $\lambda_p > \lambda_c$ )

```
desquar@desquar-VirtualBox:~/os-assignment1
sleep 0.35638 secondThread 339298048 of Process 212482 write data 49
sleep 0.034516 secondThread 330905344 of Process 212482 write data 64
sleep 0.012089 secondThread 222512640 of Process 212482 write data 91
sleep 0.225278 secondThread 339298048 of Process 212482 write data 41
sleep 0.009079 secondThread 330905344 of Process 212482 write data 49
sleep 0.104087 secondThread 222512640 of Process 212482 write data 75
sleep 0.010990 secondThread 339298048 of Process 212482 write data 37
sleep 0.220115 secondThread 222512640 of Process 212482 write data 35
sleep 0.043795 secondThread 330905344 of Process 212482 write data 42
sleep 0.240643 secondThread 222512640 of Process 212482 write data 33
sleep 0.173896 secondThread 339298048 of Process 212482 write data 17
sleep 0.076862 secondThread 330905344 of Process 212482 write data 84
sleep 0.192571 secondThread 339298048 of Process 212482 write data 46
sleep 0.079165 secondThread 222512640 of Process 212482 write data 51
sleep 0.038020 secondThread 330905344 of Process 212482 write data 50
sleep 0.309095 secondThread 339298048 of Process 212482 write data 73
sleep 0.001887 secondThread 222512640 of Process 212482 write data 37
sleep 0.104462 secondThread 339298048 of Process 212482 write data 18
sleep 0.320593 secondThread 330905344 of Process 212482 write data 56
sleep 0.019803 secondThread 222512640 of Process 212482 write data 28
sleep 0.032139 secondThread 330905344 of Process 212482 write data 3
sleep 0.052245 secondThread 222512640 of Process 212482 write data 41
sleep 0.303360 secondThread 339298048 of Process 212482 write data 05
sleep 0.025839 secondThread 330905344 of Process 212482 write data 76
sleep 0.277490 secondThread 222512640 of Process 212482 write data 11
sleep 0.343966 secondThread 339298048 of Process 212482 write data 13
sleep 0.325813 secondThread 330905344 of Process 212482 write data 86
sleep 0.029972 secondThread 330905344 of Process 212482 write data 95
sleep 0.225761 secondThread 339298048 of Process 212482 write data 90
sleep 0.153811 secondThread 222512640 of Process 212482 write data 40
sleep 0.015111 secondThread 330905344 of Process 212482 write data 56
sleep 0.305020 secondThread 339298048 of Process 212482 write data 97
sleep 0.193587 secondThread 222512640 of Process 212482 write data 2
sleep 0.051179 secondThread 330905344 of Process 212482 write data 59
sleep 0.007965 secondThread 339298048 of Process 212482 write data 58
sleep 0.291779 secondThread 222512640 of Process 212482 write data 16
sleep 0.501249 secondThread 330905344 of Process 212482 write data 38
sleep 0.190289 secondThread 339298048 of Process 212482 write data 23
sleep 0.490684 secondThread 222512640 of Process 212482 write data 2
sleep 0.020801 secondThread 339298048 of Process 212482 write data 82
sleep 0.069976 secondThread 330905344 of Process 212482 write data 25
sleep 0.020876 secondThread 339298048 of Process 212482 write data 68
sleep 0.030458 secondThread 330905344 of Process 212482 write data 64
sleep 0.000770 secondThread 222512640 of Process 212482 write data 57
sleep 0.148836 secondThread 330905344 of Process 212482 write data 6
sleep 0.010142 secondThread 339298048 of Process 212482 write data 62
sleep 0.032120 secondThread 330905344 of Process 212482 write data 55
sleep 0.004202 secondThread 222512640 of Process 212482 write data 46
sleep 0.306357 secondThread 339298048 of Process 212482 write data 38

desquar@desquar-VirtualBox:~/os-assignment1
sleep 1.023077 secondThread 414009856 of Process 212486 get data 50
sleep 2.346805 secondThread 4154402560 of Process 212486 get data 12
sleep 0.279393 secondThread 4154402560 of Process 212486 get data 8
sleep 0.333328 secondThread 4162795264 of Process 212486 get data 69
sleep 0.052331 secondThread 4162795264 of Process 212486 get data 52
sleep 0.107291 secondThread 4162795264 of Process 212486 get data 29
sleep 0.092461 secondThread 4154402560 of Process 212486 get data 58
sleep 1.630811 secondThread 4162795264 of Process 212486 get data 67
sleep 0.866735 secondThread 414009856 of Process 212486 get data 95
sleep 0.947142 secondThread 4154402560 of Process 212486 get data 7
sleep 0.999714 secondThread 4162795264 of Process 212486 get data 54
sleep 0.895622 secondThread 414009856 of Process 212486 get data 46
sleep 0.242173 secondThread 4154402560 of Process 212486 get data 89
sleep 0.829083 secondThread 4162795264 of Process 212486 get data 19
sleep 0.959604 secondThread 414009856 of Process 212486 get data 85
sleep 1.034629 secondThread 4154402560 of Process 212486 get data 12
sleep 0.267469 secondThread 4162795264 of Process 212486 get data 49
sleep 0.215273 secondThread 4154402560 of Process 212486 get data 4
sleep 1.638864 secondThread 4162795264 of Process 212486 get data 57
sleep 2.358645 secondThread 414009856 of Process 212486 get data 49
sleep 2.292895 secondThread 4154402560 of Process 212486 get data 64
sleep 1.907807 secondThread 4162795264 of Process 212486 get data 91
sleep 0.273228 secondThread 4162795264 of Process 212486 get data 41
sleep 0.149918 secondThread 4162795264 of Process 212486 get data 49
sleep 0.003866 secondThread 4162795264 of Process 212486 get data 75
sleep 1.746313 secondThread 414009856 of Process 212486 get data 37
sleep 3.471786 secondThread 4154402560 of Process 212486 get data 35
sleep 2.548957 secondThread 4162795264 of Process 212486 get data 42
sleep 0.178802 secondThread 4162795264 of Process 212486 get data 33
sleep 0.991121 secondThread 4162795264 of Process 212486 get data 17
sleep 0.880032 secondThread 4154402560 of Process 212486 get data 84
sleep 1.632606 secondThread 414009856 of Process 212486 get data 46
sleep 0.761780 secondThread 4162795264 of Process 212486 get data 51
sleep 1.766022 secondThread 414009856 of Process 212486 get data 50
sleep 0.092025 secondThread 414009856 of Process 212486 get data 73
sleep 0.877387 secondThread 4154402560 of Process 212486 get data 37
sleep 2.699648 secondThread 414009856 of Process 212486 get data 18
sleep 1.124046 secondThread 4162795264 of Process 212486 get data 56
sleep 0.488402 secondThread 4162795264 of Process 212486 get data 28
sleep 0.714170 secondThread 414009856 of Process 212486 get data 3
sleep 0.343060 secondThread 414009856 of Process 212486 get data 41
sleep 0.018581 secondThread 414009856 of Process 212486 get data 85
sleep 0.107639 secondThread 4162795264 of Process 212486 get data 76
sleep 0.702705 secondThread 414009856 of Process 212486 get data 11
sleep 0.877177 secondThread 4162795264 of Process 212486 get data 13
sleep 1.209414 secondThread 4154402560 of Process 212486 get data 80
sleep 0.370732 secondThread 414009856 of Process 212486 get data 95
sleep 1.096334 secondThread 4154402560 of Process 212486 get data 99
sleep 2.438205 secondThread 4162795264 of Process 212486 get data 40
```

## 1.2 实验结果分析

当  $\lambda p \leq \lambda c$  时, cons 几乎同步输出 prod 产生的数据。说明两进程间通讯的进度主要受限于 prod 进程产生数据的速度, 这 and 实际代码相符。

而当  $\lambda p > \lambda c$  时, cons 进程消费的数据为 prod 进程在 BUFFER\_SIZE 个数据中产生最早的那个。说明两者通讯速度受限于 cons 消费速度, prod 进程在等待 cons 消耗后空出缓存区。

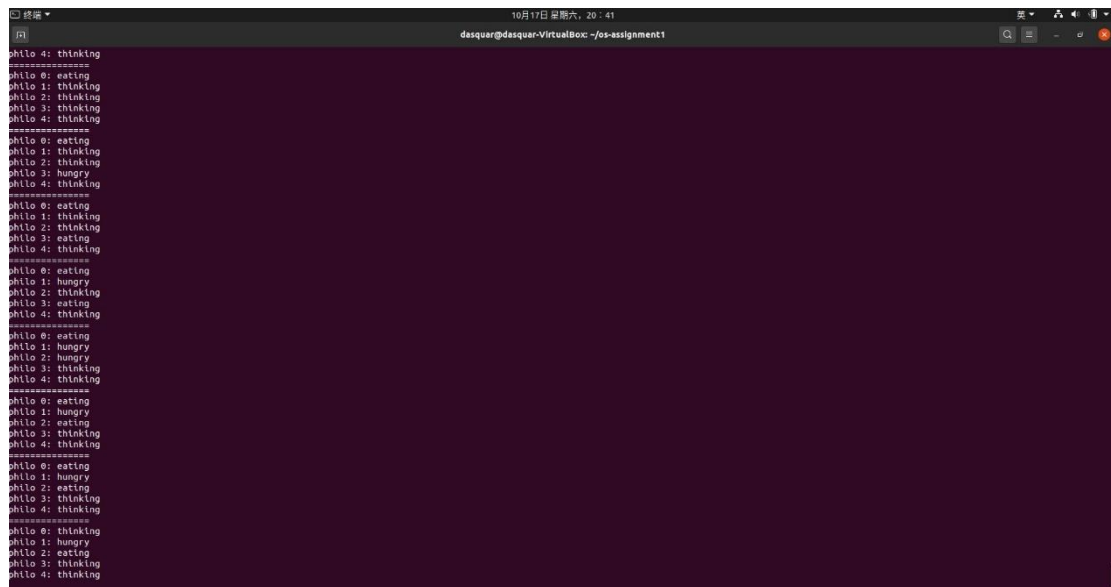
## 2. （20 分）哲学家就餐问题

参考课本（第十版）第 7 章 project 3 的要求和提示

1. 使用 POSIX 实现

5. 要求通过 make，能输出 dph 文件，输出哲学家们的状态。打印结果，截屏放到作业报告中。

### 2.1 实验结果



```
philo 4: thinking
*****
philo 0: eating
philo 1: thinking
philo 2: thinking
philo 3: thinking
philo 4: thinking
*****
philo 0: eating
philo 1: thinking
philo 2: thinking
philo 3: hungry
philo 4: thinking
*****
philo 0: eating
philo 1: thinking
philo 2: thinking
philo 3: eating
philo 4: thinking
*****
philo 0: eating
philo 1: hungry
philo 2: thinking
philo 3: eating
philo 4: thinking
*****
philo 0: eating
philo 1: hungry
philo 2: hungry
philo 3: thinking
philo 4: thinking
*****
philo 0: eating
philo 1: hungry
philo 2: eating
philo 3: thinking
philo 4: thinking
*****
philo 0: eating
philo 1: hungry
philo 2: eating
philo 3: thinking
philo 4: thinking
*****
philo 0: thinking
philo 1: hungry
philo 2: eating
philo 3: thinking
philo 4: thinking
```

### 2.2 实验分析

本实验中解决哲学家就餐问题的思路是，每当一名哲学家进入饥饿状态，就会“尝试就餐”。“尝试就餐”行为将检查其相邻两位哲学家是否在就餐。如果两者都在非就餐状态，就马上将自身状态转为“就餐”。否则，就会将自身挂起等待信号 `self[i]` 并释放 `mutex[i]` 锁。

```
1. void pickup_forks(int i) {
2.     state[i] = HUNGRY;
3.     tryeat(i);
4.     pthread_mutex_lock(&mutex[i]);
5.     while(state[i] != EATING) {
6.         pthread_cond_wait(&self[i], &mutex[i]);
7.     }
8.     pthread_mutex_unlock(&mutex[i]);
9. }
```

而当一名哲学家完成“就餐”，他有义务通知相邻的两名哲学家他已完成就餐，并使得他们再次“尝试就餐”，同时自身回到思考状态。

```
1. void return_forks(int i) {
2.     state[i] = THINKING;
```

```
3.  printf("philosopher %d says he has finished dinner\n", i);
4.  tryeat((i+1)%5);
5.  tryeat((i+4)%5);
6. }
```

“尝试就餐”行为的源代码如下。

```
1. void tryeat(int i) {
2.  printf("philosopher %d try to eat\n", i);
3.  if (state[i] == HUNGRY && (state[(i+4)%5] != EATING) && state[(i+1)%5] != EATING) {
4.      pthread_mutex_lock(&mutex[i]);
5.      state[i] = EATING;
6.      pthread_cond_signal(&self[i]);
7.      pthread_mutex_unlock(&mutex[i]);
8.  }
9.  else printf("philosopher %d fail to eat\n", i);
10. }
```

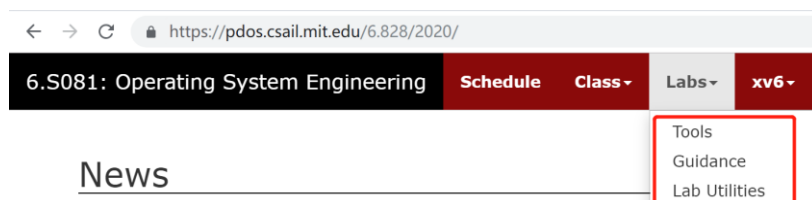
显然，当哲学家  $i$  尝试调用相邻哲学家的 `tryeat` 函数时，如果满足条件，将会夺取其下标对应的互斥锁 `mutex[i2]` ( $i_2$  为相邻哲学家下标) 避免在此过程中其他哲学家更改该哲学家状态 `state[i2]`，同时使  $i_2$  哲学家进入进餐状态并通知其解除挂起等待，释放互斥锁。这样就使得哲学家们能够有序就餐。

如实验结果所示，这种处理方式能够有效避免两名哲学家争夺筷子导致的“死锁”问题。同时具有一个缺陷，就是如果一名哲学家左右的两名哲学家恰好交替进行就餐行为，将会导致当前哲学家被“饿死”。这是本算法无法避免地问题。



### 3. (50 分) MIT 6.S081 课程实验

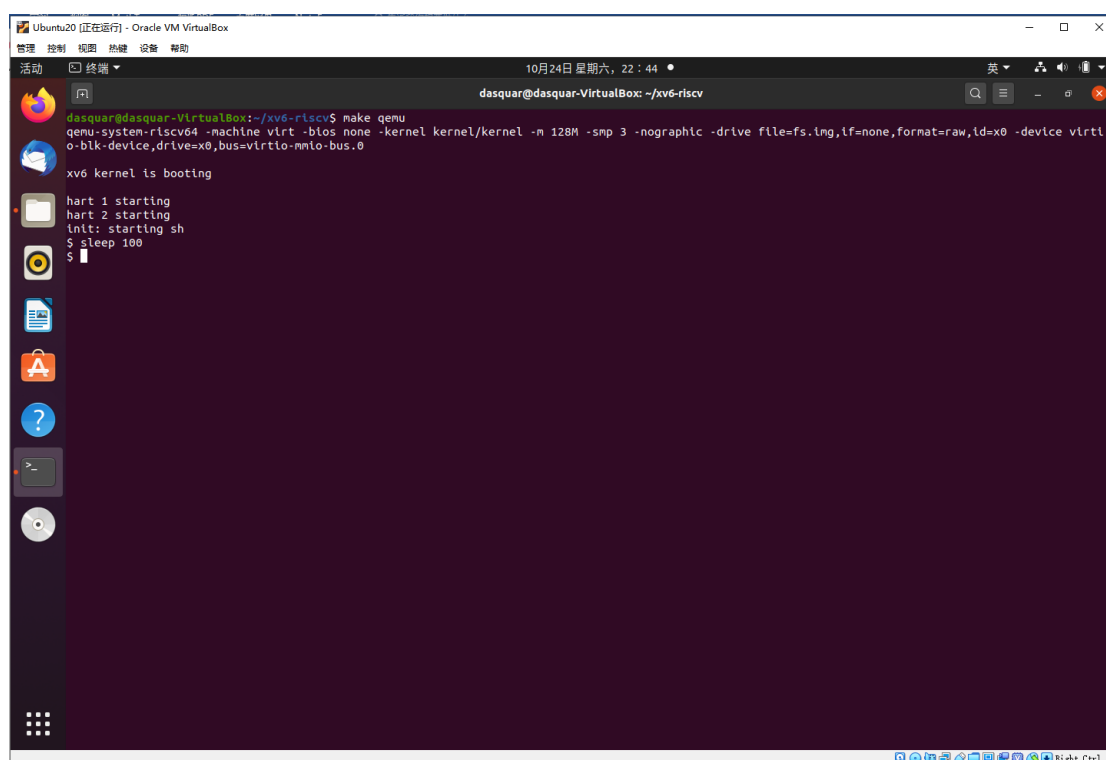
1. (20 分) 阅读 MIT 6.S081 [项目介绍](#), 如下图; 完成 xv6 的安装和启动 (Ctrl+X 可退出); 完成 Lab: Xv6 and Unix utilities 中的 sleep (easy) 任务, 即在 user/下添加 sleep.c 文件。在报告中提供 sleep.c 的代码, 并提供 sleep 运行的屏幕截图。提示: 在 vmware 下安装 ubuntu20, 可以较为顺利完成 xv6 安装和编译。



#### 3.1 实验结果

源代码详见附件。

运行结果如图所示。



可以看到, 当 sleep 指令没有参数时, 将会向用户抛出异常。当收到参数时, 首先将其转为数字, 然后调用根据 user/user.h 内给出的用户程序可用的 sleep 函数, 进一步调用 system call 并最后使用汇编完成 sleep 操作。需要注意在代码最后要使用 exit() 退出程序。通过编辑 Makefile 中的 UPROGS, 使得程序成为 xv6 sleep 指令指向的代码。

3.2 (30 分) 结合 [xv6 book](#) 第 1、2、7 章, 阅读 xv6 内核代码(kernel/目录下)的进程和调度相关文件, 围绕 `swtch.S`, `proc.h/proc.c`, 理解进程的基本数据结构, 组织方式, 以及调度方法。提示: 用 `source insight` 阅读代码较为方便。

- a) 修改 `proc.c` 中 `procdump` 函数, 打印各进程的扩展信息, 包括大小(多少字节)、内核栈地址、关键寄存器内容等, 通过 `^p` 可以查看进程列表, 提供运行屏幕截图。

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
=====
PID: 1 State: sleep Name: init
Kernal Stack Address: 0x0000003fffffd000 Process Memory Size(Byte): 0x0000000080008358
context: ra 0x000000008000204e sp 0x0000003fffffdef0
s0 0x0000003fffffd20 s1 0x00000000800116a8 s2 0x0000000080011290
s3 0x0000000000000001 s4 0x0000000000000004 s5 0x0000000000000001
s6 0x0000000000000000 s7 0x0000000000000000 s8 0x00000000800116a8
s9 0x0505050505050505 s10 0x0505050505050505 s11 0x0505050505050505
=====
PID: 2 State: sleep Name: sh
Kernal Stack Address: 0x0000003fffffb000 Process Memory Size(Byte): 0x0000000080008358
context: ra 0x000000008000204e sp 0x0000003fffffbe80
s0 0x0000003fffffb00 s1 0x0000000080011810 s2 0x0000000080011290
s3 0x0000000000000001 s4 0x0000000000000001 s5 0x0000000000003f2f
s6 0x0000000000000001 s7 0x0000000000000001 s8 0x0505050505050505
s9 0x0000000000000004 s10 0xffffffffffffffff s11 0x000000000000000a
```

- b) 在报告中, 要求逐行对 `swtch.S`, `scheduler(void)`, `sched(void)`, `yield(void)` 等函数的核心部分进行解释, 写出你对 xv6 中进程调度框架的理解。阐述越详细、硬件/软件接口部分理解越深, 评分越高。

### 3.2.1 `proc.h`

作为开始, 首先要在 `proc.h` 中了解一些要用到的数据结构的定义。

#### 1. `context` 上下文结构

1. //整个 `context` 结构为当前进程的上下文, 用于进程切换时保存状态和恢复状态
2. `struct context` {
3. `uint64 ra;` // `ra` 寄存器, 存储了当前进程的 `pc` 值
4. `uint64 sp;` //堆栈寄存器值
- 5.
6. // `callee-saved`
7. `uint64 s0;` //一些易失性寄存器的值
8. `uint64 s1;`
9. `uint64 s2;`
10. `uint64 s3;`
11. `uint64 s4;`
12. `uint64 s5;`
13. `uint64 s6;`

```

14. uint64 s7;
15. uint64 s8;
16. uint64 s9;
17. uint64 s10;
18. uint64 s11;
19. };

```

## 2. CPU 结构

```

1. // 每个 cpu 的状态
2. struct cpu {
3.     struct proc *proc;    // 在当前 cpu 上运行的进程的指针
4.     struct context context; // scheduler 的 context, switch 函数通过
    进入 scheduler 的上下文来转换进程
5.     int noff;             // push_off 的深度
6.     int intena;           // push_off 前中断的开启状态
7. };

```

## 3. Proc 结构

```

2. // 每个进程的状态
3. struct proc {
4.     struct spinlock lock; // 自旋锁
5.
6.     // p->lock must be held when using these:
7.     enum procstate state; // 进程状态, 枚举为 UNUSED、SLEEPING、
    RUNNING 等
8.     struct proc *parent;  // 父进程指针
9.     void *chan;           // 非空则在 chan 指针上睡眠, 相同 chan 的进程将在
    wake 时一同唤醒
10.    int killed;            // 如果非零, 进程已被杀死
11.    int xstate;            // 退出时返回父进程的返回值
12.    int pid;              // 进程 id
13.
14.    // these are private to the process, so p->lock need not be held.
15.    uint64 kstack;         // 内核栈虚拟地址
16.    uint64 sz;             // 进程内存大小 (byte 为单位)
17.    pagetable_t pagetable; // 用户页表
18.    struct trapframe *trapframe; // trampoline.S 的数据页
19.    struct context context; // 进程上下文
20.    struct file *ofile[NOFILE]; // 打开的文件
21.    struct inode *cwd;      // 最近目录
22.    char name[16];         // 进程名, 用于 debug
23. };

```



### 3.2.2 swtch.S

swtch 的主要作用用于交换 old 和 new 进程的上下文。它将进程的上下上下文压入到内核栈中,并加载 new 进程先前压入的上下文。这里可以看出,当进程在执行时,其内核栈上下文总是空的,进程在使用用户栈

```
1. # Context switch
2. #
3. # void swtch(struct context *old, struct context *new);
4. #
5. # Save current registers in old. Load from new.
6.
7.
8. .globl swtch          # .global 关键字用来让一个符号对链接器可见,
                        # 可以供其他链接对象模块使用。
9. swtch:                # 第一步, 保存 old 进程的上下文。a0 传递了 old
                        # 指针, a1 传递了 new 指针。
10.    sd ra, 0(a0)        # 保存 ra 寄存器的值, 即 pc 值, 到 a0 指向位置
                        # 的第 0 位起的 8 位, 便于重新调度时返回到 swtch 前的状态
11.    sd sp, 8(a0)        # 保存堆栈寄存器的值, 即存放栈的偏移地址, 到
                        # a0 指向地址的第 8 位起 8 位。后面同理
12.    sd s0, 16(a0)       # 保存 12 个易失性寄存器的值, 如果这些值需要
                        # 在调用返回时恢复的话。
13.    sd s1, 24(a0)
14.    sd s2, 32(a0)
15.    sd s3, 40(a0)
16.    sd s4, 48(a0)
17.    sd s5, 56(a0)
18.    sd s6, 64(a0)
19.    sd s7, 72(a0)
20.    sd s8, 80(a0)
21.    sd s9, 88(a0)
22.    sd s10, 96(a0)
23.    sd s11, 104(a0)
24.                        # 从 a1, 即 new 指针指向的地址上获取保存的上下文
25.    ld ra, 0(a1)         # 从 a1 第 0 位起读取 8 位保存到值 ra 寄存器, 恢
                        # 复新进程的 pc 值
26.    ld sp, 8(a1)        # 同理, 恢复堆栈寄存器值
27.    ld s0, 16(a1)       # 同理, 恢复 12 个易失性寄存器值
28.    ld s1, 24(a1)
29.    ld s2, 32(a1)
30.    ld s3, 40(a1)
31.    ld s4, 48(a1)
```

```

32.    ld s5, 56(a1)
33.    ld s6, 64(a1)
34.    ld s7, 72(a1)
35.    ld s8, 80(a1)
36.    ld s9, 88(a1)
37.    ld s10, 96(a1)
38.    ld s11, 104(a1)
39.
40.    ret

```

### 3.2.3 yield() 函数（由于定时器中断进入进程调度）

```

1.  void
2.  yield(void)
3.  {
4.
5.      struct proc *p = myproc(); //myproc 函数获取到当前 cpu 上的进程
6.      acquire(&p->lock); //请求获得自旋锁
7.      p->state = RUNNABLE; //设置进程为可执行状态
8.      sched(); //调用 sched
9.      release(&p->lock); //释放自旋锁
10. }

```

### 3.2.4 sched() 函数

```

1.  void
2.  sched(void)
3.  {
4.      int intena;
5.      struct proc *p = myproc(); //获取当前 cpu 上的进程
6.
7.      /*检查当前进程是否持有自旋锁、push_off 深度是否不为 1、进程是否
        回到 RUNNABLE 状态以及是否关闭中断*/
8.      if(!holding(&p->lock))
9.          panic("sched p->lock");
10.     if(mycpu()->noff != 1)
11.         panic("sched locks");
12.     if(p->state == RUNNING)
13.         panic("sched running");
14.     if(intr_get())
15.         panic("sched interruptible");
16.
17.     intena = mycpu()->intena; //由于中断是内核线程的权利而不是 cpu,
        因此保存其 intena

```

```

18. swtch(&p->context, &mycpu()->context); //从当前进程的上下文转
    换到 scheduler 的上下文
19. mycpu()->intena = intena; //恢复 intena
20. }

```

### 3.2.5 scheduler 函数

```

4. void
5. scheduler(void)
6. {
7.     struct proc *p;
8.     struct cpu *c = mycpu(); //获取当前 cpu 结构, 在 start 时
9.                               //已经通过对每个 cpu 的 tp 寄存器赋值
10.                               //来作为每个 cpu 的 id, 并映射到 cpu 结构
    数组
11.
12.     c->proc = 0; //将当前 cpu 上的进程设为空
13.     for(;;) { //显然是个死循环, 将会不断遍历进程表来查找是否有
        RUNNABLE 的进程, 并切换到该进程
14.         //确保设备中断打开避免死锁.
15.         intr_on(); //打开中断
16.
17.         for(p = proc; p < &proc[NPROC]; p++) {
18.             acquire(&p->lock) //进程 p 请求获得自旋锁
19.             if(p->state == RUNNABLE) { //如果 p 是可运行状态
20.                 //进入进程 p。进程 p 需要释放自旋锁和重新请求自旋锁来返回
                    scheduler
21.                 p->state = RUNNING; //将 p 进程状态设为运行
22.                 c->proc = p; //并将 cpu 上进程设为 p
23.                 swtch(&c->context, &p->context); //交换 p 和 scheduler 上
                    下文。开始执行进程 p
24.
25.                 // Process is done running for now.
26.                 // It should have changed its p->state before coming back.
27.                 c->proc = 0; //清空当前 cpu 上的进程
28.             }
29.             release(&p->lock); //进程 p 释放自旋锁
30.         }
31.     }
32. }

```

### 3.2.6 sleep 和 wakeup

```

1. // 在 chan 上 sleep 和 wake 的原子操作
2. // 被唤醒时从新索取锁
3. void

```

```

4. sleep(void *chan, struct spinlock *lk)
5. {
6.     struct proc *p = myproc();
7.
8.     // 必须获取 p 上的自旋锁
9.     // 以更改 p 的 state 并调用 sched
10.    // 只要持有当前进程锁，就能保证不会错过 wakeup
11.    // 因为 wakeup 将索取进程锁，所以可以释放 lk
12.    if (lk != &p->lock) {
13.        acquire(&p->lock);
14.        release(lk);
15.    }
16.
17.    // 开始在 chan 上睡眠
18.    p->chan = chan;
19.    p->state = SLEEPING;
20.
21.    sched(); //调用 sched
22.
23.    //清空 chan
24.    p->chan = 0;
25.
26.    // 重新获得 lk 锁
27.    if (lk != &p->lock) {
28.        release(&p->lock);
29.        acquire(lk);
30.    }
31.}
32.
33.// 唤醒在 chan 上睡眠的所有进程，也就是多个进程等待一个事件的发生
34.// 必须在不持有任何进程中自旋锁的前提下调用
35.void
36.wakeup(void *chan)
37.{
38.    struct proc *p;
39.
40.    //查找睡眠中的进程，并将它设为 RUNNABLE
41.    //可以看到，进程在结束睡眠后并不是离开开始执行的
42.    //而要进入到进程表中重新等待调度
43.    for (p = proc; p < &proc[NPROC]; p++) {
44.        acquire(&p->lock);
45.        if (p->state == SLEEPING && p->chan == chan) {
46.            p->state = RUNNABLE;

```

```

47. }
48. release(&p->lock);
49. }
50. }

```

### 3.2.7

基于上文对于 `swtch.S`, `scheduler(void)`, `sched(void)`, `yield(void)` 等函数的注释, 我们开始分析整个 xv6 系统的进程调度过程。

#### 1. 自旋锁

对于多处理器的系统来说, 不能采用在标志寄存器中打开、关闭中断标志位的方式来防止中断处理中的并发, 因此就需要因此 `spinlock` 自旋锁这个结构来解决。从保证临界区访问原子性目的来考量, 自旋锁应该阻止包括中断、内核抢占和其他处理器对同一临界区的访问。我们查看 `spinlock` 结构可以看到, 里面主要包括了 `locked` 属性, 用于表示是否上锁; `cpu` 属性, 用于获得持有该锁的 `cpu`。

```

1. struct spinlock {
2.     uint locked;    // Is the lock held?
3.
4.     // For debugging:
5.     char *name;     // Name of lock.
6.     struct cpu *cpu; // The cpu holding the lock.
7. };

```

#### 2. 请求和释放

相关的两个重要函数就是 `acquire()` 和 `release()`。在此之前要先分析 `push_off()` 和 `pop_off()` 函数。

```

1. void
2. push_off(void)
3. {
4.     int old = intr_get(); // 获取当前中断使能状态
5.
6.     intr_off();           // 关闭中断
7.     if(mycpu()->noff == 0) // 如果这是第一层 push_off
8.         mycpu()->intena = old; // 在当前 cpu 的 intena 参数中存储
                                // push_off 前的中断使能状态
9.     mycpu()->noff += 1;
10. }

```

其中, `intr_get()` 通过内嵌汇编获取 `SIE` 寄存器的值, 判断中断使能状态。而 `intr_off()` 同样通过内嵌汇编向 `sstatus` 寄存器写入来关闭中断使能。



至于 `pop_off()` 函数则于 `push_off()` 函数相反。

```
1. void
2. pop_off(void)
3. {
4.     struct cpu *c = mycpu(); //获取当前 cpu
5.     if(intr_get()) //如果当前中断使能打开, 说明出现异常
6.         panic("pop_off - interruptible");
7.     if(c->noff < 1) //如果当前 push_off() 深度小于 1, 说明异常
8.         panic("pop_off");
9.     c->noff -= 1; //cpu 的 push_off() 深度-1
10.    if(c->noff == 0 && c->intena) //如果 push_off() 深度为 0 且
        push_off() 前中断打开, 则打开中断
11.    intr_on();
12. }
```

每执行一层 `push_off`, 就对应于执行一层 `pop_off`。只有当所有 `pop_off()` 执行完成, 即 `noff == 0`, `cpu` 才能够恢复到 `push_off` 前的中断使能状态。

我们回到 `acquire` 函数。其中 `holding` 函数返回当前 `cpu` 是否已经持有当前锁。

```
1. // 请求锁并在获得前自旋
2. void
3. acquire(struct spinlock *lk)
4. {
5.     push_off(); // push_off 一次, 关闭中断
6.     if(holding(lk)) //如果已经持有该锁, 再次请求将导致死锁, 抛出异常
7.         panic("acquire");
8.
9.     // 在 RISC-V, sync_lock_test_and_set 用于进行原子交换操作
10.    while(__sync_lock_test_and_set(&lk->locked, 1) != 0) //成功获取到锁前保持自旋
11.        ;
12.    //告诉 C 编译器和进程不要把加载和存储操作移到本点后
13.    //保证临界区内存的操作严格在锁被请求后执行
14.    __sync_synchronize();
15.
16.    // 记录持有锁的 cpu 方便 debug
17.    lk->cpu = mycpu();
18. }
```

`Realse` 函数也很简单。

```

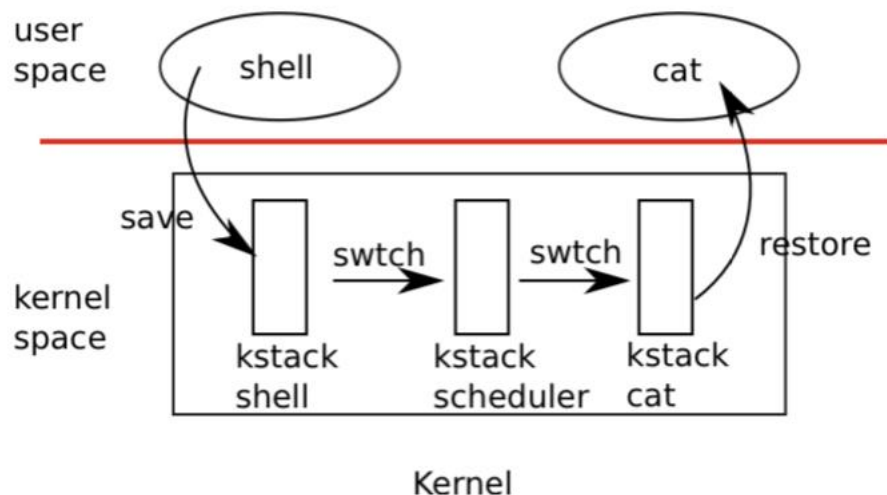
1. // 释放锁
2. void
3. release(struct spinlock *lk)
4. {
5.     if(!holding(lk)) //没有获取就释放, 抛出异常
6.         panic("release");
7.
8.     lk->cpu = 0; //清空锁上的 cpu
9.
10.    __sync_synchronize();
11.
12.    //释放锁, 等效于 lk->locked=0。
13.    //不使用 C 因为 C 标准可能用多个存储指令实现。
14.    __sync_lock_release(&lk->locked);
15.
16.    pop_off(); //进行 pop_off
17.}

```

进行了一些基本操作的解释后, 我们可以开始分析 xv6 的进程调度过程了。对于单个的 CPU, scheduler 是主要的函数。CPU 初始化后将会不断循环从进程队列中选择进程并执行。这一点在前文的注释中已经详细分析了 scheduler 的运行方式。

进程回到 sched() 的方式有多种。进程结束时, 将执行 exit() 函数, exit() 函数的结尾调用了 sched() 函数, 并由 sched() 函数将控制权移交给 scheduler。而 sleep 函数同样具有主动切换到调度器的功能。除此之外, 定时中断导致的 yield 函数, 也同样会调用 sched 函数来返回到 scheduler。

总的来说, 一个进程在执行过程中, 通过 wait、sleep、exit 和定时中断等系统调用和中断, 主动或被动地进入到 sched 函数中, sched 函数负责调用 swtch 函数将 old 进程的上下文保存到内核堆栈中, 并将控制权交换给 scheduler 调度器。



Schedular 调度器对于每个 CPU 都是专有的, 有着独立的寄存器和堆栈, 以免被其他 CPU 访问到。Schedular 调度器通过遍历进程表并获取第一个准备好运行的进程 new, 然后索取自旋锁避免多个 cpu 同时尝试切换同一进程 new 导致的竞争, 接着调用 swtch 从内核栈中恢复 new 进程先保存的上下文, 并从该上下文中 ra 指针指向的位置开始继续运行 new 进程 (因为 new 进程在此之前可能执行过一段时间后被中断或处于等待而切换到其他进程)。进程如果要主动放弃 CPU, 则必须请求其进程的自旋锁, 释放持有的其他锁, 更新自己的状态并主动调用 sched() (例如在程序结束时, 调用 exit() 中的 sched())。而当一个程序长时间地执行时, 将会由计时器中断调用 yield() 执行类似的过程。Sched 中会检查每次程序进入是否满足进程切换的条件, 并且调用 scheduler 继续上述循环。

### 3.3 对照 Linux 的 CFS 进程调度算法, 指出 xv6 的进程调度有何不足; 设计一个更好的进程调度框架, 可以用自然语言 (可结合伪代码) 描述, 但不需要编码实现。

我们可以看到, xv6 进程调度主要是使用轮询的方法实现。调度器每次都从进程表中选取第一个状态为 Runnable 的进程执行。而 Linux 的 CFS 进程调度算法, 其关键在于给每个进程的权重分配运行时间。

简化来说, CFS 中进程在调度周期中占用的时间与进程权重与所有进程权重和有关。同时每个进程还有一个虚拟运行时间 vruntime 的量, 其满足  $Vruntime = \text{实际运行时间} * NI$  值为 0 的进程权重 / 进程权重 = (调度周期 \* 进程权重 / 总权重) \* NI 值为 0 的进程的权重 / 进程权重 = 调度周期 \* NI 值为 0 的进程权重 / 总权重。即是说, 对于实际运行时间相同的进程, 权重小的 Vruntime 增长更快。但总体上来说, 每个进程的 Vruntime 又是同时推进的。根据这一点, 选取 vruntime 小的进程来执行, 一方面可以保证程序运行时间在虚拟时钟上是公平的, 同时宏观上来说优先级高的程序总是有着更长的实际运行时间。

和 CFS 比起来, xv6 的进程调度缺少对于每个进程的优先级别的判断, 一方面导致了部分紧急的任务无法被优先执行。

CFS 算法本身就是一个更好调度框架的例子。同时也是一种公平调度算法。整个调度器需要维护一个以上述 vruntime 组成的红黑树, 其最左节点即 vruntime 最小的节点就是优先级最高的节点。每次进入调度器时, 就可以加锁并将红黑树中最左节点设为运行状态再释放锁, 同时向进程提供一个与 A 调度周期 \* 进程权重 / 总权重的时间片, 并在进程执行完该时间片后中断并上下文切换回到调度器。而当进程调用 sleep、exit 函数时, 则从红黑树中删除该节点, 直到其重新进入就绪状态。通过这种方式, 一方面能够保证每个进程在一个 CPU 的调度周期中都会被执行一段时间, 使得短时间优先级低的进程也能被很快地完成。同时通过对 nice 值等优先级的设定, 可以确保紧急进程更快地完成任务。