进程调度器

对于调度器，一般概括为：

- 一个 CPU 同时只运行一个进程
- 在同时运行多个进程时，每个进程都会获得适当的时长，**轮流**在 CPU 上执行 (轮流的具体方式下面会通过实验的方式得到)

这里指出，Linux 会将多核 CPU 的每个核心是做一个 CPU (这里称呼其为 **逻辑CPU**)

1. 设计实验程序

在同时运行一个或多个一味消耗 CPU 时间执行处理的进程时，采集：

- 在某一时间点运行在逻辑 CPU 上的进程是哪一个
- 每个进程的运行进度

> 实验要求：

- 命令行参数
  - 第 1 个参数 (n)：同时运行的进程数量
  - 第 2 个参数 (total)：程序运行的总时长（单位：毫秒）
  - 第 3 个参数 (resol)：采集统计信息的间隔（单位：毫秒）
  - 令 n 个进程同时运行，然后在全部进程都结束后结束程序的运行。各个进程按照以下要求运行：
    - 在消耗 total 毫秒的 CPU 时间后结束运行
    - 每 resol 毫秒记录以下 3 个数值：
      - 每个进程唯一的 ID（0 ~ n - 1 各个进程独有的编号）
      - 从程序开始运行到记录的时间点为止经过的时间（单位：毫秒）
      - 进程的进度（单位：%）

- 在结束运行后，把所有统计信息用制表符分隔并逐行输出

> 实验代码 (sched.cpp)

```cpp
#include <sys/types.h>
#include <sys/wait.h>
#include <bits/stdc++.h>
#include <unistd.h>
#include <err.h>
#include <time.h>
#include <memory>

constexpr uint64_t NLOOP_FOR_ESTIMATION = 1000000000ULL;
constexpr uint64_t NSECS_PER_MSEC = 1000000ULL;
constexpr uint64_t NSECS_PER_SEC = 1000000000ULL;

// timespec {tv_sec, tv_nsec} 秒数，纳秒
constexpr uint64_t calc(timespec t) {
    return t.tv_sec * NSECS_PER_SEC + t.tv_nsec;
}

static inline int64_t diff_nsec(timespec before, timespec after) {
    return calc(after) - calc(before);
}

static uint64_t loops_per_msec() {
    timespec before, after;
    clock_gettime(CLOCK_MONOTONIC, &before);

    // 执行空循环测试循环指定次数，需要的 CPU 运行时间
    for (uint64_t i = 0;i < NLOOP_FOR_ESTIMATION;i ++);

    clock_gettime(CLOCK_MONOTONIC, &after);

    // 返回每 ms 运行的循环的次数
    return NLOOP_FOR_ESTIMATION * NSECS_PER_MSEC / diff_nsec(before, after);
}

static inline void load(uint64_t nloop) {
```

```
    for (uint64_t i = 0;i < nloop;i ++ );
}

FILE *out1, *out2;
static void child_fn(int id, std::shared_ptr<timespec[]> buf, int nrecord,
uint64_t nloop_per_resol, \
        timespec start, FILE* out1, FILE* pic2) {

    for (int i = 0;i < nrecord;i ++ ) {
        timespec ts;
        load(nloop_per_resol);// 模拟每个 resol 收集一次数据
        clock_gettime(CLOCK_MONOTONIC, &ts);
        buf[i] = ts;
    }
    // 进程id & 每个resol距开始的 ms 数 & 运行进度
    for (int i = 0;i < nrecord;i ++ ) {
        // printf("%d\t%ld\t%d\n", id, diff_nsec(start, buf[i]) /
NSECS_PER_MSEC, \
        //     (i + 1) * 100 / nrecord);
        int finish_rate = (i + 1) * 100 / nrecord;
        uint64_t cost_time = diff_nsec(start, buf[i]) / NSECS_PER_MSEC;
        fprintf(out2, "%llu %d\n", cost_time, finish_rate);
        fprintf(out1, "%llu %d\n", cost_time, id);
    }
    exit(EXIT_SUCCESS);
}

std::shared_ptr<pid_t[]> pids;


// args = 命令行参数 + 1
int main(int args, char *argv[]) {
    out1 = fopen("tmp1.txt", "w"); // 打开名为tmp1.txt的文件，以写入模式打开
    out2 = fopen("tmp2.txt", "w"); // 打开名为tmp2.txt的文件，以写入模式打开
    if (args !=  4) {
        fprintf(stderr, "usage: %s <nproc> <total[ms]> <resolution[ms]>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    int nproc = atoi(argv[1]);
    int total = atoi(argv[2]);
    int resol = atoi(argv[3]);

    // std::cout << "pars: " << nproc << " " << total << " " << resol <<
std::endl;
     // 下面是对于命令行参数错误使用的输出
    std::vector<std::string> names = {"nproc", "total", "resol"};
    for (int i = 0;i < 3;i ++ ) {
        int x = atoi(argv[i + 1]);
        if (x < 1) {
            fprintf(stderr, "<%s>(%d) should be >= 1\n", names[i].c_str(),
x);
            exit(EXIT_FAILURE);
        }
    }

    if (total % resol) {
        fprintf(stderr, "<total>(%d) should multiple of <resolution>(%d)\n",
total, resol);
        exit(EXIT_FAILURE);
    }

    int nrecord = total / resol; // 收集信息的次数
     // 申请动态的 timespec 结构体指针
    std::shared_ptr<timespec[]> logbuf(new timespec[nrecord]);

    if (!logbuf) {
        err(EXIT_FAILURE, "new(logbuf) failed");
    }

    puts("estimating workload which takes just one milisecond");
    uint64_t nloop_per_resol = loops_per_msec() * resol;
    puts("end estimation");
    fflush(stdout);

    pids = std::shared_ptr<pid_t[]>(new pid_t[nproc]);
    if (!pids) {
        warn("new(pids) failed");
    }
```

```
    timespec start;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0;i < nproc;i ++ ) {
        pids[i] = fork();
        if (pids[i] < 0) {
            // fork() 失败，代表实验失败，父进程杀掉所有子进程
            for (int j = 0;j < i;j ++ )
                if (kill(pids[i], SIGINT) < 0)
                    warn("kill (%d) failed", pids[i]);
            // 等待子进程全部退出，确保父进程回收所有子进程的资源
            for (int j = 0;j < i;j ++ )
                if (wait(NULL) < 0)
                    warn("wait() failed.");
            exit(EXIT_FAILURE);
        } else if (pids[i] == 0) {
             // 子进程
             child_fn(i, logbuf, nrecord, nloop_per_resol, start, out1,
out2);
             // 此前该个子进程已经结束，要么在 wait(NULL) 后被收回资源，要么等待父
进程收回
        }

    }

     // 运行到这，说明成功创建了 nproc 个并行子进程
    for (int i = 0;i < nproc;i ++ )
        if (wait(NULL) < 0)
            warn("wait() failed.");
    ;
    exit(EXIT_SUCCESS);
}
```

> 上述代码主要是对于 `wait(NULL)` 的理解，每个 `wait(NULL)` 唯一对应一个子进程(多出的 wait(NULL)，会响应一个不存在的进程 ID -1，参考 text.cpp)，它既可以响应已经运行完的自己子进程 (参考 `kill.cpp`)，

编译并且运行后，产生两个文件 `tmp1.txt`，`tmp2.txt`，分别存储：

- 每个 **resol** 统计的数据：x 坐标 (开始运行后经过的时间，毫秒)，y 坐标 (进程编号)。
- 每个 **resol** 统计的数据：x 坐标 (开始运行后经过的时间，毫秒)，y 坐标 (进程进度)。

两个文件大致内容为：

接下来，使用 `C/C++` 写一段利用 `popen()` 创建子进程，并通过管道传输命令来控制 `gnuplot` 的代码：

> txt2png.cpp

```
#include <bits/stdc++.h>

int main(int args, char* argv[]) {

        // txt2png tmp1.txt res1.png
        if (args != 3) {
                std::cerr << "Usage expected: txt2png xxx[string]
xxx[string]" << std::endl;
                exit(EXIT_FAILURE);
        }

        // windows 平台下已经将所有的 popen/pclose 换成 _popen/_pclose
        // 以管道的形式 使用 popen() 打开一个 gnuplot 的进程
        FILE *pipe = popen("gnuplot -persist", "w");// 这里不用保持窗口也可以(
-persist 可以去掉 )
        if (!pipe) {
                std::cerr << "Error executing gnuplot" << std::endl;
                exit(EXIT_FAILURE);
        }

        // 通过管道传输命令，注意每条命令换行

        fprintf(pipe, "set terminal png\n");
```

```
        fprintf(pipe, "set output '%s'\n", argv[2]);
        fprintf(pipe, "plot '%s' with points\n", argv[1]);

        pclose(pipe);
        exit(EXIT_SUCCESS);
}
```

所以接下来的流程是先运行 `sched.cpp` 得到的可执行程序 `sched`，然后得到两个数据文件 `tmp1.txt`，`tmp2.txt`，之后使用 `txt2png.cpp` 编译得到的可执行程序 `txt2png`，将两个数据文件转化为散点图。

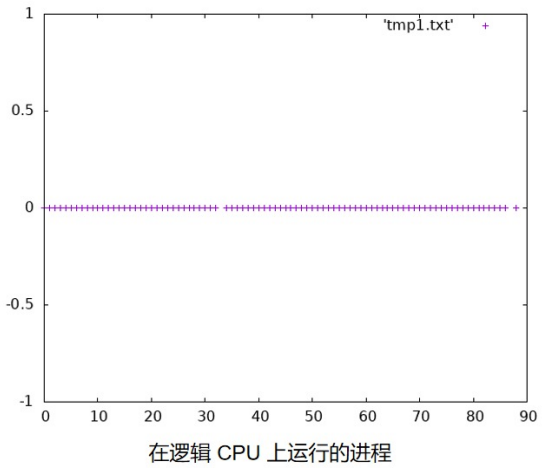
同时，为了让我们的命令只在我们规定的 逻辑 CPU 上运行，需要 `taskset` 命令。

> taskset -c

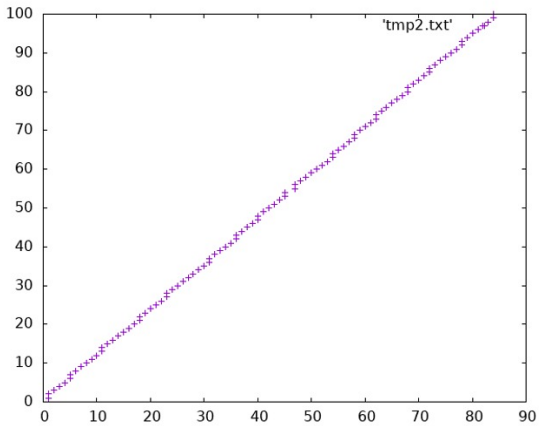
1.1 单核(1个进程)

```
taskset -c 0 ./sched 1 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
[可选] feh res1.png
[可选] feh res2.png
```

结果为：



在逻辑 CPU 上运行的进程



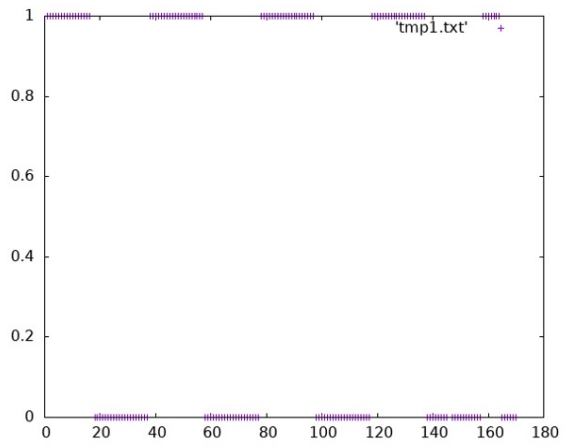
进程 0 的进度

> 该实验只运行 **1** 个进程，因此逻辑 CPU 0 上一直运行 **1** 个进程，进程进度呈线性上升。

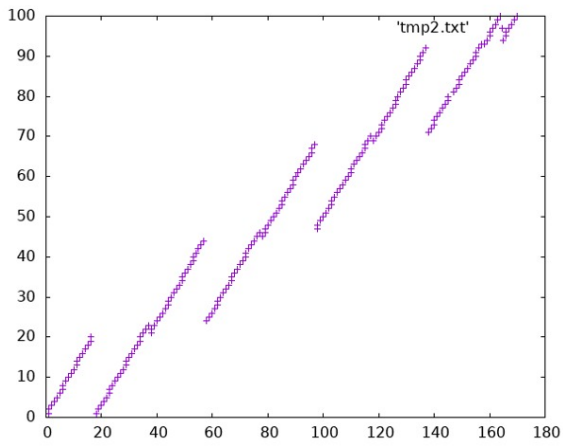
1.2 单核(2个进程)

```
taskset -c 0 ./sched 2 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
[可选] feh res1.png
[可选] feh res2.png
```

结果为:



在逻辑 CPU 上运行的进程编号



进程 0, 1 的进度

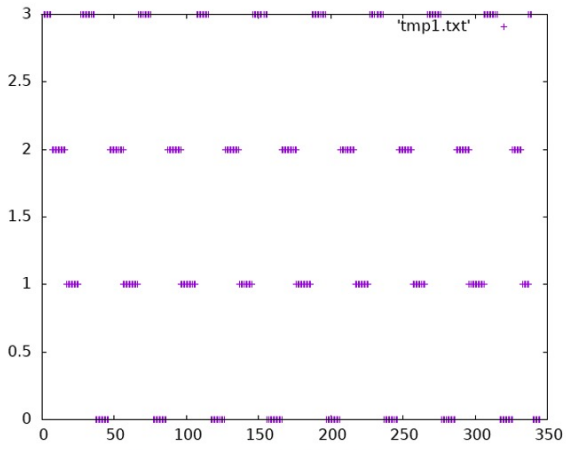
此处由于没有对不同的进程指定不同的颜色 (要实现这也可以做到，只是可能要改动很多，不过这偏离了我们的初衷)。可以看到:

- 2 个进程轮流使用 `cpu`
- 2 个进程获得时间片几乎相等。

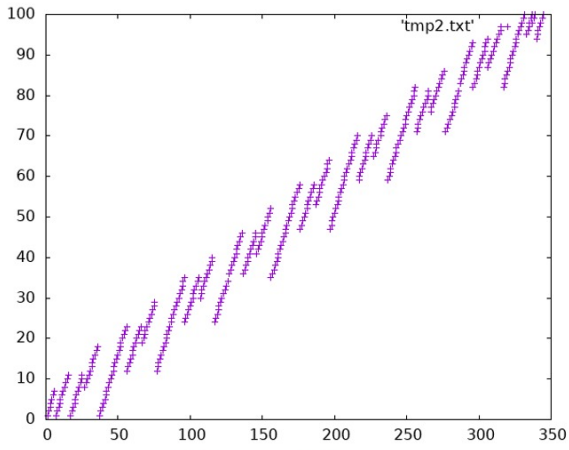
1.3 单核(4 个进程)

```
taskset -c 0 ./sched 4 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
[可选] feh res1.png
[可选] feh res2.png
```

结果为:



在逻辑 CPU 上运行的进程编号



进程 0,1,2,3 的进度

综上：

- 不管运行多少个进程，在任意时间点上，只能有一个进程运行在逻辑 CPU 上
- 在逻辑 CPU 上运行多个进程，会采取 **轮询调度** 的方式循环进行，即 **所有进程按顺序逐个运行，一轮过后重新从第一个进程开始轮询**
- 每个进程被分配到的时间片大致相等
- 全部进程运行结束所消耗的时间吗，随着进程数量的增加而等比例增加。、

2. 上下文切换

由于上面发现，Linux 系统在运行不同进程采用 **轮询调度** 的方式，因此对于某进程中的两个相邻操作，它们不一定执行时间相邻：

```
int main() {
    foo();
    bar();
}
```

当 `foo()` 执行完，刚好消耗了该进程的 1 个时间片时，就会发生 **上下文切换** 转而执行另外一个进程的代码，等到下一个时间片来临，才会执行 `bar()` 的代码。

3. 进程的状态

使用 `ps ax` 可以列举当前系统的所有进程，另外统计其行数，就能得到正在运行的进程总数 `ps ax | wc -l`，我的 `wsl` 在不运行其它程序下，返回的结果为：

```
$ps ax | wc -l
36
```

进程存在各种状态，进程的一部分状态如下所示：

状态名	含义
运行态	正在逻辑 CPU 上运行
就绪态	进程具备运行条件，等待分配 CPU 时间
睡眠态	进程不准备运行，除非发生某事件。在此期间不消耗 CPU 时间
僵死状态	进程运行结束，正在等待父进程将其收回

一般来说，处于睡眠态的进程所等待的事件有以下几种：

- 等待指定的时间
- 等待用户通过键盘或鼠标等设备进行输入
- 等待 HDD 或者 SDD 等外部存储器的读写结束
- 等待网路的数据收发结束

查看 `ps ax` 输出结果的第 3 个字段 STAT 的首字母，就可以得知进程处于那种状态：

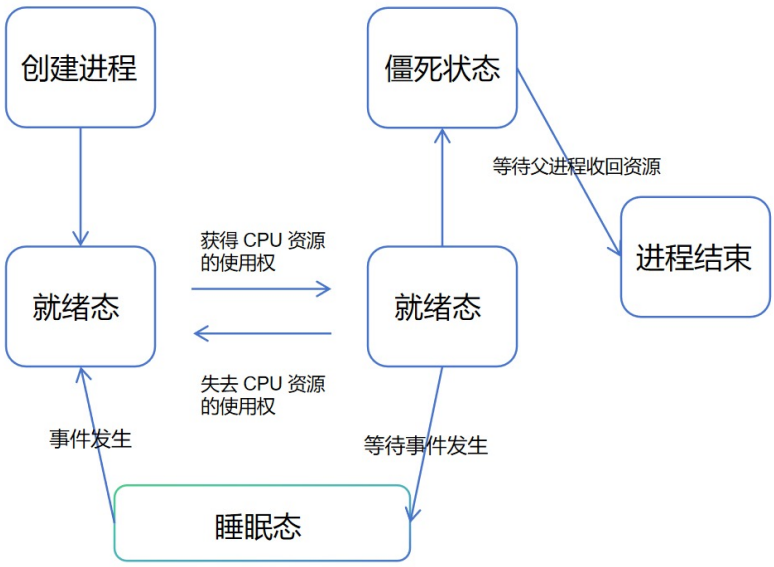
STAT 字段的首字母	状态
R	运行态或者就绪态
S 或 D	睡眠态。S 指可通过接收信号回到运行态，D 指 S 意外的情况 (D 主要出现于等待外部存储器的访问时)
Z	僵死状态

例如，观察：

```
  339 ?        Sl     0:04 /snap/ubuntu-desktop-installer/1284/usr/bin/python3.10 -m subiquity.cmd.server --
  343 ?        Ss     0:00 /init
  344 ?        R      0:00 /init
  345 pts/0    Ss     0:00 -bash
  346 pts/1    Ss     0:00 /bin/login -f
  404 ?        Ss     0:00 /lib/systemd/systemd --user
  405 ?        S      0:00 (sd-pam)
  410 pts/1    S+     0:00 -bash
  429 ?        S      0:06 python3 /snap/ubuntu-desktop-installer/1284/usr/bin/cloud-init status --wait
 1088 pts/0    R+     0:00 ps ax
```

- `ps ax` 进程为运行态(R)，因为它需要一直输出进程状态
- `bash` 为睡眠态，等待用户输入变成运行态
- 处于 `D` 状态的进程通常在几毫秒之内就会迁移到别的状态，长时间处于 D 状态的进程时，需要考虑：
  - 存储器的 I/O 处理尚未结束。
  - 内核中发生了某种问题。

4. 状态转换



5. 空闲状态

当 CPU 不执行任何处理时，会运行一个称为 **空闲进程** 的特殊进程。一种简单的设计该种进程方法是，创建一个无意义的循环的进程，直到需要唤起新进程。显然，实际上不会这样处理，事实上计算机使用了特殊的 CPU 指令使得逻辑 CPU 进入休眠状态，这也是智能手机，计算机在待机下能长时间续航的原因。

> 可以通过 `sar -P ALL 1 1` 的 `%idle` 字段查看 CPU 的空闲状态时间占比

6. 吞吐量和延迟

吞吐量和延迟两个指标不仅适用于逻辑 CPU ，还可以用于评价其他硬件 (例如外部存储器)；这里仅叙述逻辑 CPU 下的情况：

- 吞吐量 = 处理完成的进程数量 / 耗费的总时间
- 延迟 = 结束处理的时间 - 开始处理的时间

对于吞吐量，一般空闲状态的时间占比低，吞吐量就越大。例如：

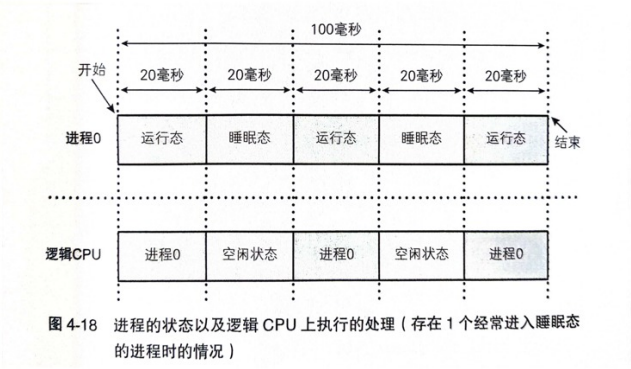


图 4-18 进程的状态以及逻辑 CPU 上执行的处理（存在 1 个经常进入睡眠态的进程时的情况）

吞吐量 = 1进程 / 100ms = 10 进程/s

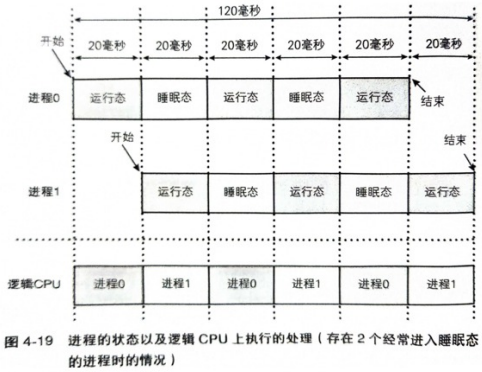


图 4-19 进程的状态以及逻辑 CPU 上执行的处理（存在 2 个经常进入睡眠态的进程时的情况）

吞吐量 = 2进程 / 120ms = 16.7 进程/s

左图空闲时间 (%idle) = 40ms / 100ms = 40%；右图空闲时间 (%idle) = 0ms / 120ms = 0;

回顾之前第 `1` 节的 **3** 个实验，将其吞吐量和延迟总结在下表：

进程数量	吞吐量 (进程数量 / s)	平均延迟 (ms)
1	1000 / 85	85
2	2000 / 170	170
4	4000 / 340	340

得出结论：

- 在耗尽逻辑 CPU 的计算能力后，也就是说，所有的逻辑 CPU 都不处于空闲状态，不管增加多少个进程，吞吐量都不会发生变化（事实上下文切换的开销会增大因此导致吞吐量会下降）

- 随着进程数量的增加，延迟会越来越长
- 每个进程的平均延迟是相等的

7. 现实中的系统

之前的实验都是设计成逻辑 CPU 始终处于工作状态，并且没有程序处于就绪态时，此时吞吐量与延迟就是最优值。然而，现实中一般会出现：

- 空闲状态，由于逻辑 CPU 处于空闲状态，吞吐量降低
- 进程正在运行，且没有程序处于就绪态的进程，这是理想的状态，但是这样的状态下加入一个处于就绪态的进程，两个进程的延迟都会变长
- 进程正在运行，且存在就绪态的进程，这时吞吐量很大，但是延迟也会变长。

在设计系统时，为了使得系统达到目标性能，也会基于一些数据对系统进行优化。

- `sar` 命令中的 `%idle` 字段
- `sar -q` 的 `runq-sz` 字段。该字段显示的是处于运行态或就绪态的进程总数 (全部逻辑 CPU 的合计值)，不会显示系统进程的运行态(例如你在 `ps ax` 里看到的 `/init`)

例如：

```
syz@syz:~/projects/class4$ sar -q 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  02/25/24        _x86_64_        (20 CPU)

15:32:47      runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15   blocked
15:32:48            0       305      0.06      0.02      0.00         0
Average:            0       305      0.06      0.02      0.00         0
syz@syz:~/projects/class4$ ./loop &
[1] 626
syz@syz:~/projects/class4$ sar -q 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  02/25/24        _x86_64_        (20 CPU)

15:33:07      runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15   blocked
15:33:08            1       306      0.12      0.03      0.01         0
Average:            1       306      0.12      0.03      0.01         0
syz@syz:~/projects/class4$
```

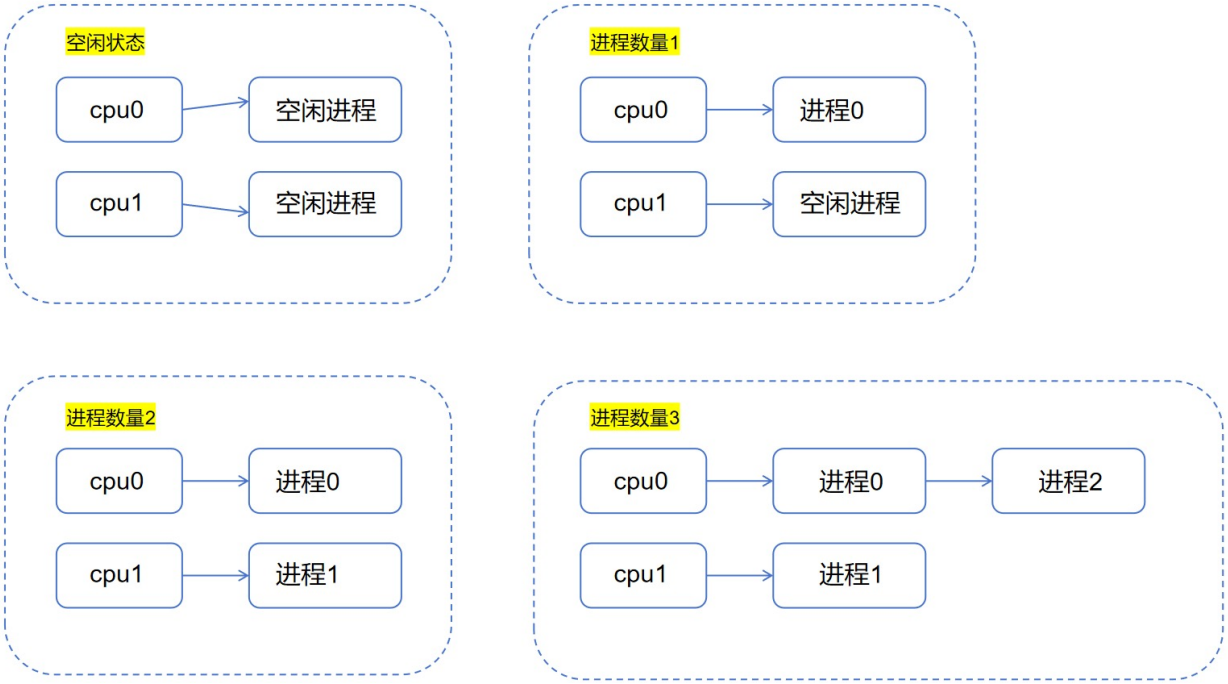
注意看，运行这个死循环程序，`runq-sz` 字段增加 **1**

> 注意事后关掉这个进程

8. 多个 CPU 核心

当存在多个逻辑 CPU 时，调度器会采取被称为负载均衡或全局调度的策略，即，负载均衡负责公平地把进程分配个多个逻辑 CPU，与只有单个逻辑 CPU 相同，各个逻辑 CPU 内，调度器为在逻辑 CPU 上运行地各个进程分配均等的 CPU 时间。

假设只有两块 CPU 核心：



> 如果有 4 个进程来临，那么第 4 个进程会安排在第 2 块 cpu 上

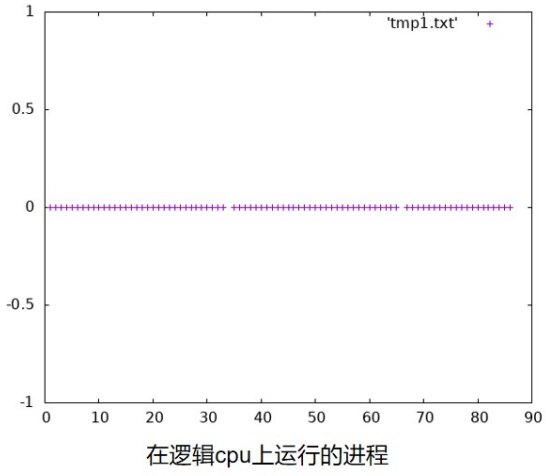
你可以通过 `grep -c processor /proc/cpuinfo` 查看你的电脑的 cpu 核心数，也可以通过 `sar -P ALL 1 1` 得知 cpu 核心数目。

下面我们从电脑的上众多 `cpu` 核心取出两个进行上述实验。根据书上所说，使用 `cpu0` 与 `cpu核心数 / 2` 的逻辑 cpu 效果最好，这二者较为独立。

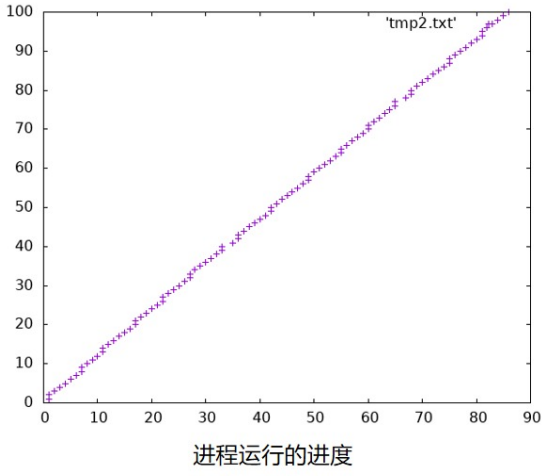
8.1 双核(1个进程)

```
taskset -c 0,10 ./sched 1 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
```

结果为：



在逻辑cpu上运行的进程



进程运行的进度

> 这与实验 1.1 结果一致

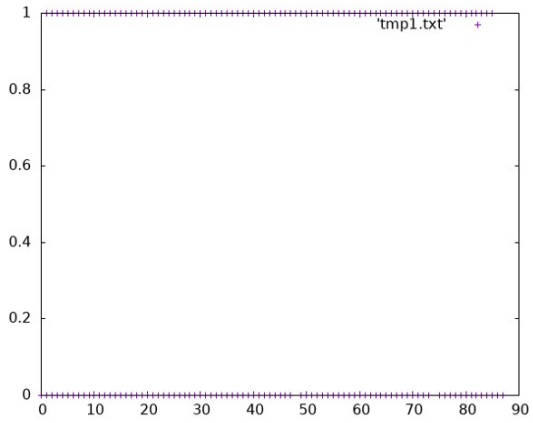
8.2 双核(2 个进程)

```
taskset -c 0,10 ./sched 2 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
```

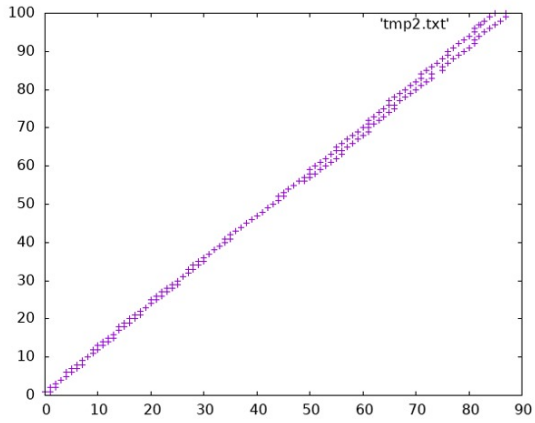
结果为：

> 可以看到这两个进程确实分在两块 cpu 核心上并行，使得延迟与只运行单个进程一致

8.3 双核(4个进程)



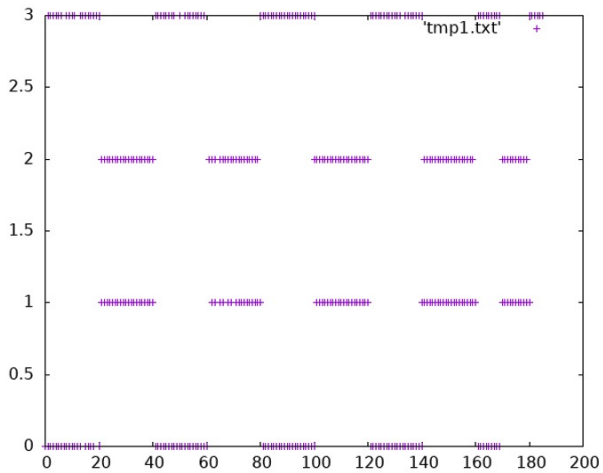
在逻辑cpu上运行的进程



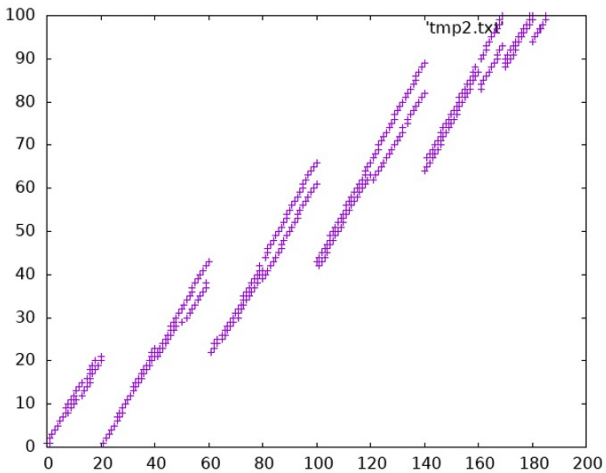
进程运行的进度

```
taskset -c 0,10 ./sched 4 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
```

结果为：



在逻辑cpu上运行的进程



进程运行的进度

> 可以看到左图其实相当于将 1.2 的结果复制两遍，右图的延迟与 1.2 实验一致。

(双核)将其吞吐量与延迟总结如下：

进程数量	吞吐量 (进程数量 / s)	平均延迟 (ms)
1	1000 / 85	85
2	2000 / 85	85

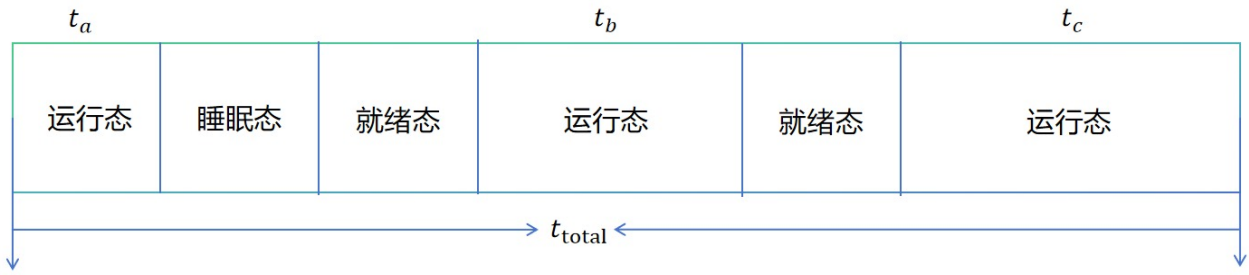
进程数量	吞吐量 (进程数量 / s)	平均延迟 (ms)
4	4000 / 180	180

得到结论：

- “ n 个核心等于 n 倍性能 ”，至少应该要同时运行的进程数量达到核心数目
- 当进程数量超过核心数目，吞吐量不再提高。

9. 运行时间和执行时间

使用 `time` 命令运行进程，可以得到进程从开始到结束所有经过的时间，以及实际处执行处理消耗的时间。观察下图，



- 执行时间 = $t_a + t_b + t_c$
- 运行时间 = t_{total}

9.1 单核(1 个进程)

执行 `time taskset -c 0 ./sched 1 10000 10000`

得到结果;

```
syz@syz:~/projects/class4$ time taskset -c 0 ./sched 1 10000 10000
estimating workload which takes just one milisecond
end estimation

real    0m8.790s
user    0m8.789s
sys     0m0.001s
```

其中：

- `real` 是运行时间，`user + sys` 是执行时间

由于我们设计的程序大部分时间处于运行态，因此运行时间 ≈ 执行时间，同时存在轻微的系统调用消耗时间，可能是由于 `fork()` 以及 `wait(NULL)`

9.2 单核(2个进程)

执行 `time taskset -c 0 ./sched 2 10000 10000`

得到结果:

```
syz@syz:~/projects/class4$ time taskset -c 0 ./sched 2 10000 10000
estimating workload which takes just one milisecond
end estimation

real    0m17.713s
user    0m17.712s
sys     0m0.001s
```

> 两个进程争用一个 cpu 核心，因此执行时间和运行时间都变为 2 倍

9.3 双核(1个进程)

执行 `time taskset -c 0,10 ./sched 1 10000 10000`

得到结果:

```
syz@syz:~/projects/class4$ time taskset -c 0,10 ./sched 1 10000 10000
estimating workload which takes just one milisecond
end estimation

real    0m8.812s
user    0m8.792s
sys     0m0.020s
```

此时计算 `real` 以及 执行时间 的逻辑是:

9.4 双核(2个进程)

执行 `time taskset -c 0,10 ./sched 4 10000 10000`

得到结果

```
syz@syz:~/projects/class4$ time taskset -c 0,10 ./sched 4 10000 10000
estimating workload which takes just one milisecond
end estimation

real    0m17.476s
user    0m34.564s
sys     0m0.011s
```

> 此时只用 17.476s 的运行时间，在两块 cpu 上总共执行了 34.564s

10. 进程睡眠

如果我们执行一个睡眠命令，并统计其运行时间和执行时间：

```
syz@syz:~/projects/class4$ time sleep 10

real    0m10.001s
user    0m0.001s
sys     0m0.001s
syz@syz:~/projects/class4$
```

> 此时，由于睡眠指令不消耗 cpu 运行时间，没有几乎处于执行态的进程，所以执行时间几乎为 0，但是程序从开始到结束需要经过10s，所以运行时间为 10s

11. 其它得到运行时间和执行时间的方法

使用 `ps -eo pid,comm,etime,time` 可以分别获取各个进程的进程 ID，命令名，运行时间，和执行时间。

结果为：

```
syz@syz:~/projects/class4$ ps -eo pid,comm,etime,time
    PID COMMAND             ELAPSED     TIME
      1 systemd            02:23:41 00:00:00
      2 init-systemd(Ub    02:23:41 00:00:00
      5 init               02:23:40 00:00:00
     38 systemd-journal    02:23:40 00:00:00
     58 systemd-udevd      02:23:40 00:00:00
     70 snapfuse           02:23:40 00:00:00
     73 snapfuse           02:23:40 00:00:00
     75 snapfuse           02:23:40 00:00:00
     78 snapfuse           02:23:40 00:00:00
     84 snapfuse           02:23:40 00:00:00
     86 snapfuse           02:23:40 00:00:01
     87 snapfuse           02:23:40 00:00:00
     94 systemd-resolve    02:23:40 00:00:00
    119 cron               02:23:40 00:00:00
    121 dbus-daemon        02:23:40 00:00:00
```

> 此时我已经学习了 2h23min ，运行时间基本上都这么大，但是大部分系统程序处于睡眠态，未被执行。

12. 变更优先级

此前，我们介绍的都是系统均等分配 CPU 时间给所有可以运行的进程。但是，也可以使用 `nice()` 为特定的进程指定优先级。

```
int nice(int inc);
```

- `inc` 可以取得 [-19, 20] 之间的整数，数字越大优先级越小，优先级越大，可以获得更多的 CPU 时间

下面，考虑修改 `sched.cpp` 为 `sched_nice.cpp` 实现：

- 将运行的进程数量固定为 `2` 个。
- 第 `1` 个参数为 `total`，第 `2` 个参数为 `resol`
- 将 `2` 个进程的优先级分别设为默认的 `0` 和 `5`
- 剩余部分与原本的 `sched` 程序保持一致

> sched_nice.cpp

```
#include <sys/types.h>
#include <sys/wait.h>
#include <bits/stdc++.h>
#include <unistd.h>
#include <err.h>
#include <time.h>
#include <memory>

constexpr uint64_t NLOOP_FOR_ESTIMATION = 1000000000ULL;
constexpr uint64_t NSECS_PER_MSEC = 1000000ULL;
constexpr uint64_t NSECS_PER_SEC = 1000000000ULL;

// timespec {tv_sec, tv_nsec} 秒数，纳秒
constexpr uint64_t calc(timespec t) {
    return t.tv_sec * NSECS_PER_SEC + t.tv_nsec;
}
```

```
static inline int64_t diff_nsec(timespec before, timespec after) {
    return calc(after) - calc(before);
}

static uint64_t loops_per_msec() {
    timespec before, after;
    clock_gettime(CLOCK_MONOTONIC, &before);

    // 执行空循环测试循环指定次数，需要的 CPU 运行时间
    for (uint64_t i = 0;i < NLOOP_FOR_ESTIMATION;i ++);

    clock_gettime(CLOCK_MONOTONIC, &after);

    // 返回每 ms 运行的循环的次数
    return NLOOP_FOR_ESTIMATION * NSECS_PER_MSEC / diff_nsec(before, after);
}

static inline void load(uint64_t nloop) {
    for (uint64_t i = 0;i < nloop;i ++ );
}

FILE *out1, *out2;
static void child_fn(int id, std::shared_ptr<timespec[]> buf, int nrecord,
uint64_t nloop_per_resol, \
        timespec start, FILE* out1, FILE* pic2) {

    for (int i = 0;i < nrecord;i ++ ) {
        timespec ts;
        load(nloop_per_resol);// 模拟每个 resol 收集一次数据
        clock_gettime(CLOCK_MONOTONIC, &ts);
        buf[i] = ts;
    }
    // 进程id & 每个resol距开始的 ms 数 & 运行进度
    for (int i = 0;i < nrecord;i ++ ) {
        // printf("%d\t%ld\t%d\n", id, diff_nsec(start, buf[i]) /
NSECS_PER_MSEC, \
        //     (i + 1) * 100 / nrecord);
        int finish_rate = (i + 1) * 100 / nrecord;
        uint64_t cost_time = diff_nsec(start, buf[i]) / NSECS_PER_MSEC;
        fprintf(out2, "%llu %d\n", cost_time, finish_rate);
```

```
        fprintf(out1, "%llu %d\n", cost_time, id);
    }
    exit(EXIT_SUCCESS);
}

std::shared_ptr<pid_t[]> pids;


// args = 命令行参数 + 1
int main(int args, char *argv[]) {
    out1 = fopen("tmp1.txt", "w"); // 打开名为tmp1.txt的文件，以写入模式打开
    out2 = fopen("tmp2.txt", "w"); // 打开名为tmp2.txt的文件，以写入模式打开
    if (args !=  3) {
        fprintf(stderr, "usage: %s <total[ms]> <resolution[ms]>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    int nproc = 2;
    int total = atoi(argv[1]);
    int resol = atoi(argv[2]);

    // std::cout << "pars: " << nproc << " " << total << " " << resol <<
std::endl;
     // 下面是对于命令行参数错误使用的输出
    std::vector<std::string> names = {"total", "resol"};
    for (int i = 0;i < 2;i ++ ) {
        int x = atoi(argv[i + 1]);
        if (x < 1) {
            fprintf(stderr, "<%s>(%d) should be >= 1\n", names[i].c_str(),
x);
            exit(EXIT_FAILURE);
        }
    }

    if (total % resol) {
        fprintf(stderr, "<total>(%d) should multiple of <resolution>(%d)\n",
total, resol);
        exit(EXIT_FAILURE);
    }
```

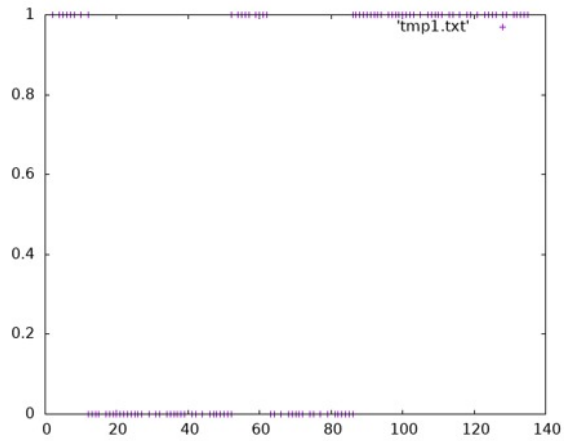
```
    int nrecord = total / resol; // 收集信息的次数
     // 申请动态的 timespec 结构体指针
    std::shared_ptr<timespec[]> logbuf(new timespec[nrecord]);

    if (!logbuf) {
        err(EXIT_FAILURE, "new(logbuf) failed");
    }

    puts("estimating workload which takes just one milisecond");
    uint64_t nloop_per_resol = loops_per_msec() * resol;
    puts("end estimation");
    fflush(stdout);

    pids = std::shared_ptr<pid_t[]>(new pid_t[nproc]);
    if (!pids) {
        warn("new(pids) failed");
    }

    timespec start;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0;i < nproc;i ++ ) {
        pids[i] = fork();
        if (pids[i] < 0) {
            // fork() 失败，代表实验失败，父进程杀掉所有子进程
            for (int j = 0;j < i;j ++ )
                if (kill(pids[i], SIGINT) < 0)
                    warn("kill (%d) failed", pids[i]);
            // 等待子进程全部退出，确保父进程回收所有子进程的资源
            for (int j = 0;j < i;j ++ )
                if (wait(NULL) < 0)
                    warn("wait() failed.");
            exit(EXIT_FAILURE);
        } else if (pids[i] == 0) {
             // 子进程
             if (i == 1) nice(5);
             child_fn(i, logbuf, nrecord, nloop_per_resol, start, out1,
out2);
```

```
            // 此前该个子进程已经结束，要么在 wait(NULL) 后被收回资源，要么等待父进程收回
        }

    }

     // 运行到这，说明成功创建了 nproc 个并行子进程
    for (int i = 0;i < nproc;i ++ )
        if (wait(NULL) < 0)
            warn("wait() failed.");
    ;
    exit(EXIT_SUCCESS);
}
```

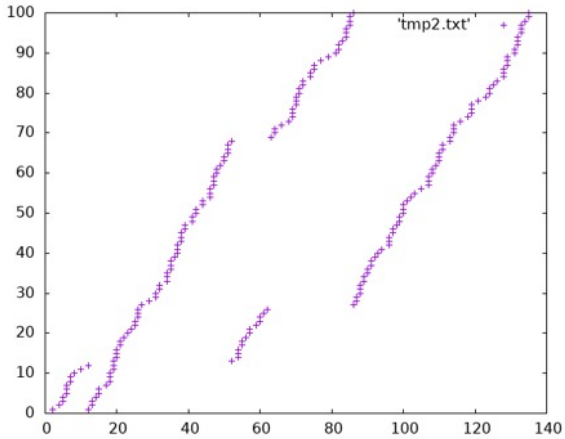
接着运行：

```
taskset -c 0 ./sched_nice 100 1
./txt2png tmp1.txt res1.png
./txt2png tmp2.txt res2.png
```

得到结果：



逻辑cpu0上运行的进程



进程 0 与进程 1 的进度

- 可以看到，进程0的优先级更大，确实被分配了更多的 CPU 时间，使得其能更快地完成。

同时 `bash` 里也可以使用 `nice -n` 来指定程序运行的的优先级：

```
nice -n 5 echo hello
```

`sar` 指令的 `%nice` 字段，表示的是程序在默认值更改为其它优先级后，进程运行时间所占的比例：

输入：

```
$ ./loop &
[1] 799

$ sar -P ALL 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  02/26/24        _x86_64_
   (20 CPU)

13:02:36        CPU     %user     %nice   %system   %iowait    %steal
%idle
13:02:37          8    100.00      0.00      0.00      0.00      0.00
 0.00

$ kill 799

$ nice -n 5 ./loop &
[2] 802
[1]   Terminated              ./loop

$ sar -P ALL 1 1
Linux 5.15.133.1-microsoft-standard-WSL2 (syz)  02/26/24        _x86_64_
   (20 CPU)

13:03:11        CPU     %user     %nice   %system   %iowait    %steal
%idle
13:03:12          8      0.00    100.00      0.00      0.00      0.00
 0.00
```

- 观察上述输出，前者(默认优先级)是，`%user` 字段为 `100%`，后者（`nice -n 5`）是 `%nice` 字段为 `100%`