4.17　运行时间和执行时间

通过 time 命令运行进程，就能得到进程从开始到结束所经过的时间，以及实际执行处理所消耗的时间。

- **运行时间：进程从开始运行到运行结束为止所经过的时间。类似于利用秒表从开始运行的时间点开始计时，一直测量到运行结束**
- **执行时间：进程实际占用逻辑 CPU 的时长**

运行时间和执行时间如图 4-31 所示。

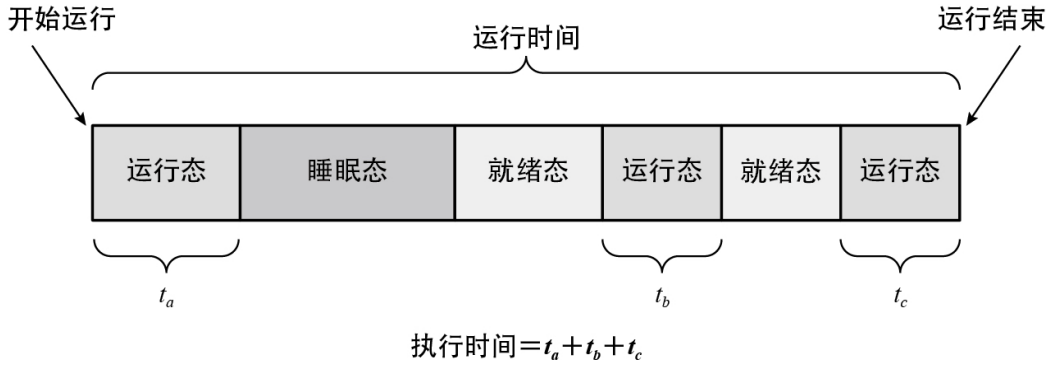

图 4-31　运行时间和执行时间

接下来，我们尝试使用 time 命令，来获取 sched 程序的运行时间和执行时间。本次实验的重点并非考察运行进度等细节，而是计算进程结束时的运行时间和执行时间。因此，在本次实验中，我们将 sched 程序的参数调整为下列数值。

- **total**（运行的总时长）：10 秒
- **resol**（输出进度的时间间隔）：10 秒

● 逻辑 CPU 数量=1，进程数量=1

```
$ time taskset -c 0 ./sched 1 10000 10000
0       9811    100

real    0m11.567s
user    0m11.560s
```

```
sys     0m0.000s
$
```

real 的值为运行时间，user 与 sys 的和为执行时间。

user 表示进程在用户模式下耗费的 CPU 时间。与此对应的 sys 则表示内核为了响应用户模式发出的请求而执行系统调用所耗费的时间。

在这个例子中，进程独占逻辑 CPU，因此运行时间与执行时间几乎相等。另外，由于大部分时间用于在用户模式下执行循环处理，所以 sys 的值几乎为 0。

运行完该进程需要耗费大约 11.6 秒，但完成 100% 进度只需要大约 9.8 秒。之所以存在这个时间差，是由于在开始处理前需要估算 1 毫秒 CPU 时间对应的计算量（sched.c 的 loops_per_msec() 函数），这个预处理消耗了一部分时间（图 4-32）。

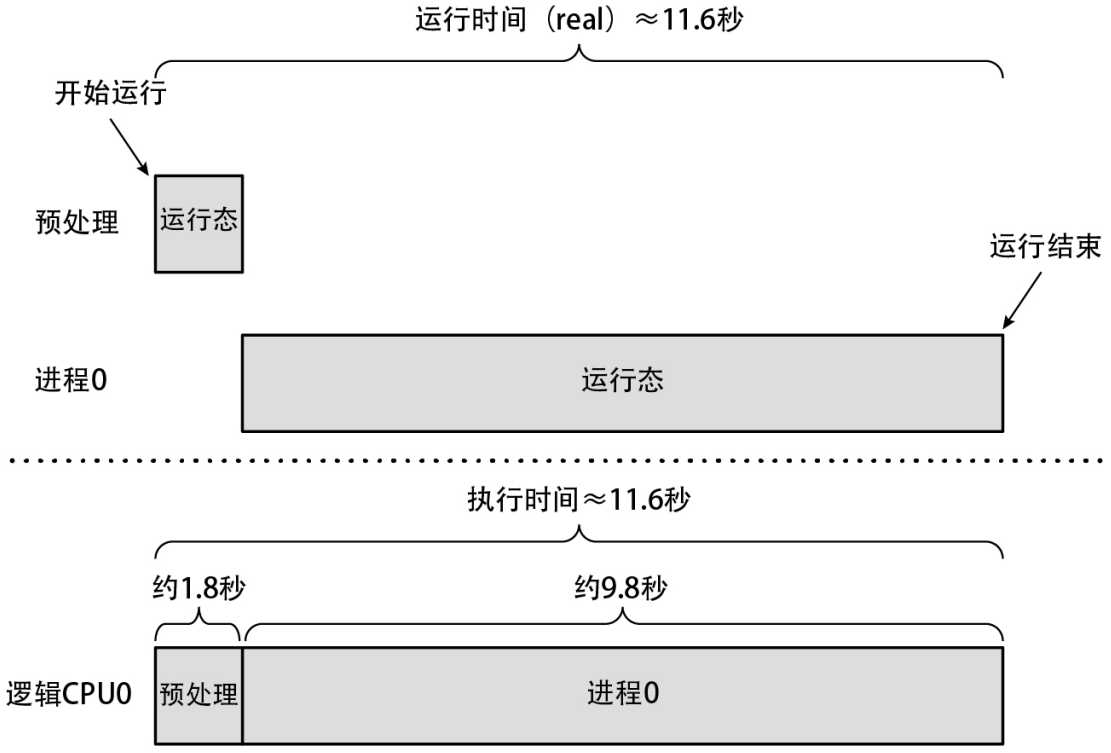

图 4-32　运行时间与执行时间的时间差

● 逻辑 CPU 数量=1，进程数量=2

```
$ time taskset -c 0 ./sched 2 10000 10000
1       19716   100
0       19732   100

real    0m21.487s
user    0m21.480s
sys     0m0.000s
$
```

可以看到，这次的运行时间与执行时间也几乎相等。除去预处理的时间后，运行时间与执行时间都几乎变为前一次实验的 2 倍。这是因为在单位时间内，每个进程只能使用其中一半的时间（图 4-33）。

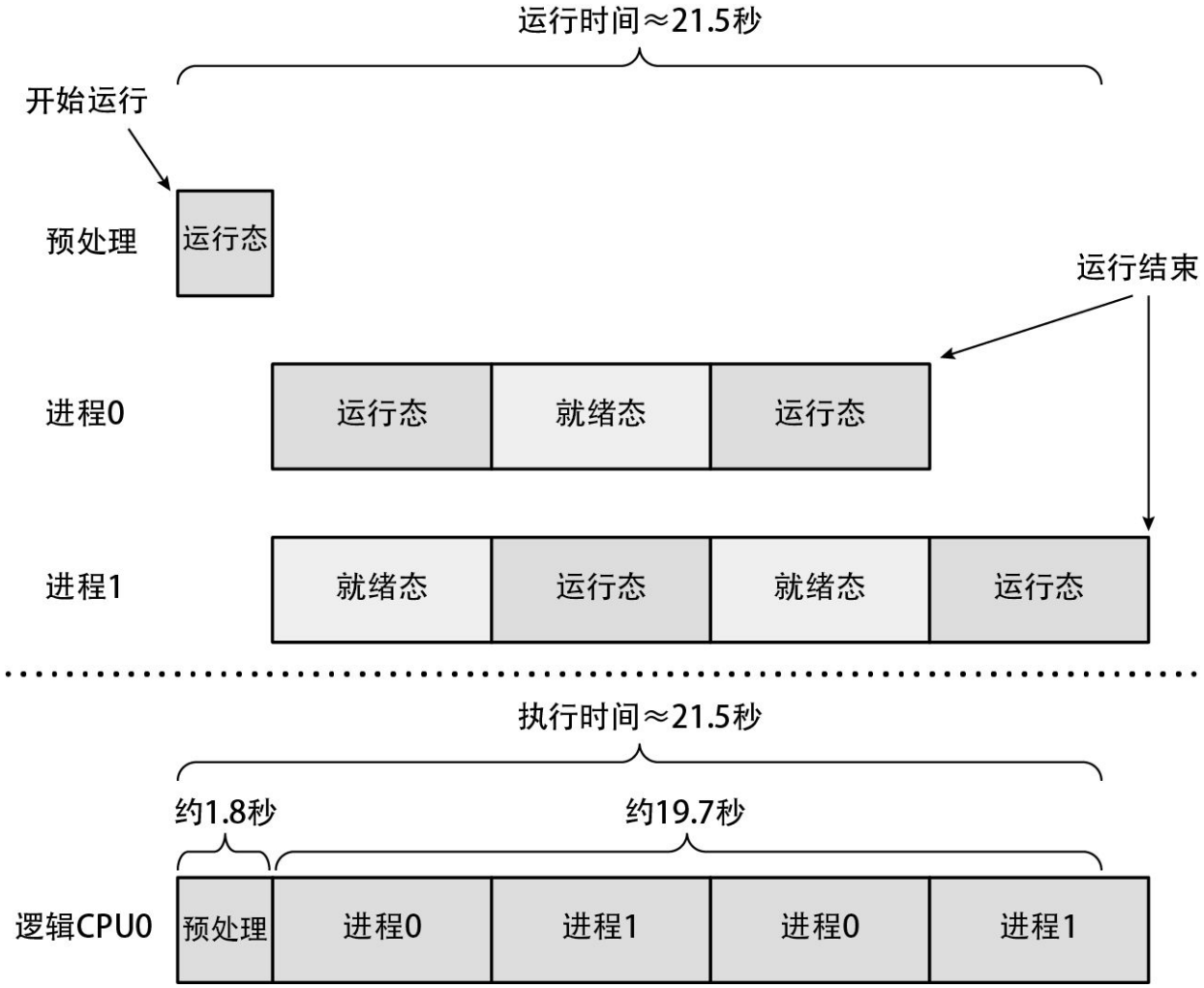

图 4-33　运行时间与执行时间的时间差（逻辑 CPU 数量 =1，进程数量 =2）

● 逻辑 CPU 数量=2，进程数量=1

当存在 2 个逻辑 CPU 时，情况会变成什么样呢？

```
$ time taskset -c 0,1 ./sched 1 10000 10000
0       9813    100

real    0m11.569s
user    0m11.564s
sys     0m0.010s
```

与单个逻辑 CPU 时的数据几乎一模一样。通过实验数据可以得知，就算存在 2 个逻辑 CPU，其中 1 个也会一直处于空闲状态（图 4-34）。

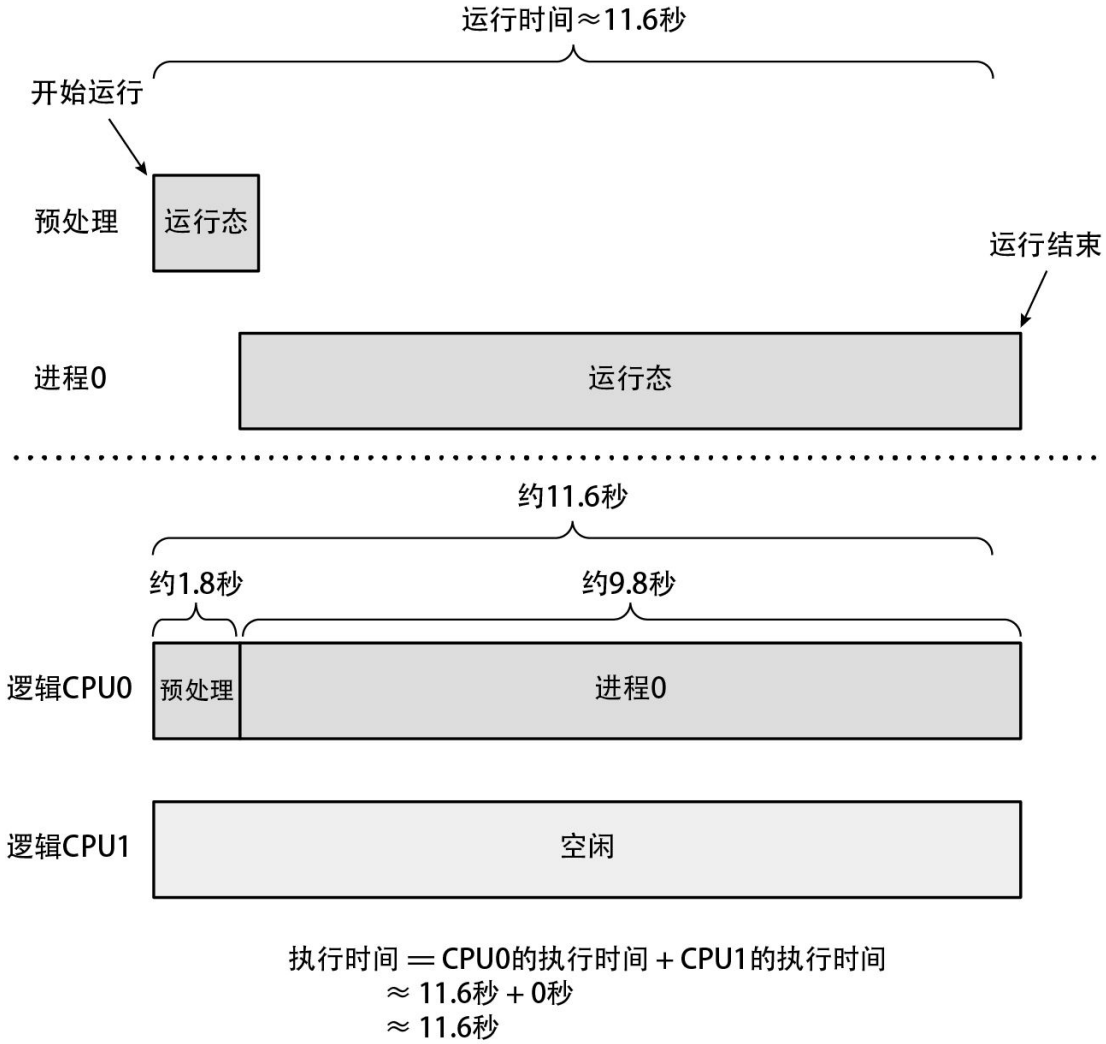

图 4-34　运行时间与执行时间的时间差（逻辑 CPU 数量 =2，进程数量 =1）

● 逻辑 CPU 数量=2，进程数量=4

```
$ time taskset -c 0,1 ./sched 4 10000 10000
0       20850   100
1       20584   100
2       19910   100
3       20871   100

real    0m22.599s
user    0m43.388s
sys     0m0.000s
```

虽然运行时间与“逻辑 CPU 数量 =1，进程数量 =2”时的实验数据相差不大，但是执行时间几乎是它的 2 倍（图 4-35）。这是因为 2 个逻辑 CPU 上的进程能够在同一时间内各自运行。

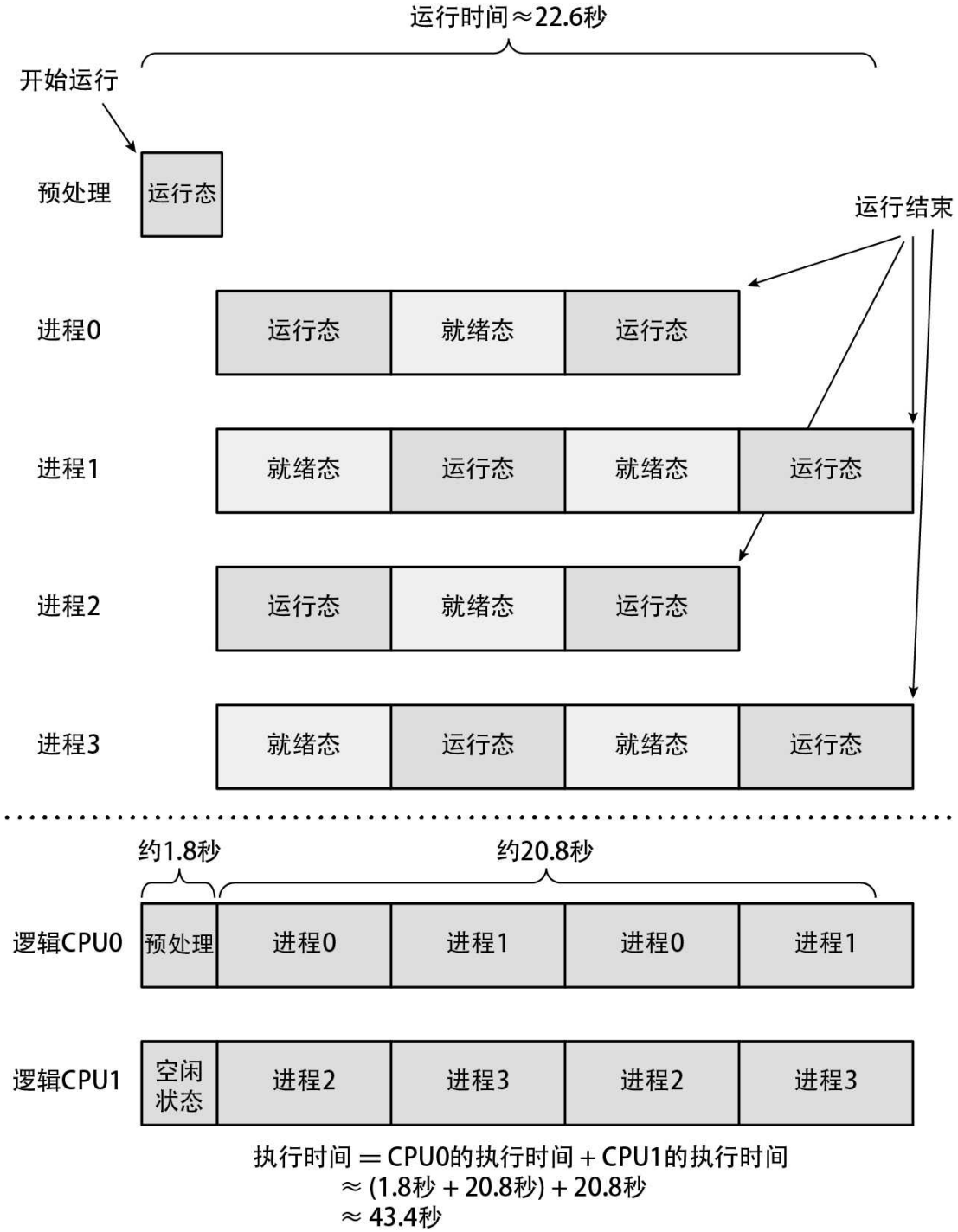

图 4-35　运行时间与执行时间的时间差（逻辑 CPU 数量 =2，进程数量 =4）

如果没有调度器的相关知识，我们将很难单凭感觉去理解为什么执行时间比运行时间长。但是，通过图 4-35 就能明白，对于多核系统来

说，得到这样的数据是理所当然的。

4.18　进程睡眠

令进程从一开始就睡眠指定的时长，然后就此结束运行，会发生什么呢？让我们尝试执行 `sleep 10`，让进程从开始运行就进入睡眠状态 10 秒，然后立刻结束运行。

```
$ time sleep 10

real    0m10.001s
user    0m0.000s
sys     0m0.000s
$
```

此时，虽然直到进程结束为止消耗了 10 秒，但是基本上没有使用逻辑 CPU，因此执行时间几乎为 0（图 4-36）。

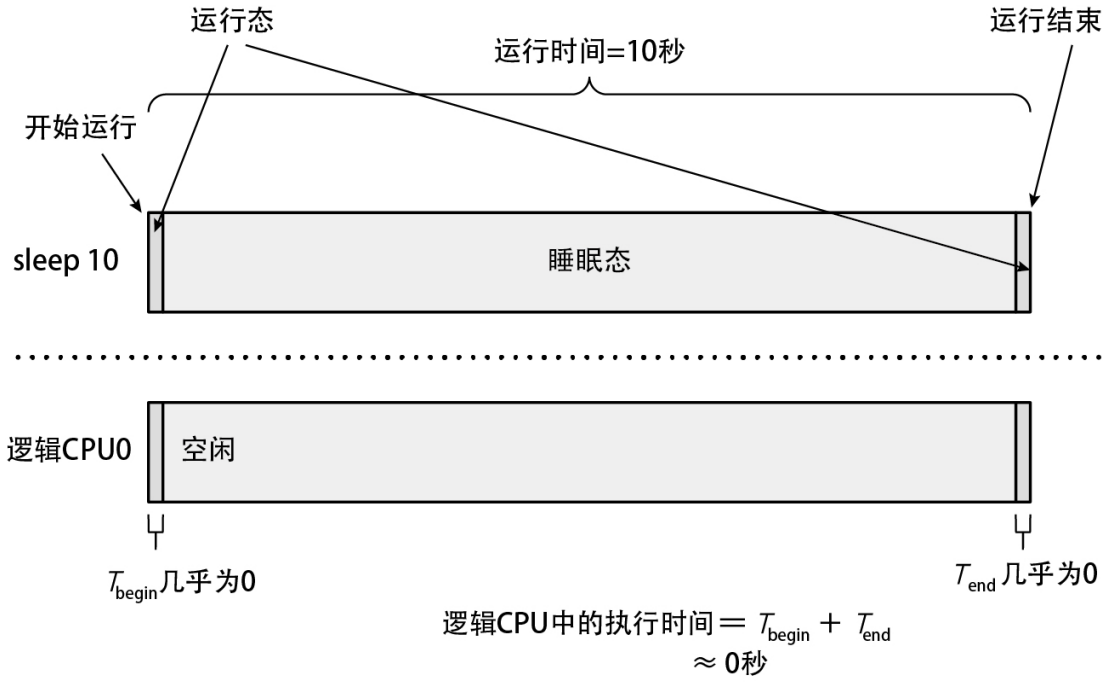

图 4-36　运行时间与执行时间（运行一个睡眠后就结束的进程时的情形）

4.19 现实中的进程

在现实中的系统上，进程会发生包含睡眠态在内的各种复杂的状态转换，因此不会出现像前面的例子那样完美的结果，但希望大家至少能读懂这些数值。下面将介绍一下 time 命令以外的获取运行时间和执行时间的方法。

ps -eo 命令的 etime 字段和 time 字段分别表示进程从开始到执行命令为止的运行时间和执行时间。下面，我们尝试通过这个命令来输出各个进程的进程 ID、命令名、运行时间和执行时间。

```
$ ps -eo pid,comm,etime,time
  PID COMMAND                 ELAPSED            TIME
( 略 )
 3739 chromium-browse    3-00:35:00         00:10:04
                         ←进程已经运行了3天零35分钟，在这期间共使用逻辑CPU
10分4秒

( 略 )
 8562 emacs              2-23:18:06         00:00:33
                         ←进程已经运行了约2天23小时18分钟，在这期间，共使用
逻辑CPU 33秒

( 略 )
$
```

在上面的运行结果中，进程 ID 为 3739 的 chromium-browse 是一个网页浏览器，进程 ID 为 8562 的 emacs 是一个文本编辑器。这些进程虽然已经运行了几天，但几乎没怎么使用逻辑 CPU。这是因为，网页浏览器与文本编辑器都属于与用户进行交互的进程，这些进程大部分时间处于睡眠态，等待用户的输入。

对于持续消耗 CPU 时间的处理（比较典型的例子是科学计算），只要不存在其他正在运行的进程，则执行时间就会接近运行时间与并行数量之积。接下来，我们利用无限循环的 loop.py 程序来进行一下实验。首先是当“逻辑 CPU 数量 =1，进程数量 =1”时的情况。

```
$ taskset -c 0 python3 ./loop.py &
[1] 20999
```

```
$ ps -eo pid,comm,etime,time | grep python3
                          ←在程序至少运行10秒后再执行这条命令
  PID COMMAND             ELAPSED      TIME
( 略 )
20999 python3                 00:11 00:00:10
                          ←运行时间与执行时间几乎相等
$
```

在实验结束后，记得结束正在运行的进程。

```
$ kill 20999
$
```

然后，看一下当“逻辑 CPU 数量 =1，进程数量 =2”时的情况。

```
$ taskset -c 0 python3 ./loop.py &
[1] 21304
$ taskset -c 0 python3 ./loop.py &   ←在上一个命令执行后立刻执行
[2] 21306
$ ps -eo pid,comm,etime,time | grep python3
21304 python3                 00:19 00:00:10
                ←执行时间约为运行时间的1/2，这是因为与进程ID为21306的进程共
享了逻辑CPU0

21306 python3                 00:19 00:00:09
                             ←执行时间约为运行时间的1/2，理由同上
$ kill 21304 21306           ←实验结束后记得结束进程
$
```

再来看一下当“逻辑 CPU 数量 =2，进程数量 =2”时的情况。

```
$ taskset -c 0,4 python3 ./loop.py &
[1] 21424
$ taskset -c 0,4 python3 ./loop.py & ←与上一个命令几乎同时执行
[2] 21425
$ ps -eo pid,comm,etime,time | grep python3
21424 python3                 00:05 00:00:05
              ←运行时间与执行时间一样，这是因为它独占一个逻辑CPU（CPU0或者
CPU4）

21425 python3                 00:05 00:00:04                ←同上
$ kill 21424 21425
$
```

大家觉得如何呢？通过采集平常使用的程序的数据，可以看到它们各自的特征。这非常有趣，请大家也一定尝试一下。

4.20 变更优先级

最后，向大家介绍一下与调度器相关的系统调用和程序。

到目前为止，我们说的都是系统会均等地分配 CPU 时间给所有可以运行的进程。但是，为特定的进程指定优先级也是可以的。nice() 就是为了实现这一点而提供的系统调用。

nice() 能通过 -19 和 20 之间的数来设定进程的运行优先级（默认值为 0），其中，-19 的优先级最高，20 的优先级最低。优先级高的进程可以比普通进程获得更多的 CPU 时间。与此相反，优先级低的进程会获得更少的 CPU 时间。需要注意的是，虽然谁都可以降低进程优先级，但是只有拥有 root 权限的用户才能进行提高优先级的操作。

让我们对前面使用过的 sched 程序进行改造，编写一个实现下述要求的程序。

- **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**

编写完成的 sched_nice 程序如代码清单 4-3 所示。

代码清单 4-3 sched_nice 程序（sched_nice.c）

```
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>

#define NLOOP_FOR_ESTIMATION 1000000000UL
#define NSECS_PER_MSEC 1000000UL
#define NSECS_PER_SEC 1000000000UL
```

```
static inline long diff_usec(struct timespec before, struct
                             timespec after)
{
    return ((after.tv_sec  NSECS_PER_SEC + after.tv_nsec)
            - (before.tv_sec  NSECS_PER_SEC + before.tv_nsec));
}

static unsigned long loops_per_msec()
{
    unsigned long i;
    struct timespec before, after;

    clock_gettime(CLOCK_MONOTONIC, &before);

    for (i = 0; i < NLOOP_FOR_ESTIMATION; i++)
        ;

    clock_gettime(CLOCK_MONOTONIC, &after);

    int ret;
    return  NLOOP_FOR_ESTIMATION  NSECS_PER_MSEC / diff_usec
(before, after);
}

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

static void child_fn(int id, struct timespec buf, int nrecord,
                     unsigned long nloop_per_resol,
                     struct timespec start)
{
    int i;
    for (i = 0; i < nrecord; i++) {
        struct timespec ts;

        load(nloop_per_resol);
        clock_gettime(CLOCK_MONOTONIC, &ts);
        buf[i] = ts;
    }
    for (i = 0; i < nrecord; i++) {
        printf("%d\t%ld\t%d\n", id, diff_usec(start, buf[i]) /
               NSECS_PER_MSEC, (i + 1)  100 / nrecord);
    }
    exit(EXIT_SUCCESS);
}

static void parent_fn(int nproc)
```

```
{
    int i;
    for (i = 0; i < nproc; i++)
        wait(NULL);
}

static pid_t pids;

int main(int argc, char *argv[])
{
    int ret = EXIT_FAILURE;

    if (argc < 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]);

    if (total < 1) {
        fprintf(stderr, "<total>(%d) should be >= 1
", total);
        exit(EXIT_FAILURE);
    }

    if (resol < 1) {
        fprintf(stderr, "<resol>(%d) should be >= 1
", resol);
        exit(EXIT_FAILURE);
    }

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

    struct timespec logbuf = malloc(nrecord  sizeof(struct
                                        timespec));
    if (!logbuf)
        err(EXIT_FAILURE, "malloc(logbuf) failed");

    unsigned long nloop_per_resol = loops_per_msec()  resol;

    pids = malloc(nproc  sizeof(pid_t));
    if (pids == NULL) {
        warn("malloc(pids) failed");
```

```
        goto free_logbuf;
    }

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

    int i, ncreated;
    for (i = 0, ncreated = 0; i < nproc; i++, ncreated++) {
        pids[i] = fork();
        if (pids[i] < 0) {
            goto wait_children;
        } else if (pids[i] == 0) {
            // 子进程

            if (i == 1)
                nice(5);
            child_fn(i, logbuf, nrecord, nloop_per_resol, start);
            / 不应该运行到这里 */
        }
    }
    ret = EXIT_SUCCESS;

    // 父进程

wait_children:
    if (ret == EXIT_FAILURE) {
        for (i = 0; i < ncreated; i++)
            if (kill(pids[i], SIGINT) < 0)
                warn("kill(%d) failed", pids[i]);

    for (i = 0; i < ncreated; i++)
        if (wait(NULL) < 0)
            warn("wait() failed.");

free_pids:
    free(pids);

free_logbuf:
    free(logbuf);

    exit(ret);
}
```

编译并运行这个程序。为了凸显优先级的作用，我们令本程序仅运行在逻辑 CPU0 上，运行结果如下所示。

```
$ cc -o sched_nice sched_nice.c
$ taskset -c 0 ./sched_nice 100 1
0       1       1
```