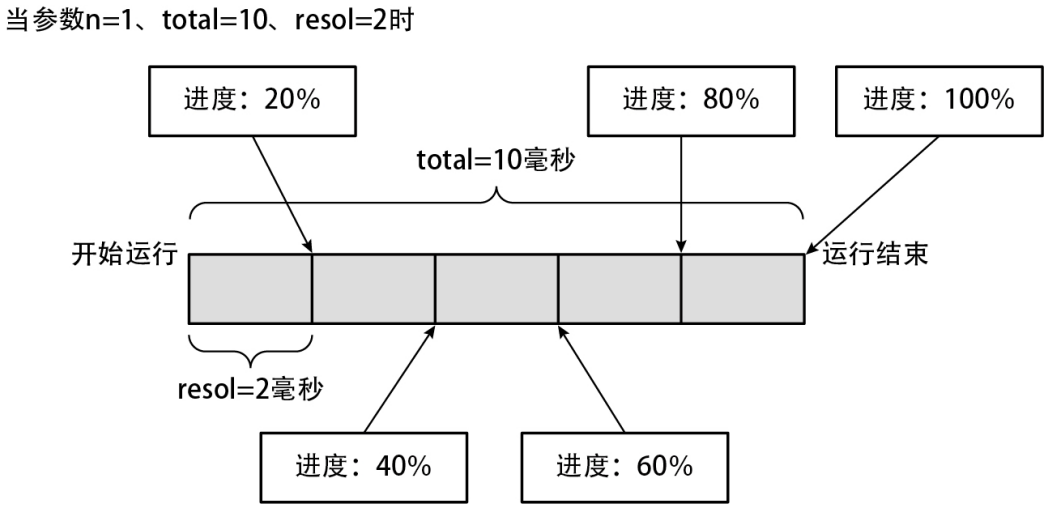
当参数n=1、total=10、resol=2时
进度：20%
进度：80%
进度：100%
total=10毫秒
开始运行
运行结束
resol=2毫秒
进度：40%
进度：60%

图 4-2　实验程序的动作

4.2　实验程序的实现

代码清单 4-1 所示为实验程序的源代码。

代码清单 4-1　sched 程序（sched.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_nsec(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()
{
    struct timespec before, after;
    clock_gettime(CLOCK_MONOTONIC, &before);

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

    clock_gettime(CLOCK_MONOTONIC, &after);

    int ret;
    return NLOOP_FOR_ESTIMATION  NSECS_PER_MSEC / diff_nsec
(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_nsec(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 < 4){
        fprintf(stderr, "usage: %s <nproc> <total[ms]>
                <resolution[ms]>
", argv[0]);
        exit(EXIT_FAILURE);
    }

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

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

```
    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)
", 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");

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

            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);
}
```

虽然代码量略大，但其内容本身并没有什么难点。这里的重点是 loops_per_msec() 函数，它用于推测消耗 1 毫秒 CPU 时间的处理所需的计算量。该函数会先运行一个不执行任何处理的循环，并重复适当次数（NLOOP_FOR_ESTIMATION），然后通过计算单次循环所消耗的时间，来推算消耗 1 毫秒总共需要循环多少次。大家在自己的计算机上运行本程序时，如果推算过程占用太长时间，可以适当减小 NLOOP_FOR_ESTIMATION 的值。

因为实验程序的实现并非我们的主要目标，所以无法理解全部细节也没有关系。

编译这个 sched 程序，结果如下。

```
$ cc -c sched sched.c
$
```

4.3 实验

事不宜迟，下面我们利用 sched 程序来探究调度器的运作方式。本节将进行如下 3 个实验。

- **实验 4-A：运行 1 个进程时的情况**
- **实验 4-B：运行 2 个进程时的情况**
- **实验 4-C：运行 4 个进程时的情况**

利用负载均衡（后述）功能，进程能够根据系统的负载跨逻辑 CPU 运行。不过在本次的实验中，为了提高实验的精确度，我们令实验程序只能运行在单个逻辑 CPU 上，这可以通过 OS 提供的 taskset 命令来实现。如下添加 -c 选项，令参数中指定的程序仅运行在指定的逻辑 CPU 上。

```
$ taskset -c 0 ./sched <n> <total> <resol>
```

在执行上述命令后，sched 程序就只能运行在 0 号逻辑 CPU 上了。

需要注意的是，应当尽量在系统没有执行其他处理时进行实验，否则有可能导致程序无法正确运行。这一点是性能测试的原则，也适用于本书之后的所有实验程序。

各个实验中 sched 程序的参数如下所示。

实验名称	n	total	resol
实验 4-A	1	100	1
实验 4-B	2	100	1
实验 4-C	4	100	1

● 实验 4-A（进程数量=1）

程序运行结果如下所示。

```
$ taskset -c 0 ./sched 1 100 1
estimating workload which takes just one milisecond
end estimation
0     1       1
0     2       2
0     3       3
0     4       4
0     4       5
( 略 )
0     96      96
0     97      97
0     98      98
0     99      99
0     100     100
$
```

大家在自己的计算机上采集的数据或许会与上面展示的数据存在些许区别，但无须在意。这里的重点并非其中数值的绝对值，而是根据数据制作的图表的形状。

下面我们把实验结果制作成如下两个图表。

【图表①】在逻辑 CPU 上运行的进程

x 轴：开始运行后经过的时间（单位：毫秒）

y 轴：进程编号

【图表②】各个进程的进度

x 轴：开始运行后经过的时间（单位：毫秒）

y 轴：进度（单位：%。0 表示什么都还没开始处理，100 表示全部处理完成）

为方便制作图表，建议事先把运行结果保存为文件。通过如下所示的命令，即可把实验 4-A 的结果保存到 1core-1process.txt 文件中。

```
$ taskset -c 0 ./sched 1 100 1 >1core-1process.txt
$
```

图 4-3 和图 4-4 是使用实验 4-A 的数据制作的两个图表。

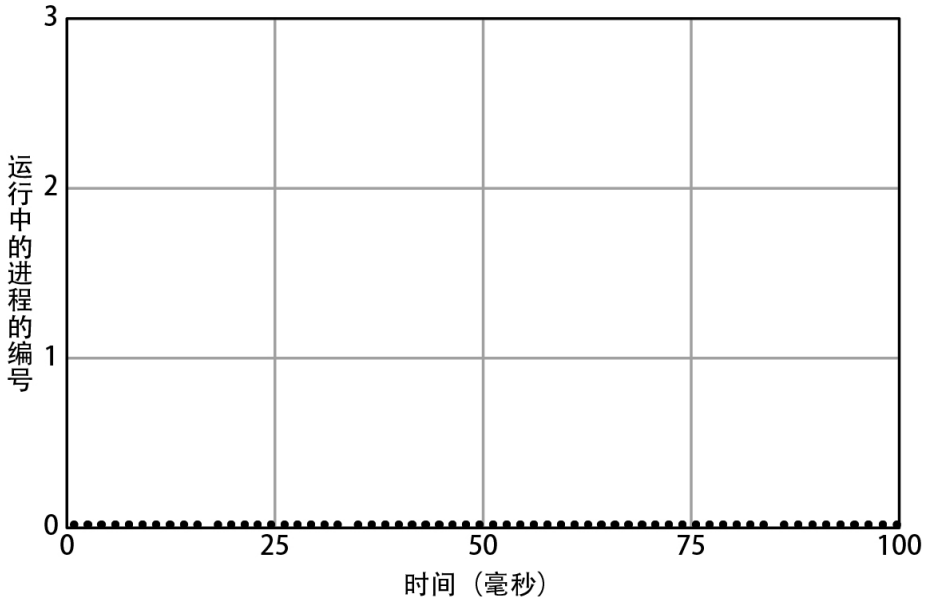

图 4-3　在逻辑 CPU 上运行的进程（实验 4-A，图表①）

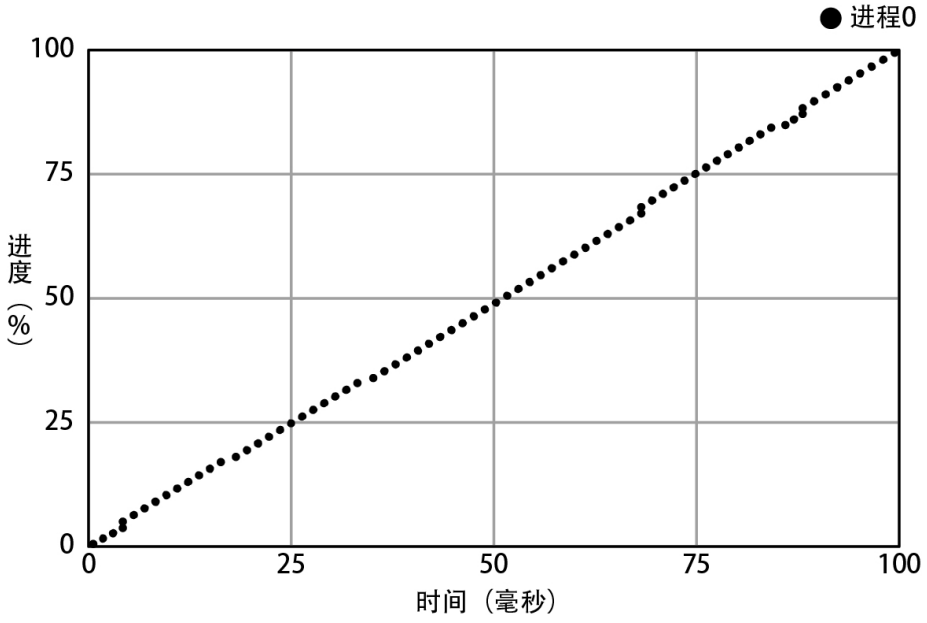

图 4-4　进程 0 的进度（实验 4-A，图表②）

图表①（图 4-3）显示，只有一个进程（进程 0）在持续运行。

由图表②（图 4-4）可以看出，由于只有进程 0 在运行，所以其进度简单地随着时间推移而等比例地增加。

● 实验 4-B（进程数量=2）

然后，根据实验 4-B 的结果制作图表，如图 4-5 和图 4-6 所示。

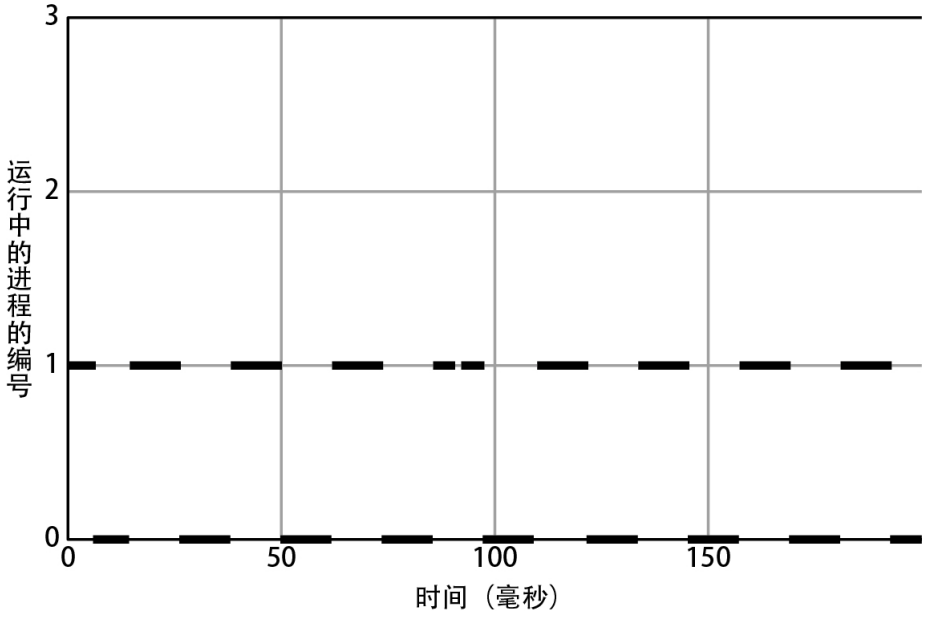

图 4-5　在逻辑 CPU 上运行的进程（实验 4-B，图表①）

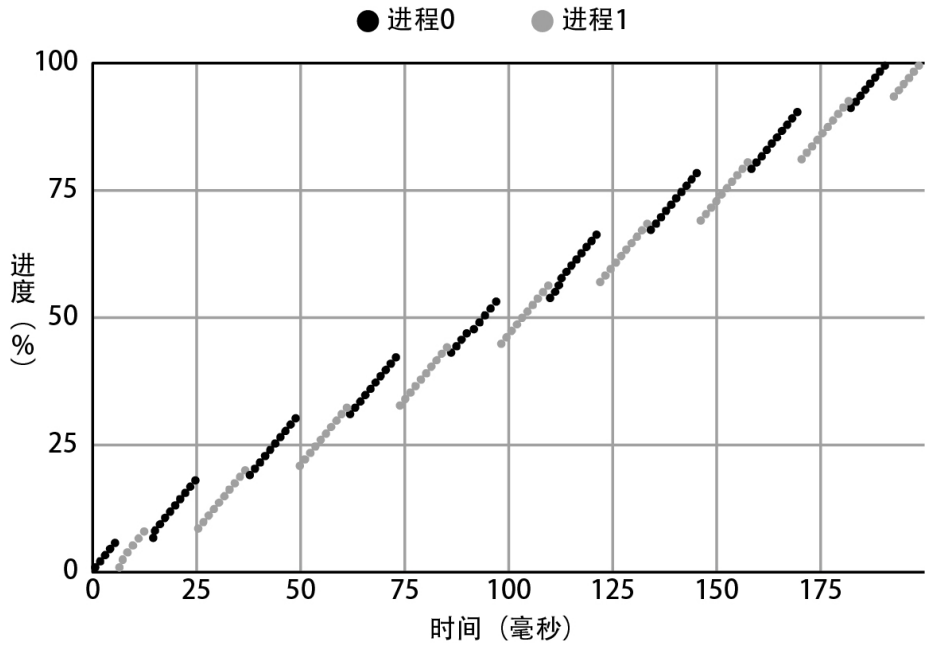

图 4-6　进程 0 与进程 1 的进度（实验 4-B，图表②）

通过图表①（图 4-5）可以得出以下结论。

- 2 个进程（进程 0 与进程 1）在轮流使用逻辑 CPU。也就是说，并非同时使用逻辑 CPU
- 2 个进程所获得的时间片（几毫秒）几乎相等

通过图表②（图 4-6）可以得出以下结论。

- 每个进程都只在使用逻辑 CPU 期间推进进度，换一种说法就是，在别的进程运行时并不会推进进度
- 单位时间推进的进度约为运行单个进程时的 1/2。在进程数量为 1 时，每毫秒推进 1% 左右；在进程数量为 2 时，每毫秒推进 0.5% 左右
- 处理完所有进程所消耗的时间约为运行单个进程时的 2 倍

● 实验 4-C（进程数量=4）

接下来是实验 4-C 的情况。

这次实验得到的图表①（图 4-7）和前一个实验一样，并非同时运行各个进程。尽管每个进程获得的时间片的长度略微不同，但每个进程

最后都大致消耗了相等的 CPU 时间。

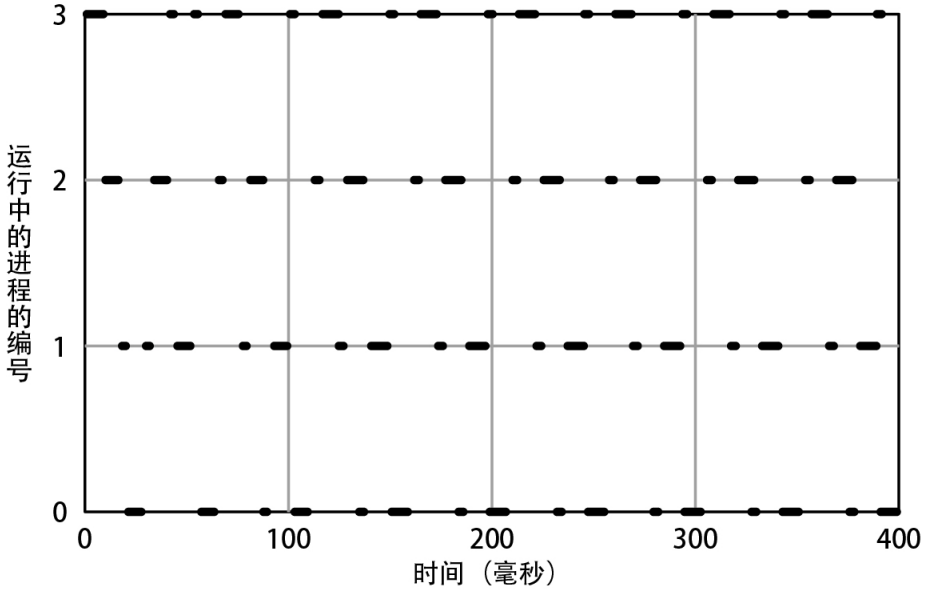

图 4-7　在逻辑 CPU 上运行的进程（实验 4-C，图表①）

把每个进程的进度制作成图表②（图 4-8）。

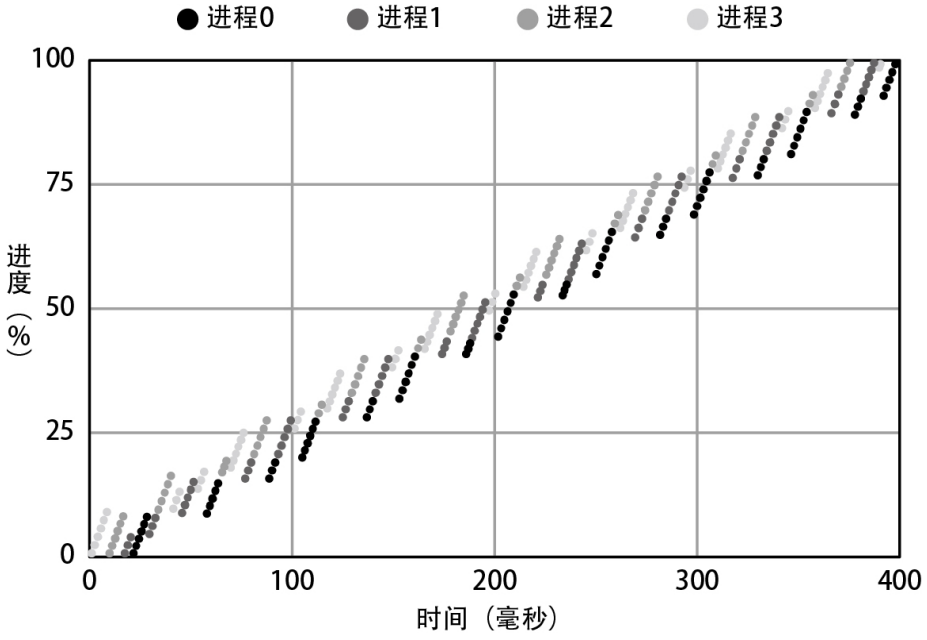

图 4-8　进程 0 ~进程 3 的进度（实验 4-C，图表②）

图表②的结果也与前一个实验大致相同。单位时间的进度约为进程数量为 1 时的 1/4。全部进程运行结束所消耗的时间约为进程数量为 1 时的 4 倍。

4.4 思考

通过以上 3 个实验，我们可以得出以下结论。

- 不管同时运行多少个进程，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 上运行多个进程时，它们将按轮询调度的方式循环运行，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流运行
- 每个进程被分配到的时间片的长度大体上相等
- 全部进程运行结束所消耗的时间，随着进程数量的增加而等比例地增加

4.5　上下文切换

上下文切换是指切换正在逻辑 CPU 上运行的进程。图 4-9 所示为当存在进程 0 和进程 1 时，在消耗完一个时间片后进行上下文切换的情况。

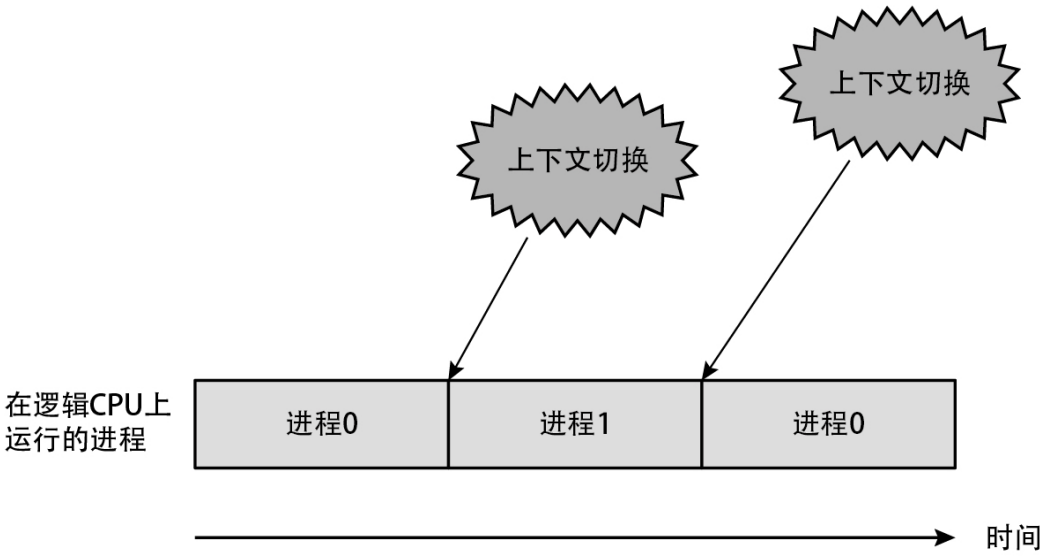

图 4-9　发生上下文切换

当一个时间片被消耗完后，不管进程正在执行什么代码，都一定会发生上下文切换。如果不理解这一点，就容易产生如图 4-10 所示的误会。

图 4-10　程序中的函数的执行时机（产生误会的例子）

但在现实的 Linux 中，并不能保证 bar() 紧接在 foo() 之后执行。如果 foo() 执行完后刚好消耗完时间片，则 bar() 的执行就会延后，如图 4-11 所示。

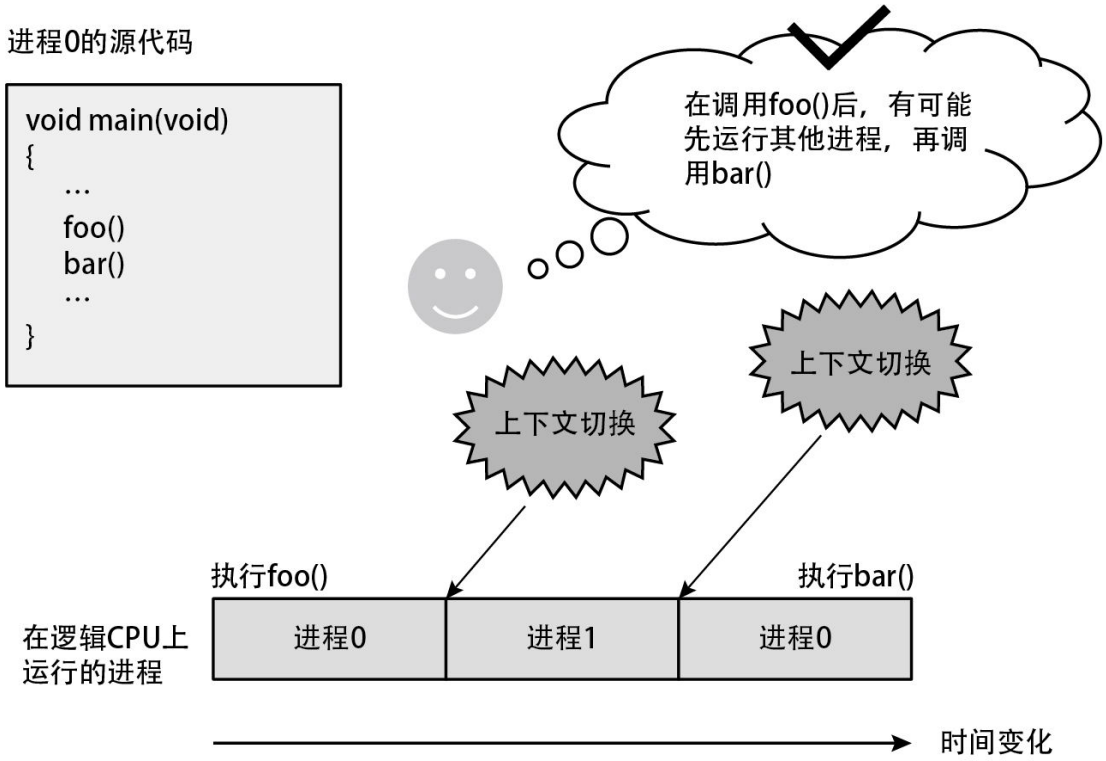

图 4-11　程序中的函数的执行时机（正确理解的例子）

理解了这一点之后，当出现某个处理消耗过长时间的情况时，你就不会理所当然地得出“肯定是处理本身有问题”这种结论，而会考虑到“可能发生了上下文切换，正在处理别的进程”等。