通过上表可以得出以下结论。

- 在耗尽逻辑 CPU 的计算能力后，也就是说，当所有逻辑 CPU 都不处于空闲状态后，不管继续增加多少个进程，吞吐量都不会再发生变化 [3]
- 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

[3]更加准确的说法是，如果在 `%idle` 的值变成 0 后进程数量继续增加，则上下文切换的系统开销增加，从而导致吞吐量降低。

下面对最后一点进行补充说明。在允许运行多个进程时，假如调度器并没有通过轮询调度的方式进行调度，结果就会变成如图 4-23 所示的情况，也就是在运行完一个进程后再开始调度下一个进程。

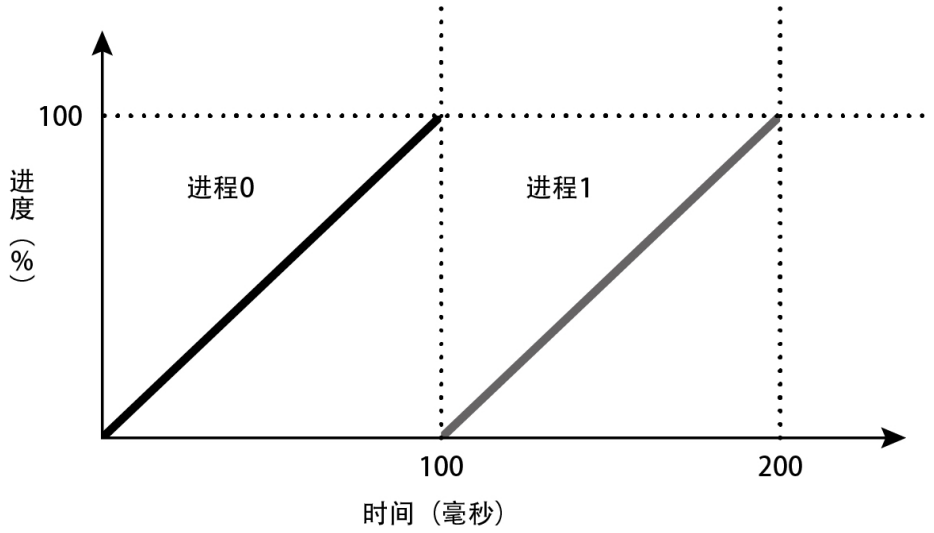

图 4-23　调度器未采用轮询调度时的情况

在这种情况下，虽然吞吐量不变，进程 0 和进程 1 也是同时开始运行的，但是前者的延迟为 100 毫秒，而后者的延迟为 200 毫秒。也就是说，出现了“不公平”的情况。正是为了避免这样的情况出现，调度器才把逻辑 CPU 的 CPU 时间划分为时长很短的时间片来分配给各个进程。

4.11 现实中的系统

首先总结一下前面的内容：当逻辑 CPU 始终处于工作状态（即没有空闲状态）并且没有程序处于就绪态时，吞吐量与延迟就是最优值。但是，现实中并没有这么多恰好的事情。在现实中的系统中，逻辑 CPU 会在下列状态之间不断转换。

- 空闲状态。由于逻辑 CPU 处于空闲状态，所以吞吐量会降低
- 进程正在运行，且没有处于就绪态的进程，这是一种理想的状态。如果在这样的状态下加入一个处于就绪态的进程，则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，但是延迟会变长

看到这里，想必大家都已经明白了，在大多数情况下，吞吐量与延迟是此消彼长的关系。

我们在设计系统时，为了使系统达到目标性能，也会对系统进行优化。比如，基于下列数据进行优化。

- **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）

下面，我们使用 4.8 节的 loop.py 程序，举例介绍一下 runq-sz 字段。

```
$ sar -q 1 1
( 略 )
11:28:28  runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15 blocked
11:28:29        0      831    6.17    5.17     2.46        0
          ←没有处于运行态或就绪态的进程，即系统处于空闲状态
Average:        0      831    6.17    5.17     2.46        0
$ taskset -c 0 python3 ./loop.py &  ←在逻辑CPU0上运行一个无限循环程序

[1] 9649
$ sar -q 1 1
( 略 )
```

```
11:28:42  runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15 blocked
10:28:43         1      831    4.88     4.93     2.43         0
                ←存在一个处于运行态或就绪态的进程，即在逻辑CPU0上正在运行一个
无限循环程序

Average:         1      831    4.88     4.93     2.43         0
$ taskset -c 0 python3 ./loop.py &  ←运行另一个上面那样的程序
[2] 9655
$ sar -q 1 1
( 略 )
11:28:47  runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15 blocked
10:28:48         2      835    4.57     4.87     2.42         0
        ←存在两个处于运行态或就绪态的进程，即在逻辑CPU0上有两个无限循环程序，轮
流进入就绪态和运行态

Average:         2      835    4.57     4.87     2.42         0
```

在运行结束后，记得结束正在运行的程序。

```
$ kill 9649 9655
$
```

4.12　存在多个逻辑 CPU 时的调度

当存在多个逻辑 CPU 时，如何进行调度呢？为了能够利用各个逻辑 CPU，调度器会运行一个被称为**负载均衡**或**全局调度**的功能。简单来说，负载均衡负责公平地把进程分配给多个逻辑 CPU。与只有单个逻辑 CPU 时的情况相同，在各个逻辑 CPU 内，调度器为在逻辑 CPU 上运行的各个进程分配均等的 CPU 时间。

图 4-24 所示为在 CPU0 和 CPU1 这 2 个逻辑 CPU 变为空闲状态后，按顺序开始运行 4 个进程（进程 0 ~进程 3）时的情形。

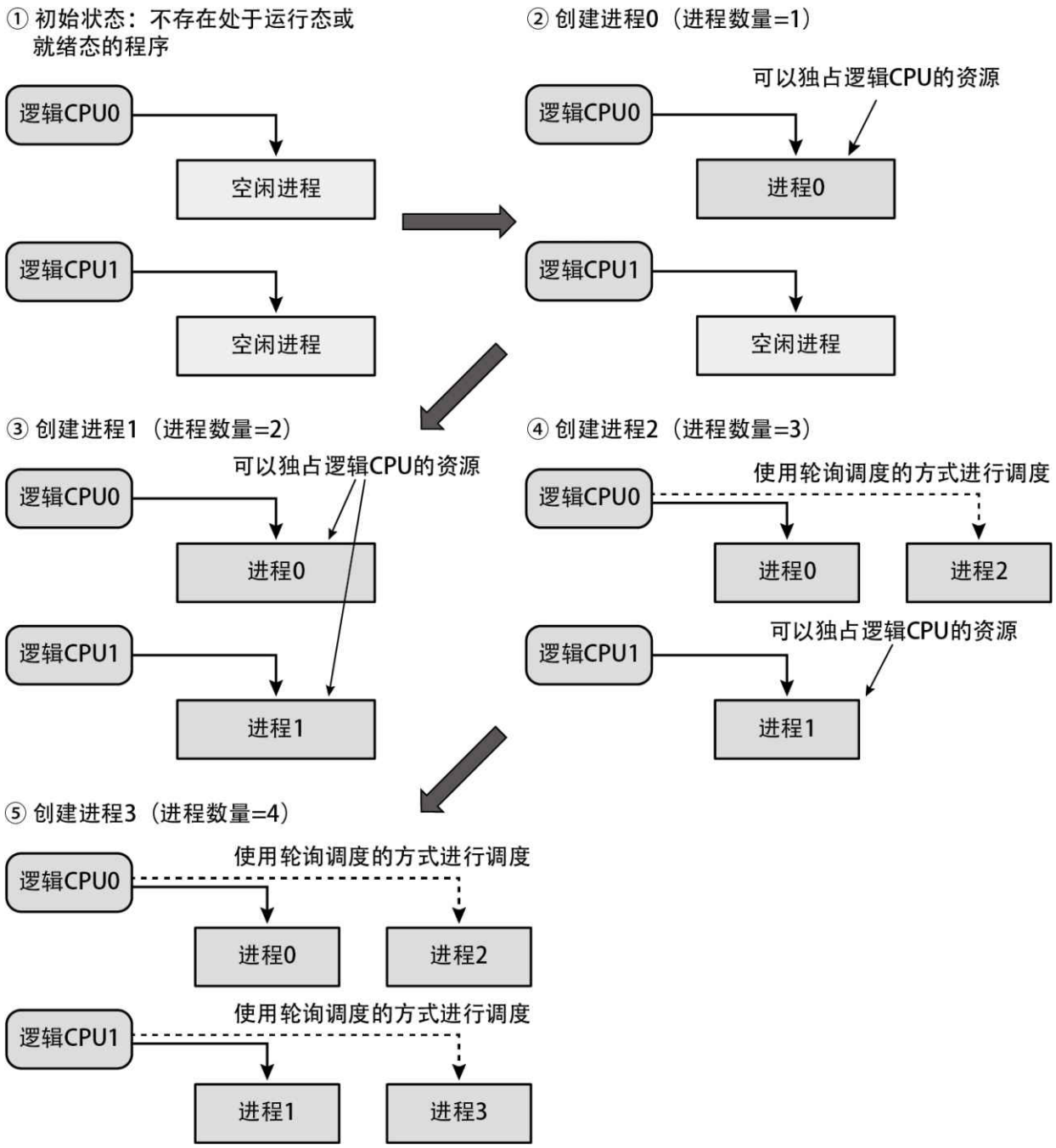

图 4-24　在 2 个逻辑 CPU 上创建并运行进程 0 ~进程 3 时的情形

接下来，我们通过实验来确认一下。

4.13 实验方法

只需利用前面使用过的程序，令其运行在多个逻辑 CPU 上即可。首先需要确认一下计算机上搭载了多少个逻辑 CPU。通过计算单词 processor 在 *proc*cpuinfo 这一保存了各个逻辑 CPU 的详细信息的文件中总共出现了多少次，即可得知逻辑 CPU 的数量。

```
$ grep -c processor proccpuinfo
8
```

显而易见，在笔者的计算机中识别出了 8 个逻辑 CPU。接下来，令 sched 程序运行在逻辑 CPU0 和逻辑 CPU4 上。

```
$ taskset -c 0,4 ./sched 进程数量 100 1
```

为什么不令其运行在 CPU0 和 CPU1 上呢？简单来说，由于 CPU0 和 CPU4 没有共用高速缓存（详见第 6 章），所以彼此高度独立，非常适合用于 sched 程序的性能测试。大家在运行这个程序时，只要选择逻辑 CPU0，以及编号为“逻辑 CPU 数量 / 2”的逻辑 CPU 即可。

需要注意的是，请在能够识别出多个逻辑 CPU 的计算机上运行 shced 程序。如果你的运行环境只能识别出单个逻辑 CPU，那么请参考笔者的实验结果。

另外，开启了超线程的计算机，以及系统只识别出 2 个逻辑 CPU 的计算机虽然可以运行实验程序，但是会得到意料之外的结果。超线程的相关内容将在第 6 章详细介绍。

下表所示为 sched 程序的参数。

实验名称	n	total	resol	
实验 4	D	1	100	1

实验名称	n	total	resol	
实验 4	E	2	100	1
实验 4	F	4	100	1

4.14　实验结果

● 实验 4-D（进程数量=1）

图 4-25 所示为只有 1 个进程时的情况。

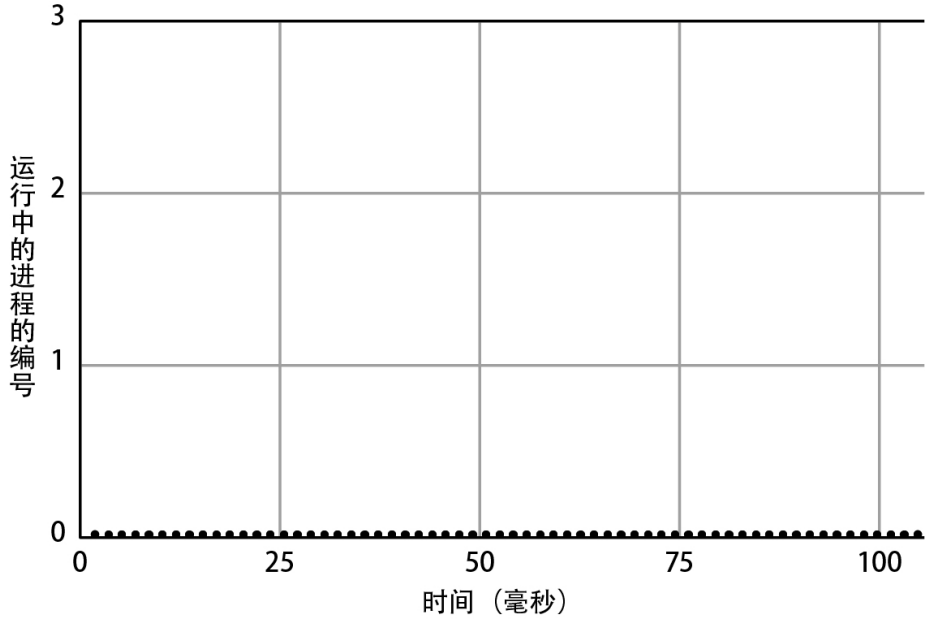

图 4-25　在逻辑 CPU 上运行的进程（实验 4-D，图表①）

该图和在单个逻辑 CPU 上测试时绘制的图表（图 4-3）一样。进程 0 始终运行在其中一个逻辑 CPU 上，另一个逻辑 CPU 处于空闲状态。

进程的进度如图 4-26 所示。

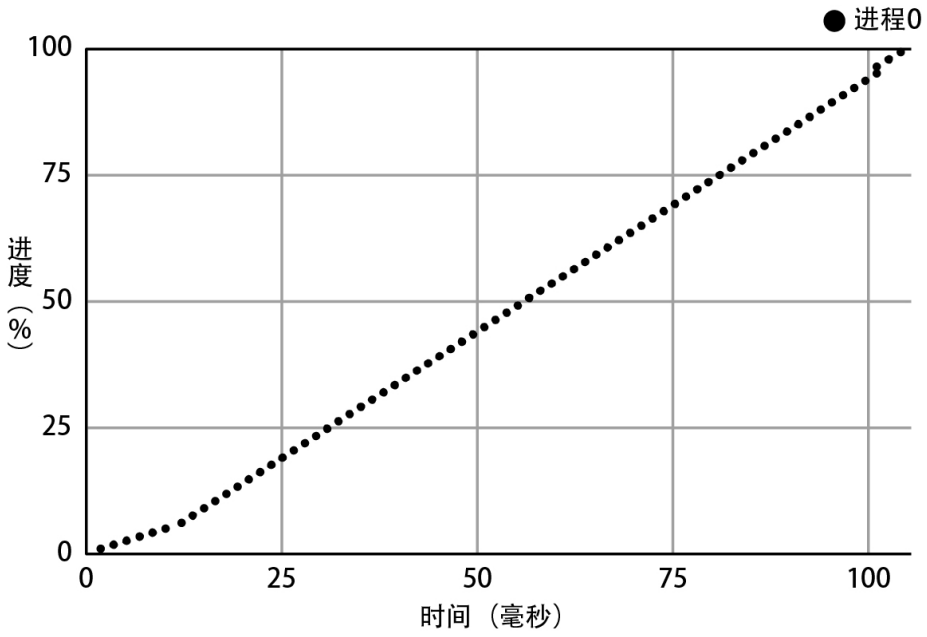

图 4-26　进程 0 的进度（实验 4-D，图表②）

该图也和在单个逻辑 CPU 上测试时绘制的图表（图 4-4）一样。

● 实验 4-E（进程数量=2）

图 4-27 所示为运行 2 个进程时的情况。可以看出，进程 0 和进程 1 分别在各自的逻辑 CPU 上同时运行着。由于没有处于空闲状态的逻辑 CPU，所以进程 0 与进程 1 处于最大限度地利用运算资源的状态。

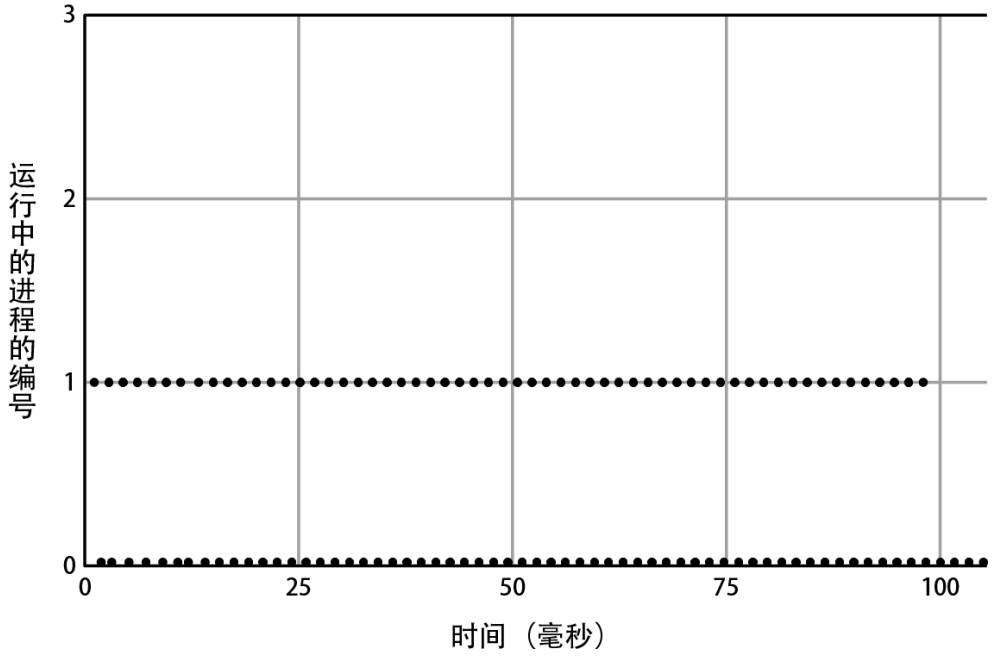

图 4-27　在逻辑 CPU 上运行的进程（实验 4-E，图表①）

各个进程的进度如图 4-28 所示。因为 2 个进程分别独自占用了 1 个逻辑 CPU，所以与只有单个逻辑 CPU 时的情况相比，2 个进程的处理时间都减少了一半。

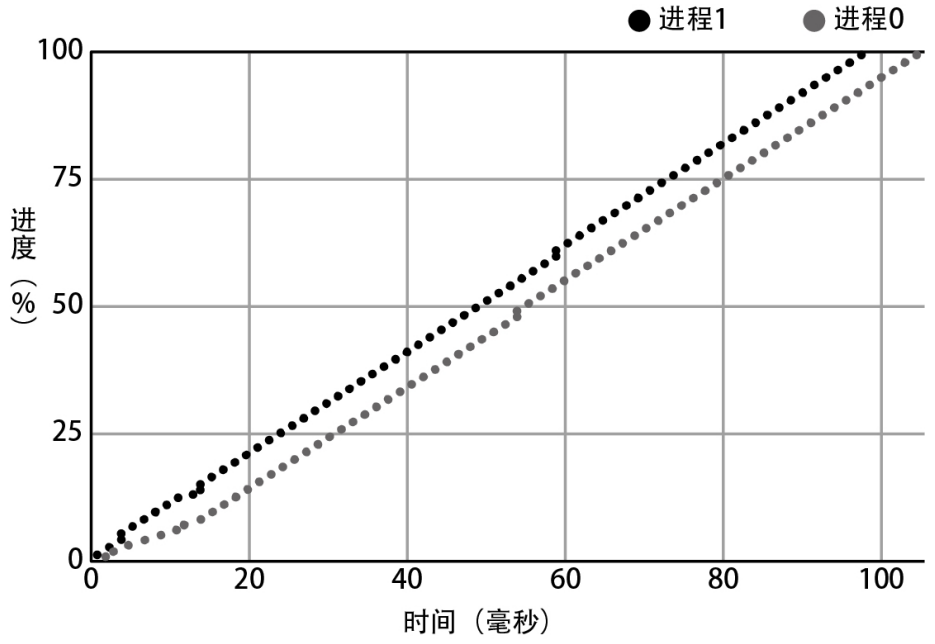

图 4-28　进程 0 与进程 1 的进度（实验 4-E，图表②）

● 实验 4-F（进程数量=4）

图 4-29 所示为运行 4 个进程时的情况。可以看出，每个逻辑 CPU 被分配 2 个进程，2 个进程在同一个逻辑 CPU 上轮流运行。

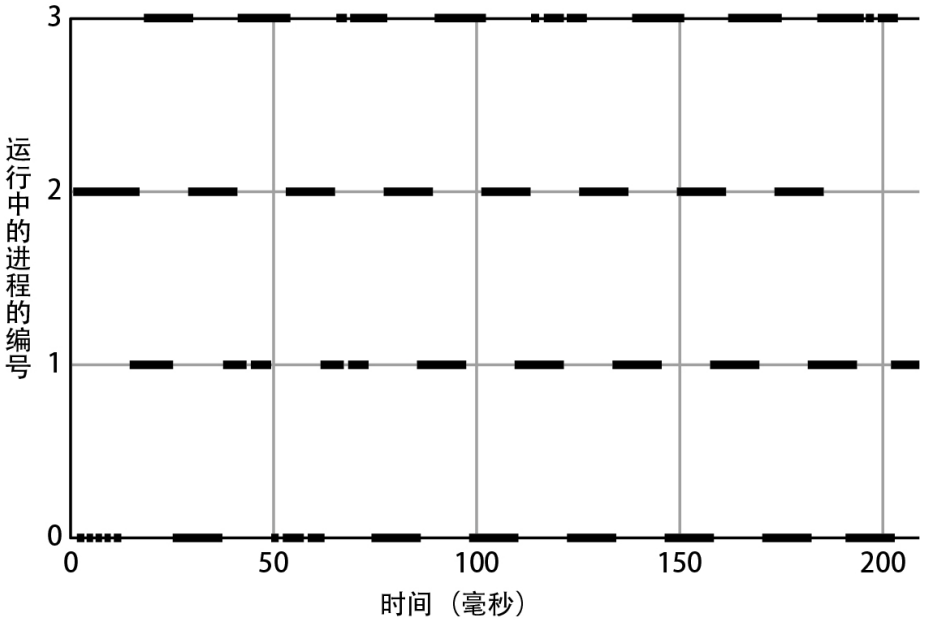

图 4-29　在逻辑 CPU 上运行的进程（实验 4-F，图表①）

这时，各个进程的进度如图 4-30 所示。

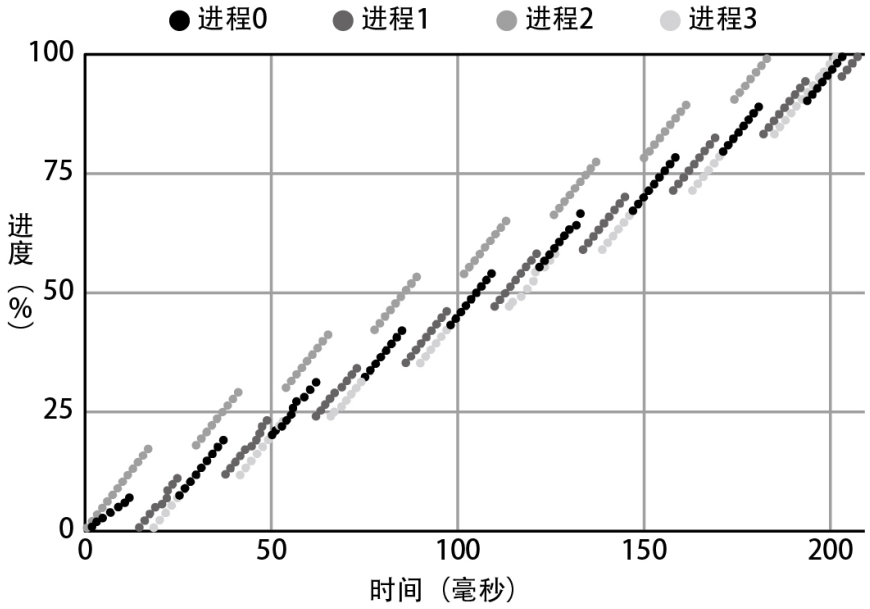

图 4-30　进程 0 ~进程 3 的进度（实验 4-F，图表②）

可以看出，各个进程几乎同时在推进进度。最后，每个进程所消耗的时间都比独占逻辑 CPU 进行处理时多了一倍。

4.15 吞吐量与延迟

接下来，我们基于实验 4-D ～实验 4-F 的结果计算吞吐量和延迟。

● 实验 4-D

在实验中 4-D，1 个进程在开始运行 100 毫秒后结束运行，所以其吞吐量和延迟的计算如下所示。

$$
\begin{aligned}
\text{吞吐量} &= 1 \text{ 个进程} / 100 \text{ 毫秒} \\
&= 1 \text{ 个进程} / 0.1 \text{ 秒} \\
&= 10 \text{ 个进程} / \text{秒}
\end{aligned}
$$

$$
\text{延迟} = 100 \text{ 毫秒}
$$

● 实验 4-E

在实验 4-E 中，2 个进程在开始运行 100 毫秒后几乎同时结束运行，所以其吞吐量和延迟的计算如下所示。

$$
\begin{aligned}
\text{吞吐量} &= 2 \text{ 个进程} / 100 \text{ 毫秒} \\
&= 2 \text{ 个进程} / 0.1 \text{ 秒} \\
&= 20 \text{ 个进程} / \text{秒}
\end{aligned}
$$

$$
\text{延迟} = 100 \text{ 毫秒}
$$

● 实验 4-F

在实验 4-F 中，4 个进程在开始运行 200 毫秒后几乎同时结束运行，所以其吞吐量和延迟的计算如下所示。

$$
\begin{aligned}
\text{吞吐量} &= 4 \text{ 个进程} / 200 \text{ 毫秒} \\
&= 4 \text{ 个进程} / 0.2 \text{ 秒} \\
&= 20 \text{ 个进程} / \text{秒}
\end{aligned}
$$

$$
\text{延迟} = 200 \text{ 毫秒}
$$

我们把计算结果汇总成下表。

进程数量	吞吐量（进程数量 / 秒）	延迟（毫秒）
1	10	100
2	20	100
4	20	200

4.16 思考

这些数据印证了本章开头提到的以下两点内容。

- 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 上执行处理

除此之外，我们还能得出以下结论。

- 对于多核 CPU 的计算机来说，只有同时运行多个进程才能提高吞吐量。另外，“有 n 个核心就有 n 倍性能”这种说法，说到底也只存在于理想状态中
- 与只有单个逻辑 CPU 时一样，当进程数量多于逻辑 CPU 数量时，吞吐量就不会再提高