4.6 进程的状态

系统中究竟运行着多少个进程呢？使用 ps ax 命令可以按照一行一个进程的格式列举出系统中当前正在运行的所有进程。然后，根据执行该命令而输出的行数，就能直接得知正在运行的进程总数。在笔者的计算机上，该命令的执行结果如下所示。

```
$ ps ax | wc -l
365
```

一眼就能看出有 300 多个进程正在运行。在不同的计算机上，这个数值可能有所不同，而且每次执行得到的结果也会略微不同，不过我们不需要在意这些细微的差别。

我们通过前面的实验了解到，在 sched 程序运行时，仅为 sched 程序内的进程分配 CPU 时间。那么，这时系统内的其他进程到底在干什么呢？实际上，进程存在几种状态，系统上的大部分进程处于**睡眠态**。

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回收

上表并没有列出 Linux 中所有的进程状态，不过暂时记住这些就没问题了。

举例来说，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（比如等待 3 分钟）
- 等待用户通过键盘或鼠标等设备进行输入
- 等待 HDD 或 SDD 等外部存储器的读写结束
- 等待网络的数据收发结束

通过查看 ps ax 的输出结果中的第 3 个字段 STAT 的首字母，就可以得知进程处于哪种状态。

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 以外的情况（D 主要出现于等待外部存储器的访问时）
Z	僵死状态

接下来我们看一下实际存在于系统中的进程的状态。

```
$ ps ax
( 略 )
10533 pts/24  Ss     0:00 bin/bash --noediting -i
10759 ?       Ss     0:00 sshd: root [priv]
10760 ?       S      0:00 sshd: root [net]
10761 pts/24  R+     0:00 ps ax
15599 ?       Ssl    0:00 /usr/lib/x86_64-linux-gnu/unity/ ↲
unity-panel-service --lockscreen-mode
22857 ?       S<     0:00 [bioset]
22859 ?       S<     0:00 [xfsalloc]
22860 ?       S<     0:00 [xfs_mru_cache]
22869 ?       S      0:00 [jfsIO]
22870 ?       S      0:00 [jfsCommit]
22871 ?       S      0:00 [jfsCommit]
22872 ?       S      0:00 [jfsCommit]
22873 ?       S      0:00 [jfsCommit]
22874 ?       S      0:00 [jfsCommit]
22875 ?       S      0:00 [jfsCommit]
```

```
22876 ?        S      0:00 [jfsCommit]
22877 ?        S      0:00 [jfsCommit]
22878 ?        S      0:00 [jfsSync]
25057 ?        S<     0:00 [kworker/u33:0]
$
```

可以看到，确实大部分进程的状态标记为 S。ps ax 被标记为 R，这是因为该程序为了输出进程状态而正在运行中。另外，由于 bash 正在等待用户输入，所以它处于睡眠态[2]。

[2]对于第 3 个字段，现在无须在意首字母以外的内容（例如 S 后面的 <）。

需要注意的是，处于 D 状态的进程通常会在几毫秒之内迁移到别的状态。当出现长时间处于 D 状态的进程时，需要考虑是否发生了以下状况。

- 存储器的 I/O 处理尚未结束
- 内核中发生了某种问题

4.7 状态转换

图 4-12 所示为进程的各种状态之间的关联。由该图可知，进程在被创建后的整个生命周期中，会不断地在运行态、就绪态和睡眠态之间辗转，并非简单地使用完分配到的 CPU 时间后就立刻结束。

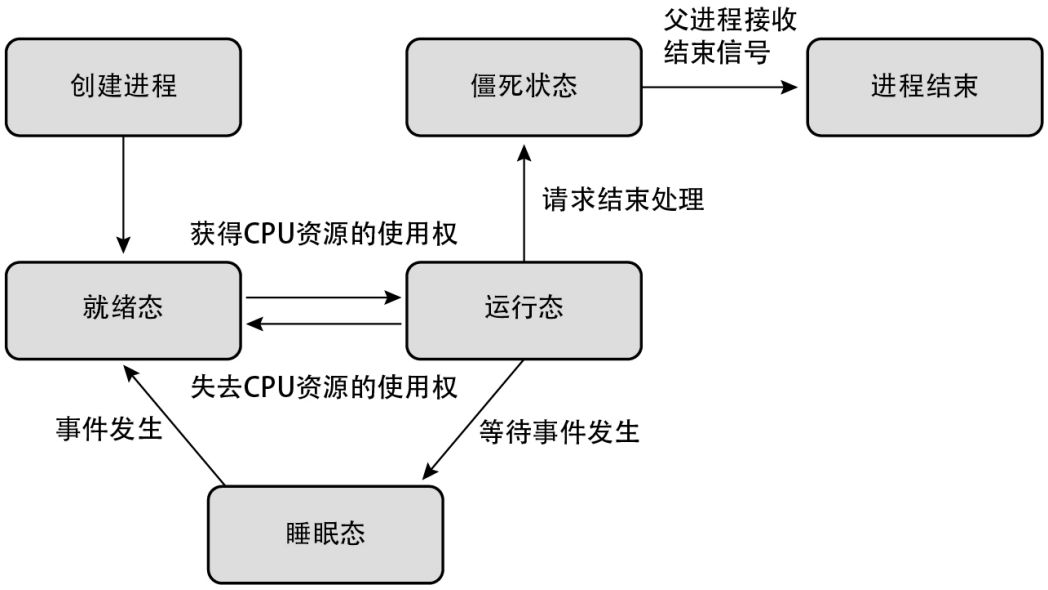

图 4-12 进程的各种状态

明白这一点后，我们来看几个状态转换的例子。

首先是最简单的不会进入睡眠态的进程。

图 4-13 所示为在 sched 程序中只运行 p0 这一个进程时该进程的状态转换，以及在此期间逻辑 CPU 上执行的处理。

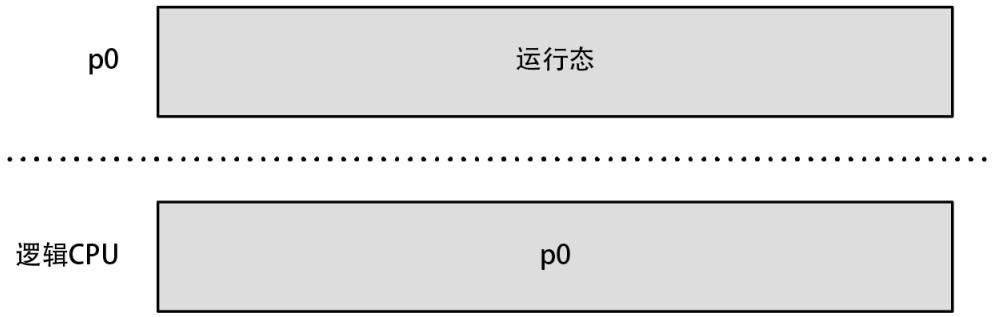

图 4-13　进程的状态以及逻辑 CPU 上执行的处理（进程不会进入睡眠态的情况）

实际上，在 p0 运行时，系统上还运行着许多其他进程，但它们全部（正确来说是绝大多数）处于睡眠态，所以图 4-13 中省略了这些进程。

运行 p0 与 p1 两个进程时的情况如图 4-14 所示。

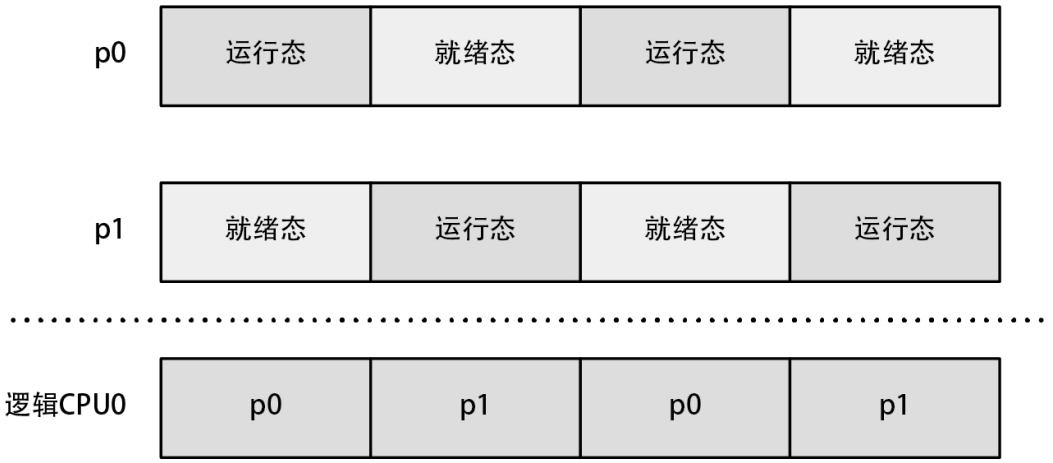

图 4-14　进程的状态以及逻辑 CPU 上执行的处理（运行 p0、p1 时的情况）

如果在逻辑 CPU 上运行着一个进程 p0，并且 p0 在运行期间睡眠过一次，情况就会变成如图 4-15 所示的那样。

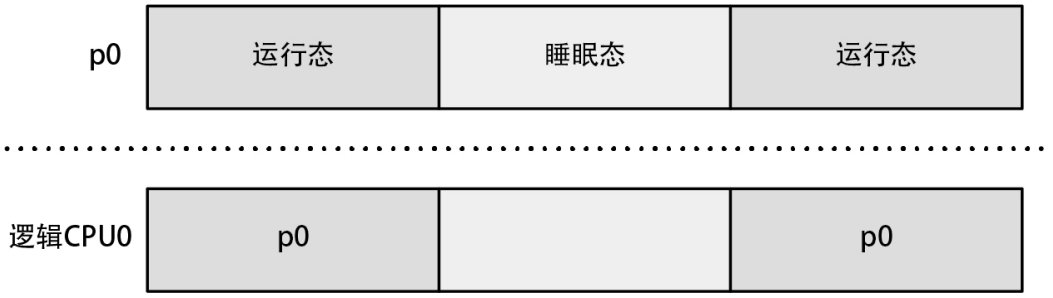

图 4-15　进程的状态以及逻辑 CPU 上执行的处理（进程进入睡眠态的情况）

4.8 空闲状态

在图 4-15 中，p0 有一段时间没有在 CPU0 上运行。在此期间，逻辑 CPU 上会发生什么呢？

实际上，在此期间，逻辑 CPU 会运行一个被称为**空闲进程**的不执行任何处理的特殊进程。空闲进程最简单的实现方式就是创建一个新进程，或者在唤醒处于睡眠态的进程之前执行无意义的循环。但因为这样会浪费电，所以通常并不会这样处理，而是使用特殊的 CPU 指令使逻辑 CPU 进入休眠状态，直到出现就绪态的进程。大家的计算机和智能手机之所以在不运行程序时能够待机更长时间，主要就得益于逻辑 CPU 能够进入空闲状态。

利用 sar 命令就可以确认单位时间内逻辑 CPU 处于空闲状态的时间占比，以及有多少空余的计算资源。

```
$ sar -P ALL 1
( 略 )
09:25:53  CPU  %user  %nice %system %iowait  %steal   %idle
09:25:54  all   0.50   0.00    0.12    0.00    0.00   99.38
09:25:54    0   0.00   0.00    0.00    0.00    0.00  100.00
09:25:54    1   0.00   0.00    0.00    0.00    0.00  100.00
09:25:54    2   0.00   0.00    0.00    0.00    0.00  100.00
09:25:54    3   0.00   0.00    0.00    0.00    0.00  100.00
09:25:54    4   1.00   0.00    0.00    0.00    0.00   99.00
09:25:54    5   1.01   0.00    0.00    0.00    0.00   98.99
09:25:54    6   0.99   0.00    1.00    0.00    0.00   99.00
09:25:54    7   0.99   0.00    0.00    0.00    0.00   99.01

09:25:54  CPU  %user  %nice %system %iowait  %steal   %idle
09:25:55  all   0.25   0.00    0.25    0.00    0.00   99.50
09:25:55    0   0.00   0.00    0.00    0.00    0.00  100.00
09:25:55    1   0.00   0.00    0.00    0.00    0.00  100.00
09:25:55    2   0.00   0.00    0.00    0.00    0.00  100.00
09:25:55    3   0.00   0.00    0.00    0.00    0.00  100.00
09:25:55    4   1.00   0.00    0.00    0.00    0.00   99.00
09:25:55    5   1.00   0.00    0.00    0.00    0.00   99.00
09:25:55    6   0.00   0.00    1.98    0.00    0.00   98.02
09:25:55    7   0.00   0.00    0.00    0.00    0.00  100.00
( 略 )
```

最后的 %idle 字段显示了 1 秒内空闲状态的时间占比。通过上面的数据可知，系统当前基本没有消耗 CPU 时间。

接下来，我们尝试在逻辑 CPU0 上运行无限循环的 Python 程序（代码清单 4-2），并在此期间采集数据。

代码清单 4-2　loop.py 程序（loop.py）

```
while True:
    pass
```

运行这个程序，结果如下。

```
$ taskset -c 0 python3 loop.py &
[4] 15009
$ sar -P ALL 1
( 略 )
09:54:40  CPU  %user  %nice %system %iowait  %steal   %idle
09:54:41  all  12.97   0.00    0.12    0.00    0.00   86.91
09:54:41    0 100.00   0.00    0.00    0.00    0.00    0.00
09:54:41    1   0.00   0.00    0.00    0.00    0.00  100.00
09:54:41    2   0.00   0.00    0.00    0.00    0.00  100.00
09:54:41    3   0.00   0.00    0.00    0.00    0.00  100.00
09:54:41    4   0.00   0.00    0.00    0.00    0.00  100.00
09:54:41    5   0.99   0.00    0.00    0.00    0.00   99.01
09:54:41    6   1.00   0.00    0.00    0.00    0.00   99.00
09:54:41    7   0.99   0.00    0.00    0.00    0.00   99.01

09:54:41  CPU  %user  %nice %system %iowait  %steal   %idle
09:54:42  all  13.00   0.00    0.00    0.00    0.00   87.00
09:54:42    0 100.00   0.00    0.00    0.00    0.00    0.00
09:54:42    1   0.00   0.00    0.00    0.00    0.00  100.00
09:54:42    2   0.00   0.00    0.00    0.00    0.00  100.00
09:54:42    3   0.00   0.00    0.00    0.00    0.00  100.00
09:54:42    4   2.00   0.00    0.00    0.00    0.00   98.00
09:54:42    5   2.00   0.00    0.00    0.00    0.00   98.00
09:54:42    6   0.00   0.00    1.00    0.00    0.00   99.00
09:54:42    7   1.00   0.00    0.00    0.00    0.00   99.00
( 略 )
```

可以看出，只有逻辑 CPU0 的 %idle 变成了 0，这是因为 loop.py 程序持续运行在逻辑 CPU0 上。

在采集完数据后，记得结束正在运行的程序。

```
$ kill 15009
$
```

4.9　各种各样的状态转换

在现实中的系统中，各个进程会根据不同的处理转换到各种不同的状态，在逻辑 CPU 上运行的进程也会随之发生变化。

我们一起来思考一下，执行以下处理的进程（这里称为进程 0）在运行时会发生什么。

① 接收用户的输入。

② 根据用户输入的信息读取文件。

在逻辑 CPU 上只存在进程 0 时，进程 0 的状态转换以及逻辑 CPU 在各种状态下执行的处理如图 4-16 所示。

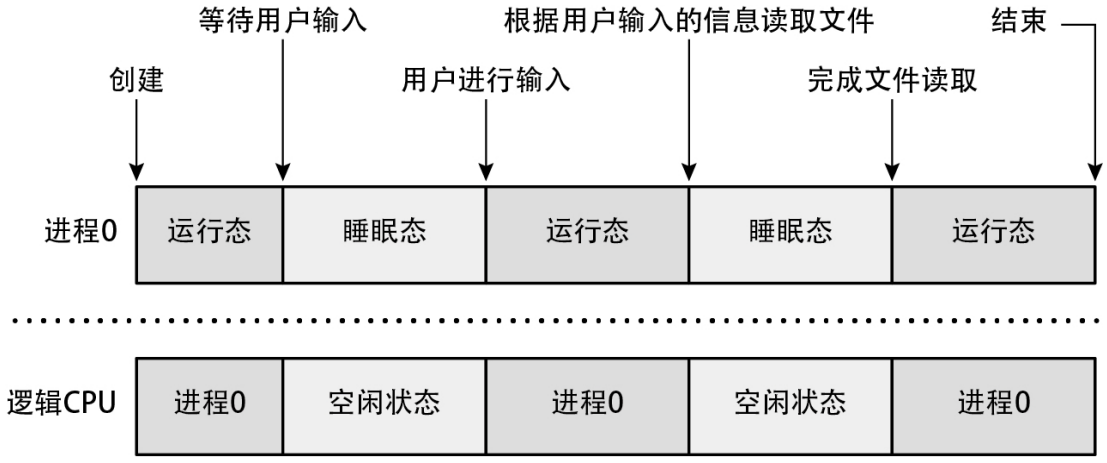

图 4-16　进程的状态以及逻辑 CPU 上执行的处理（只有进程 0 时的情况）

当存在多个这样的进程时，情况如图 4-17 所示。

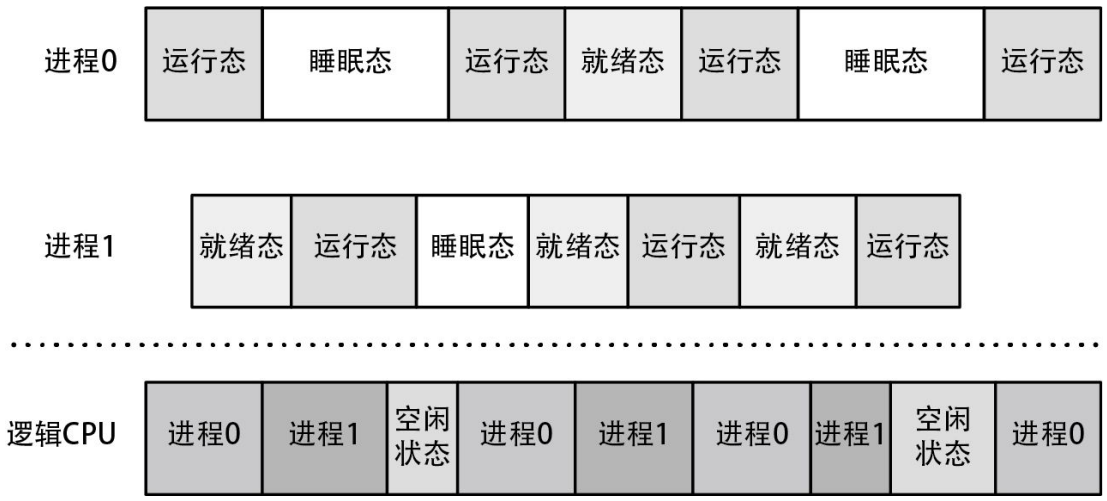

图 4-17　进程的状态以及逻辑 CPU 上执行的处理（存在多个进程时的情况）

虽然看起来比较复杂，但重点只有两个：①在逻辑 CPU 上同一时间只能运行一个进程；②睡眠态的进程不会占用 CPU 时间。如果你在开发或者使用软件时多少能想起来图 4-16 和图 4-17 的情况，那么一定可以加深对计算机系统的了解。

4.10　吞吐量与延迟

下面将介绍**吞吐量**和**延迟**这两个表示处理性能的指标的概念，它们的定义如下所示。

- **吞吐量：单位时间内的总工作量，越大越好**
- **延迟：各种处理从开始到完成所耗费的时间，越短越好**

这两个指标不仅适用于逻辑 CPU，对于评价其他硬件（例如外部存储器等）的性能来说也非常重要。但是，为了便于理解，这里将集中讲解逻辑 CPU 的处理性能。下面是这两个指标的计算公式。

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

我们先来看一下吞吐量。对于吞吐量，基本上可以简单地理解为，CPU 的计算资源消耗得越多，或者说空闲状态的时间占比越低，吞吐量就越大。

比如存在一个反复在“使用逻辑 CPU”和“进入睡眠态”之间转换的进程，这个进程和逻辑 CPU 的状态转换如图 4-18 所示。

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）

在这 100 毫秒内，逻辑 CPU 有 40 毫秒处于空闲状态（通过 `sar -P ALL` 可以看到，`%idle` 的值是 40）。当前的吞吐量的计算如下所示。

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

如果上述进程有 2 个，并且 2 个进程开始运行的时间存在时间差（虽然有点刻意），则此时的情况如图 4-19 所示。

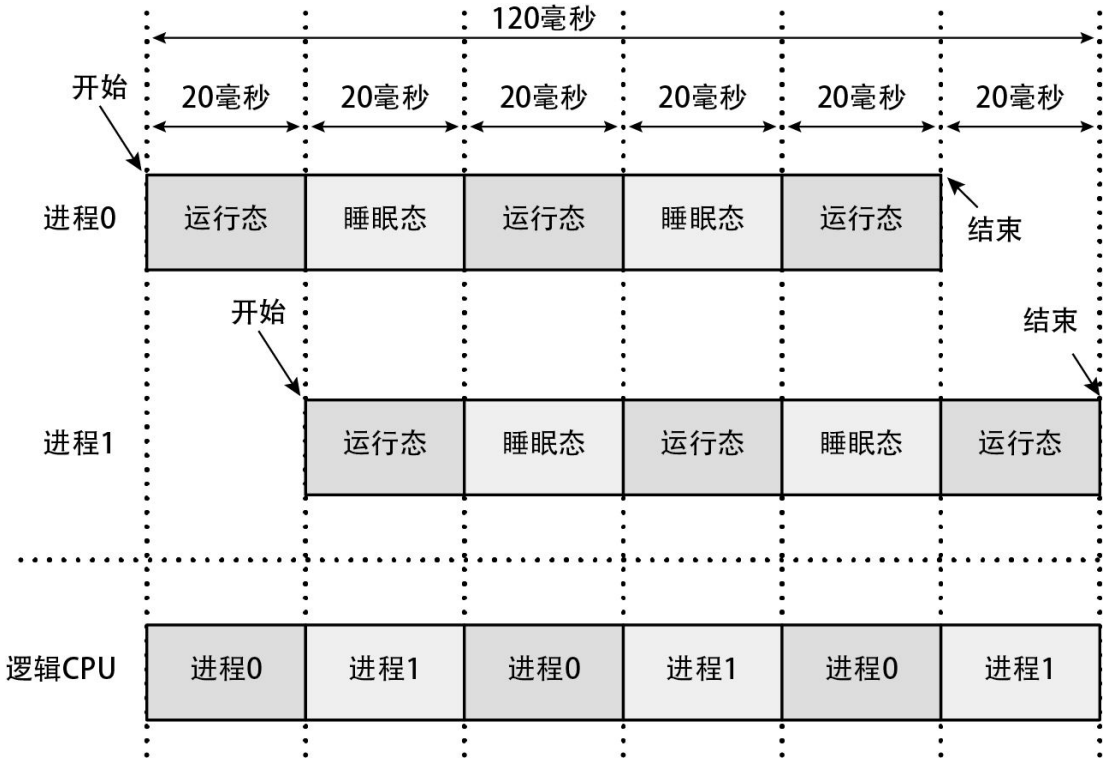

图 4-19　进程的状态以及逻辑 CPU 上执行的处理（存在 2 个经常进入睡眠态的进程时的情况）

这次逻辑 CPU 没有出现空闲状态。吞吐量的计算如下所示。

$$\begin{aligned}\text{吞吐量} &= 2\ \text{个进程} \ /\ 120\ \text{毫秒} \\ &= 2\ \text{个进程} \ /\ 0.12\ \text{秒} \\ &\approx 16.7\ \text{个进程} \ /\ \text{秒}\end{aligned}$$

我们将计算出来的数据汇总到下表中。

进程数量	空闲时间的占比（%）	吞吐量
1	40	10
2	0	16.7

通过上表可知，空闲时间的占比越低，吞吐量就越大。

接下来，我们把延迟也纳入考量范围。以前面的实验 4-A、实验 4-B 和实验 4-C 得到的数据作为对象，一边观察由实验得到的图表，一边思考其吞吐量和延迟。

当只有 1 个进程时，用 100 毫秒处理完 1 个进程（进程 0），如图 4-20 所示。

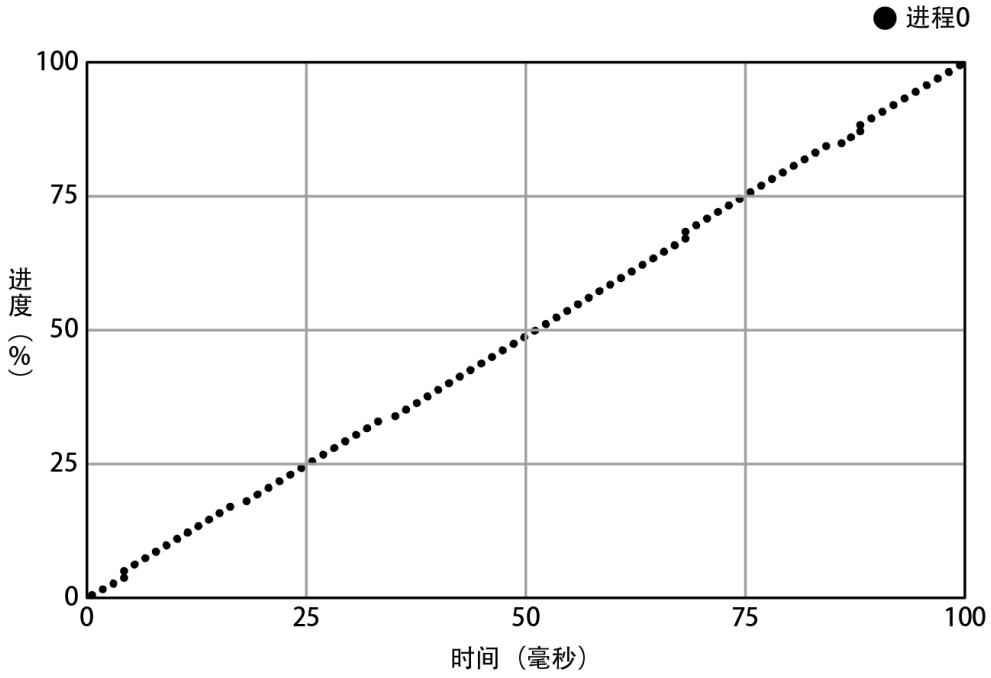

图 4-20　进程 0 的进度（同图 4-4）

吞吐量与延迟的计算如下所示。

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

$$
\begin{aligned}
\text{平均延迟} &= \text{进程 0 的延迟} \\
&= 100 \text{ 毫秒}
\end{aligned}
$$

当存在 2 个进程时，运行 2 个进程（进程 0、进程 1）共消耗 200 毫秒。另外，2 个进程几乎同时在 200 毫秒时运行结束，如图 4-21 所示。

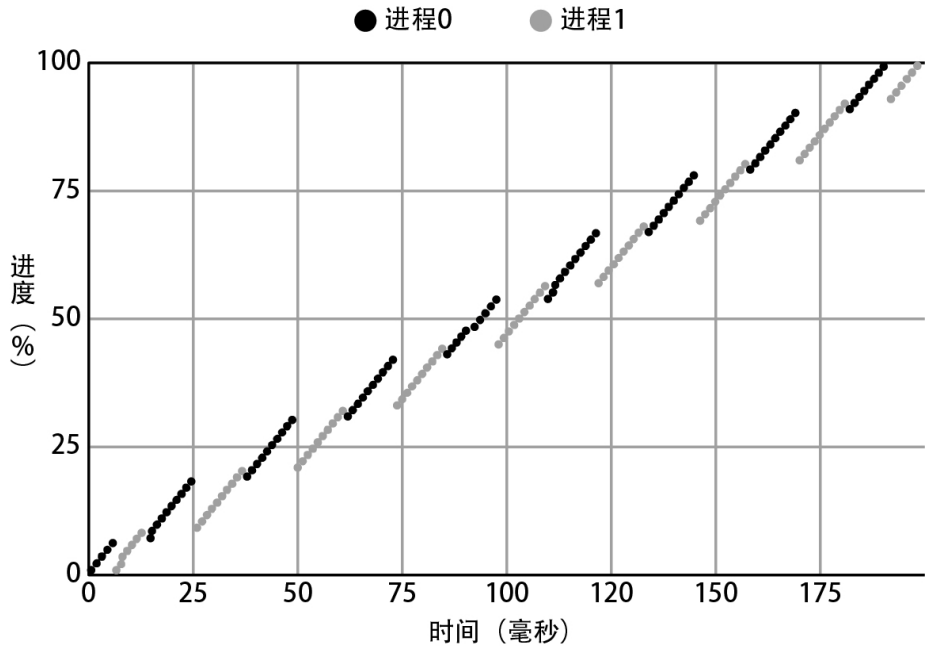

图 4-21　进程 0 与进程 1 的进度（同图 4-6）

吞吐量与延迟的计算如下所示。

$$
\begin{aligned}
\text{吞吐量} &= 2 \text{ 个进程} / 0.2 \text{ 秒} \\
&= 10 \text{ 个进程} / \text{秒}
\end{aligned}
$$

$$
\begin{aligned}
\text{平均延迟} &= \text{进程 0 与进程 1 的延迟} \\
&= 200 \text{ 毫秒}
\end{aligned}
$$

当存在 4 个进程时，运行 4 个进程（进程 0 ~进程 3）共消耗 400 毫秒。另外，4 个进程几乎同时在 400 毫秒时结束运行，如图 4-22

所示。

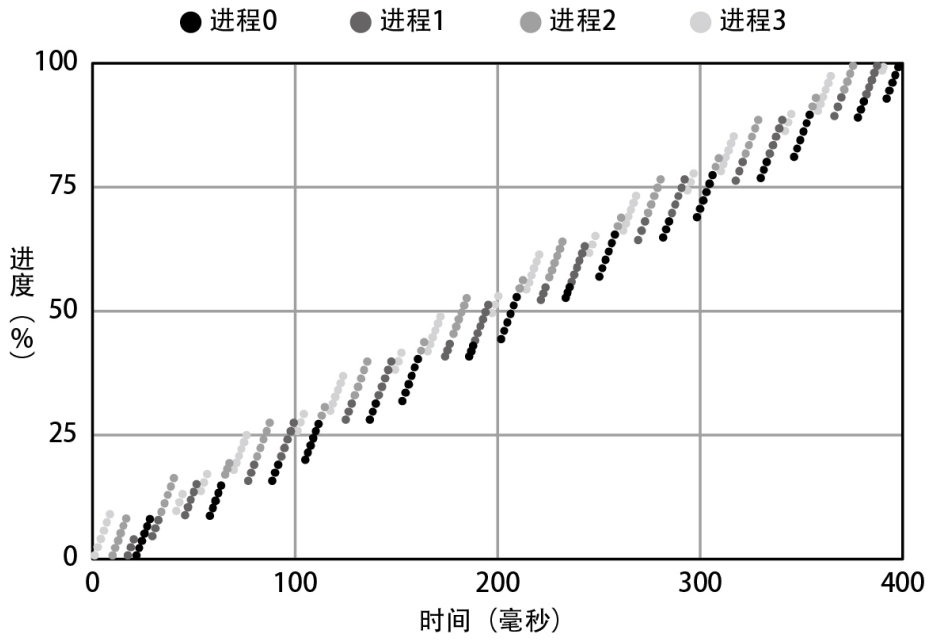

图 4-22　进程 0 ~进程 3 的进度（同图 4-8）

吞吐量与延迟的计算如下所示。

$$\text{吞吐量} = 4 \text{ 个进程} / 0.4 \text{ 秒} = 10 \text{ 个进程} / \text{秒}$$

$$\text{平均延迟} = \text{进程 0 ~进程 3 的延迟} = 400 \text{ 毫秒}$$

把所有计算结果汇总成下表。

进程数量	吞吐量（进程数量 / 秒）	平均延迟（毫秒）
1	10	100
2	10	200
4	10	400