Python

极夜酱

目录

Chapter 1 并发编程

1.1 多进程

1.1.1 中央处理器（CPU, Central Process Unit）

CPU 作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。

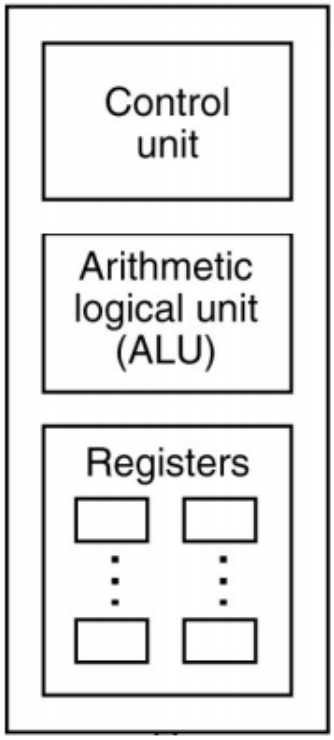

图 1.1: CPU 结构

Python 中可以通过 multiprocessing 模块获取计算机 CPU 的内核数量。

获取 CPU 可用数量

```
1 import multiprocessing  # 导入多进程模块
2 # 获取CPU的可用数量
3 print("CPU内核数量: %d" % multiprocessing.cpu_count())
```

CPU 的核心部分有：

- 控制单元（CU, Control Unit）：控制单元是 CPU 的子部件，它管理着计算机中所有在这一区域执行的操作。它负责从计算机、指令和数据中获取各种输入，并告诉处理器如何处理它们。
- 算术逻辑单元（ALU, Arithmetic and Logic Unit）：实现多组算术运算和逻辑运算的组合逻辑电路。
- 寄存器（register）：寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。

1.1.2 进程（Process）

Windows 任务管理器提供了有关计算机性能的信息，并显示了计算机上所运行的程序和进程的详细信息。

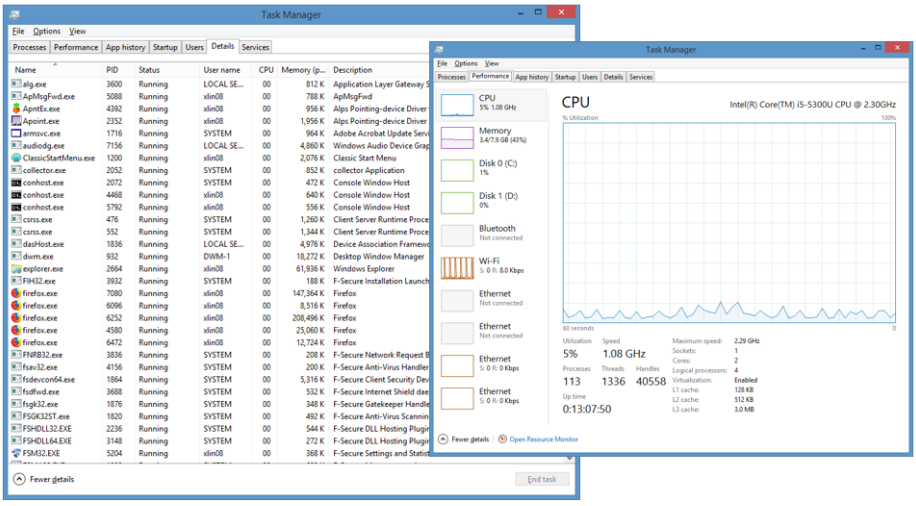

图 1.2: Windows 进程

进程指的是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。进程是系统进行资源分配和调度运行的基本单位。进程实体中包含三个组成部分：

1. 程序
2. 数据
3. 进程控制块（PCB, Process Control Block）

程序（program）与进程是有区别的，程序是静态的，进程是动态的。当程序的可执行文件被装入内存后就会变成进程。进程可以认为是执行中的程序，进程需要一定资源（如 CPU 时间、内存、文件、I/O 设备）完成任务。这些资源可以在创建的时候或者运行中分配。

1.1.3 并发编程（Concurrent）

并发编程是一种有效提高操作系统（服务器）性能的技术手段，现代的操作系统之中最为重要的代表就是并发性，例如现在的 CPU 都属于多核 CPU。

早期的 DOS 操作系统有一个非常重要的特征：一旦系统沾染了病毒，那么所有的程序就无法直接执行了。因为传统的 DOS 系统属于单进程模型，在同一个时间段上只能运行一个程序，病毒程序运行了，其它程序自然就无法运行。

后来到了 Windows 操作系统，即便有病毒，也可以正常执行。这采用的是多进程的编程模型，同一时间段上可以同时运行多个程序。

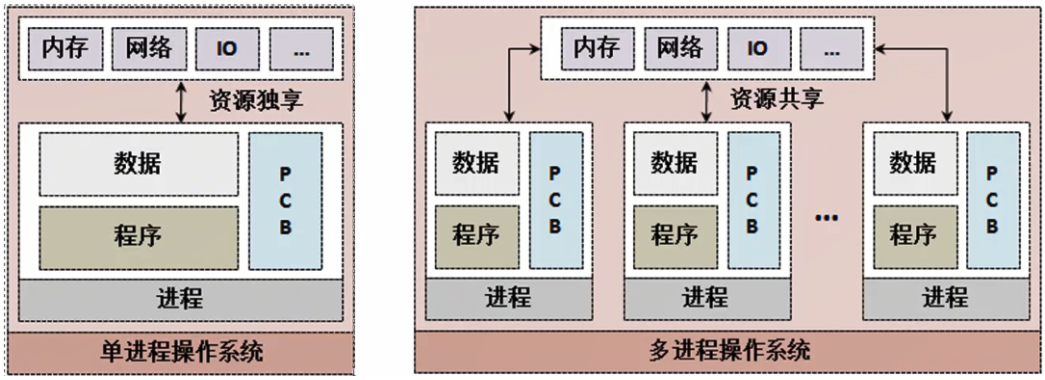

图 1.3: 并发编程

只要打开 Windows 的任务管理器，就可以直接清楚发现所有正在执行的并行进程。

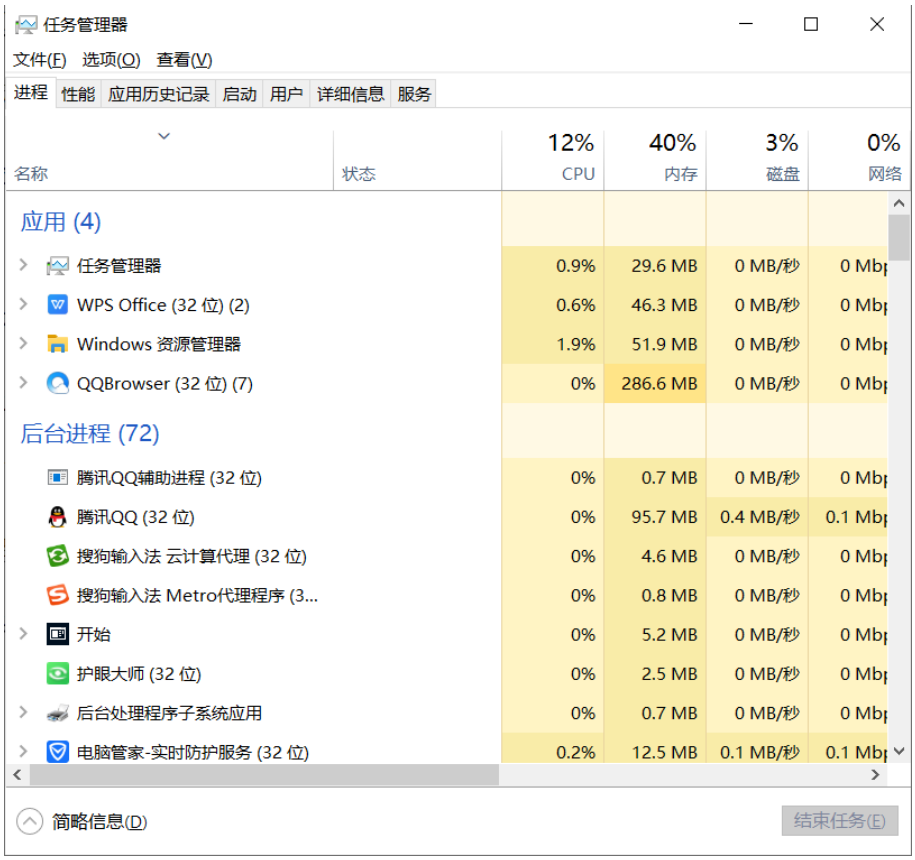

图 1.4: 任务管理器

在早期的硬件系统之中由于没有多核 CPU 的设计，利用时间片的轮转算法（Round Robin），保证在同一个时间段可以同时执行多个进行，但是在某一个时间点上只允许执行一个进程，可以实现资源的切换。

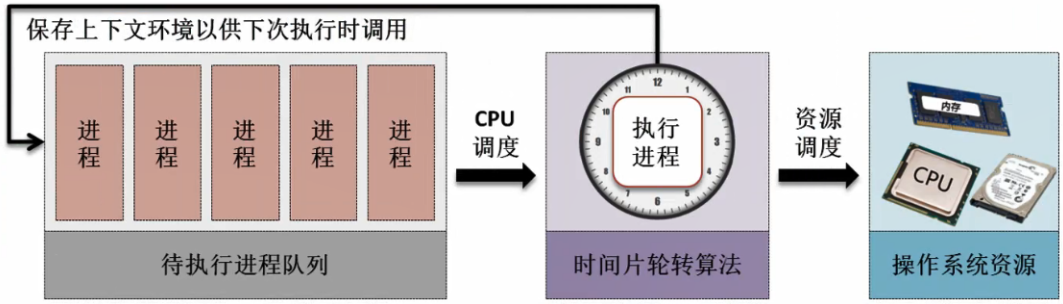

图 1.5: 时间片轮转算法

服务器的硬件性能是有限的，但是对于大部分的程序来讲都属于过剩的状态。于是如果按照传统的单进程模式来运行程序，所有的硬件资源几乎都会被浪费。

1.1.4 进程状态模型

在两状态进程模型中，进程被分为运行态（running）和非运行态（not-running）。

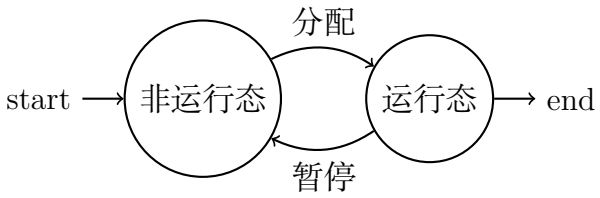

图 1.6: 两态模型

并非所有进程只要是非运行态就一定处于就绪状态，有的需要阻塞等待 I/O 完成。因此非运行态又可分为就绪态（ready）和阻塞态（block）。

所有的进程从其创建到销毁都有各自的生命周期，进程要经过如下几个阶段：

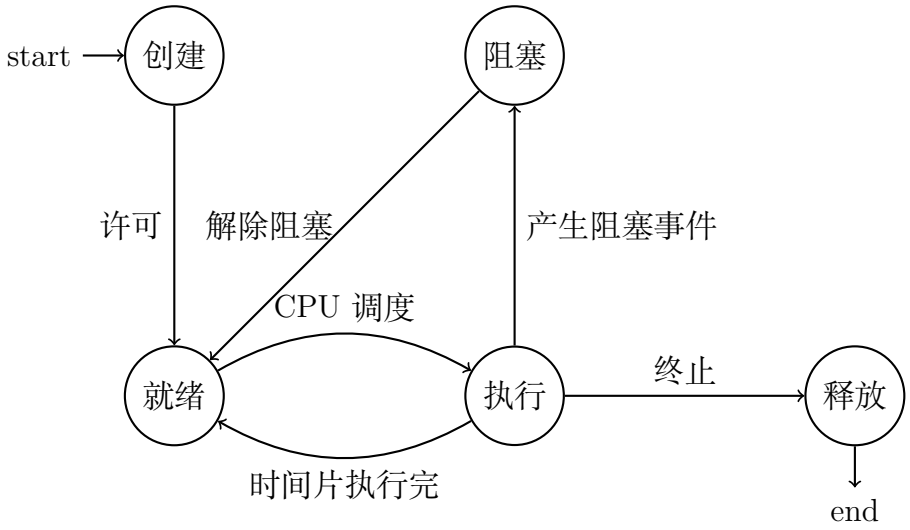

图 1.7: 五态模型

1. 创建状态：系统已经为其分配了 PCB（可以获取进程的而信息），但是所需要执行的进程的上下文环境（context）还未分配，所以这个时候的进程还无法被调度。

2. 就绪状态：该进程已经分配到除 CPU 之外的全部资源，并等待 CPU 调度。

3. 执行状态：进程已获得 CPU 资源，开始正常提供服务。

4. 阻塞状态：所有的进程不可能一直抢占 CPU，依据资源调度的算法，每一个进程运行一段时间之后，都需要交出当前的 CPU 资源，给其它进程执行。

5. 终止状态：某一个进程达到了自然终止的状态，或者进行了强制性的停止，那么进程将进入到终止状态，进程将不再被执行。

1.2 Process 类

1.2.1 Process 类

Python 中在进行多进程开发的时候可以使用 multiprocessing 模块进行多进程的编写，这个模块内容提供有一个 Process 类，利用这个类可以进行多进程的定义。所有的 Python 程序执行都是通过主进程开始的，所有通过 Process 定义的进程都属于子进程。

名称	功能
pid	获取进程 ID
name	获取进程名称
___init___()	创建进程，参数 target 表示进程处理对象；name 表示进程名称
start(self)	进程启动，进入进程调度队列
run(self)	进程处理（不指定 target 时起效）

表 1.1: Process 类

所有的 Python 程序执行都是通过主进程开始的，所有通过 Process 定义的进程都属于子进程。

创建多进程

```
import multiprocessing

def worker():
    """
        进程处理函数
    """
    print("【进程】id: %d, 名称: %s" % (
        multiprocessing.current_process().pid,
        multiprocessing.current_process().name)
    )

def main():
```

```
13      print("【主进程】id: %d, 名称: %s" % (
14          multiprocessing.current_process().pid,
15          multiprocessing.current_process().name)
16      )
17
18      # 创建3个进程
19      for i in range(3):
20          process = multiprocessing.Process(
21              target=worker, name="进程%d" % i
22          )
23          process.start()
24
25  if __name__ == "__main__":
26      main()
```

运行结果

```
【主进程】id: 4476, 名称: MainProcess
【进程】id: 14216, 名称: 进程0
【进程】id: 1424, 名称: 进程1
【进程】id: 16636, 名称: 进程2
```

1.2.2 进程控制

在多进程编程中，所有的进程都会按照既定的代码顺序执行，但是某些进程有可能需要强制执行，或者由于某些问题需要被中断，那么就可以利用 Process 类中提供的方法进行控制。

方法	功能
terminate(self)	关闭进程
is_alive(self)	判断进程是否存活
join(self, timeout)	进程强制执行

表 1.2: 进程控制

所有进程启动后，多个进程进入进程阻塞队列之中依次进行执行，那么这个时候某一个进程是不可能强占 CPU 的，但是通过 join() 可以强制执行进程。如果子进程占用的时间较长，其它的进程也需要进行等待，当子进程全部执行完毕之后就会继续执行主进程的操作。

所有的进程还可以进程中断处理，一般都会中断存活的进程，所以中断前需要对进程状态进行判断。但是从实际开发来讲，很少会出现强制性的霸占或者中断，否则有可能造成数据的丢失。

1.3 psutil 模块

1.3.1 psutil 模块

psutil 是一个进程管理的第三方模块，该模块可以跨平台（Linux、UNIX、MaxOS、Windows 都支持）地进行进程管理，可以极大地简化不同系统中的进程处理操作。

获取全部进程信息

```
import psutil

def main():
    # 获取全部进程
    for process in psutil.process_iter():
        print("【进程】id：%d，名称：%s，创建时间：%s" % (
            process.pid, process.name,
            process.create_time())
        )

if __name__ == "__main__":
    main()
```

除了与进程有关的操作之外，psutil 模块也提供了系统硬件的内容获取。

获取系统硬件信息

```
import psutil

def main():
    # CPU信息
    print("【CPU】物理数量：%d" % psutil.cpu_count(logical=False))
    print("【CPU】逻辑数量：%d" % psutil.cpu_count(logical=True))
    print("【CPU】用户用时：%f" % psutil.cpu_times().user)
    print("【CPU】系统用时：%f" % psutil.cpu_times().system)
    print("【CPU】空闲时间：%f" % psutil.cpu_times().idle)
```

```
10
11      # 磁盘信息
12      print("【磁盘】全部磁盘信息：%s" % psutil.disk_partitions())
13      print("【磁盘】D盘使用率：%s" % str(psutil.disk_usage("D:")))
14      print("【磁盘】IO使用率：%s" % str(psutil.disk_io_counters()))
15
16      # 网络信息
17      print("【网络】数据交换信息：%s" % str(psutil.net_io_counters()))
18      print("【网络】接口信息：%s" % str(psutil.net_if_addrs()))
19      print("【网络】接口状态：%s" % str(psutil.net_if_stats()))
20
21  if __name__ == "__main__":
22      main()
```

1.4 Pipe 进程管道

1.4.1 Pipe 进程通讯管道

进程是程序运行的基本单位，每一个程序内部都有属于自己的存储数据和程序单元，每一个进程都是完全独立的，彼此之前不能直接进行访问。但是可以通过一个特定的管道实现 I/O。

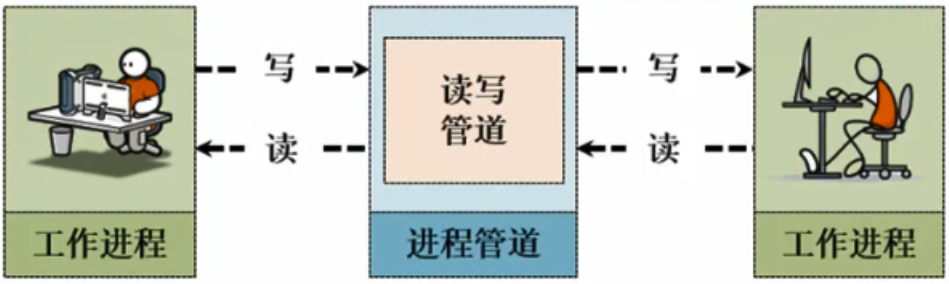

图 1.8: 进程间通信

创建进程通讯管道

```python
import multiprocessing

def send_data(pipe, data):
    """
        往管道发送数据
        Args:
            pipe (Pipe): 管道
            data (str): 发送的数据
    """
    pipe.send(data)
    print("【进程%d】发送数据: %s" % (
        multiprocessing.current_process().pid,
        data
    ))

def recv_data(pipe):
    """
        从管道接收数据
```

```
19          Args:
20              pipe (Pipe): 管道
21      """
22      print("【进程%d】接收数据：%s" % (
23          multiprocessing.current_process().pid,
24          pipe.recv()
25      ))
26
27  def main():
28      # 管道分为发送端和接收端
29      send_end, recv_end = multiprocessing.Pipe()
30      # 创建两个子进程，将管道传递到对应的处理函数
31      sender = multiprocessing.Process(
32                  target=send_data,
33                  args=(send_end, "Hello!")
34              )
35      receiver = multiprocessing.Process(
36                  target=recv_data,
37                  args=(recv_end,)
38              )
39      sender.start()
40      receiver.start()
41
42  if __name__ == "__main__":
43      main()
```

运行结果

【进程11664】发送数据：Hello!

【进程1032】接收数据：Hello!

1.5 进程队列

1.5.1 进程队列

不同的进程彼此之间可以利用管道实现数据的发送和接收，但是如果发送的数据过多并且接收处理缓慢的时候，这种情况下就需要引入队列的形式来进行缓冲的操作。

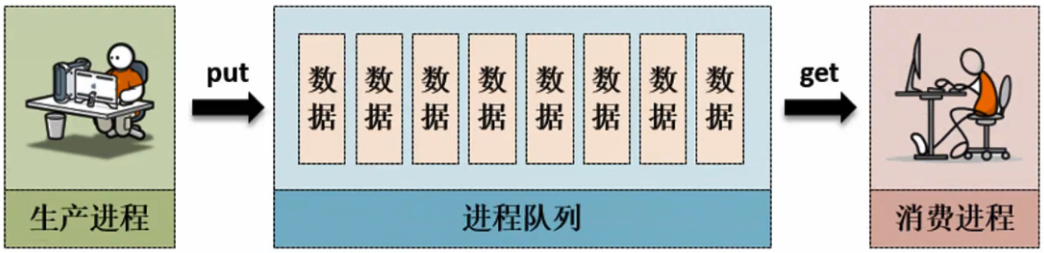

图 1.9: 进程队列

multiprocessing.Queue 是 Python 多进程编程中提供的进程队列结构，该队列采用 FIFO 的形式实现不同进程间的数据通讯，这样可以保证多个数据可以按序实现发送与接收处理。

方法	功能
___init___()	开辟队列，并设置队列保存的最大长度
put()	插入数据到队列，参数 timeout 为阻塞超时（单位：秒）
get()	从队列获取数据，参数 timeout 为阻塞超时（单位：秒）
qsize()	获取队列保存数据个数
empty()	是否为空队列
full()	是否为满队列

表 1.3: multiprocessing.Queue 类

进程队列

```
1 import multiprocessing
2 import time
3
```

```
4  def produce(queue):
5      """
6          生产数据
7          Args:
8              queue (Queue): 进程队列
9      """
10     # 生产3条数据
11     for item in range(3):
12         time.sleep(2)
13         data = "data-%d" % item
14         print("【%s】生产数据: %s" % (
15             multiprocessing.current_process().name,
16             data
17         ))
18         queue.put(data)
19
20 def consume(queue):
21     """
22         消费数据
23         Args:
24             queue (Queue): 进程队列
25     """
26     while True:     # 持续消费
27         print("【%s】消费数据: %s" % (
28             multiprocessing.current_process().name,
29             queue.get()
30         ))
31
32 def main():
33     queue = multiprocessing.Queue()
34     producer = multiprocessing.Process(
35                 target=produce, name="Producer",
36                 args=(queue,)
37             )
38     consumer = multiprocessing.Process(
39                 target=consume, name="Consumer",
40                 args=(queue,)
```

```
            )
    producer.start()
    consumer.start()

if __name__ == "__main__":
    main()
```

运行结果

```
【Producer】生产数据: data-0
【Consumer】消费数据: data-0
【Producer】生产数据: data-1
【Consumer】消费数据: data-1
【Producer】生产数据: data-2
【Consumer】消费数据: data-2
```

1.6 进程通讯

1.6.1 互斥与同步

计算机运行过程中，大量的进程在使用有限、独占、不可抢占的资源，由于进程无限，资源有限，这种矛盾称为竞争（race）。

竞争条件分为两类：

1. 互斥（mutex）：两个或多个进程彼此之间没有内在的制约关系，但是由于要抢占使用某个临界资源（不能被多个进程同时使用的资源，如打印机）而产生制约关系。
2. 同步（synchronization）：两个或多个进程彼此之间存在内在的制约关系（前一个进程执行完，其他的进程才能执行）。

在整个操作系统之中每一个进程都有自己独立的数据存储单元，也就是说不同进程之间无法直接实现数据共享。通过管道流可以实现进程之间的数据共享，相当于打通了不同进程之间的限制。但是不同的进程操作同一个资源就必须考虑数据同步的问题。

要理解同步概念，首先要清楚进程不同步所带来的问题。

售票操作（Bug 版本）

```
import multiprocessing
import time

def sell_ticket(dict):
    while True:     # 持续售票
        # 获取当前剩余票数
        num = dict.get("ticket")

        if num > 0:         # 如果还有票剩余
            time.sleep(1)   # 模拟网络延迟
```

```
            num -= 1        # 票数减1
            print("【售票员%d】售票成功，剩余票数：%d" % (
                multiprocessing.current_process().pid,
                num
            ))
            dict.update({"ticket":num})     # 更新票数
        else:                               # 已经没有票了
            break

def main():
    # 创建共享数据对象
    manager = multiprocessing.Manager()
    # 创建一个可以被多个进程共享的字典对象
    ticket_dict = manager.dict(ticket=5)   # 默认有5张票

    # 创建多个售票进程
    sellers = [
        multiprocessing.Process(
            target=sell_ticket, args=(ticket_dict,)
        )
        for _ in range(5)
    ]

    for seller in sellers:
        seller.start()
    for seller in sellers:
        seller.join()   # 进程强制执行

if __name__ == "__main__":
    main()
```

运行结果

```
【售票员9732】售票成功，剩余票数：4
【售票员1640】售票成功，剩余票数：4
【售票员5976】售票成功，剩余票数：4
【售票员8048】售票成功，剩余票数：4
【售票员10516】售票成功，剩余票数：4
【售票员9732】售票成功，剩余票数：3
【售票员1640】售票成功，剩余票数：3
【售票员5976】售票成功，剩余票数：3
【售票员8048】售票成功，剩余票数：3
【售票员10516】售票成功，剩余票数：3
【售票员9732】售票成功，剩余票数：2
【售票员1640】售票成功，剩余票数：2
【售票员5976】售票成功，剩余票数：2
【售票员8048】售票成功，剩余票数：2
【售票员10516】售票成功，剩余票数：2
【售票员9732】售票成功，剩余票数：1
【售票员1640】售票成功，剩余票数：1
【售票员5976】售票成功，剩余票数：1
【售票员8048】售票成功，剩余票数：1
【售票员10516】售票成功，剩余票数：1
【售票员1640】售票成功，剩余票数：0
【售票员9732】售票成功，剩余票数：0
【售票员5976】售票成功，剩余票数：0
【售票员8048】售票成功，剩余票数：0
【售票员10516】售票成功，剩余票数：0
```

多个进程同时进行票数判断的时候，在没有及时修改票数的情况下，就会出现数据不同步的问题。这套操作由于没有对同步的限制，所以就造成了不同步的问题。

1.6.2 Lock

并发进程的执行如果要进行同步处理，那么就必须对一些核心代码进行同步。Python 中提供了一个 Lock 同步锁机制，利用这种锁机制可以实现部分代码的同步锁定，保证每一次只允许有一个进程执行这部分的代码。

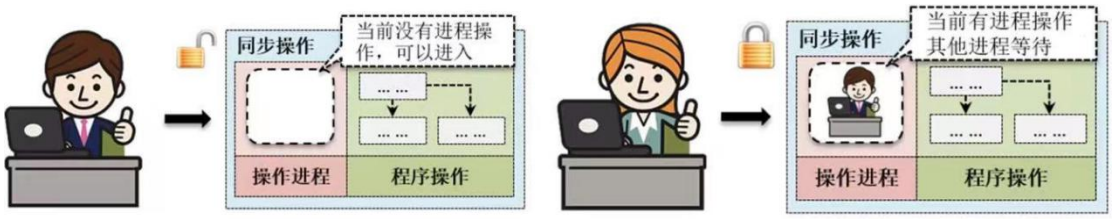

图 1.10: 互斥锁

方法	功能
acquire()	获取锁，如果当前没有可用锁资源，则进行等待
release()	操作完毕，释放锁资源

表 1.4: Lock 类

售票操作（正确版本）

```
1  import multiprocessing
2  import time
3
4  def sell_ticket(lock, dict):
5      while True:      # 持续售票
6          # 请求锁定，如果5秒没有锁定则放弃
7          lock.acquire(timeout=5)
8
9          # 获取当前剩余票数
10         num = dict.get("ticket")
11
12         if num > 0:          # 如果还有票剩余
13             time.sleep(1)    # 模拟网络延迟
14             num -= 1         # 票数减1
15             print("【售票员%d】售票成功，剩余票数：%d" % (
```

```
                multiprocessing.current_process().pid,
                num
            ))
            dict.update({"ticket":num})     # 更新票数
        else:                   # 已经没有票了
            break

        lock.release()      # 释放锁

def main():
    lock = multiprocessing.Lock()   # 同步锁
    # 创建共享数据对象
    manager = multiprocessing.Manager()
    # 创建一个可以被多个进程共享的字典对象
    ticket_dict = manager.dict(ticket=5)   # 默认有5张票

    # 创建多个售票进程
    sellers = [
        multiprocessing.Process(
            target=sell_ticket, args=(lock, ticket_dict)
        )
        for _ in range(5)
    ]

    for seller in sellers:
        seller.start()
    for seller in sellers:
        seller.join()   # 进程强制执行

if __name__ == "__main__":
    main()
```

运行结果

```
【售票员14612】售票成功，剩余票数：4
【售票员15868】售票成功，剩余票数：3
【售票员13972】售票成功，剩余票数：2
【售票员10844】售票成功，剩余票数：1
【售票员2872】售票成功，剩余票数：0
```

一旦程序中追加了同步锁，那么程序的部分代码就只能以单进程执行了，这样势必会造成程序的执行性能下降，只有在考虑数据操作安全的情况下才会使用锁机制。

1.6.3 Semaphore

Semaphore（信号量）是一种有限资源的进程同步管理机制。例如银行的业务办理，所有客户都会拿到一个号码，而后号码会被业务窗口叫号，被叫号的办理者就可以办理业务。

Semaphore 类本质上是一种带有计数功能的进程同步机制，acquire() 减少计数，release() 增加计数。当可用信号量的计数为 0 时，后续进程将被阻塞。

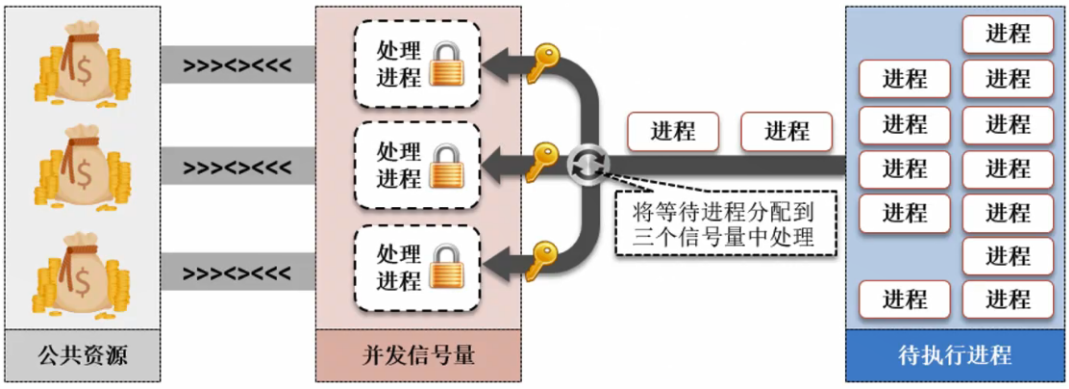

图 1.11: Semaphore

Lock 一般是针对于一个资源同步的，而 Semaphore 是针对有限资源的并行访问。

信号量同步处理

```python
import multiprocessing
import time

def work(sema):
    if sema.acquire():      # 获取信号量
        print("【进程%d】开始办理业务" %
            multiprocessing.current_process().pid)
        time.sleep(2)       # 模拟办理业务
        print("【进程%d】结束办理业务" %
            multiprocessing.current_process().pid)
        sema.release()      # 释放资源

def main():
    # 允许3个进程并发执行
    sema = multiprocessing.Semaphore(3)
    workers = [
        multiprocessing.Process(target=work, args=(sema,))
        for _ in range(10)
    ]

    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()

if __name__ == "__main__":
    main()
```

运行结果

【进程7880】开始办理业务
【进程3032】开始办理业务
【进程5412】开始办理业务
【进程7880】结束办理业务
【进程2876】开始办理业务
【进程3032】结束办理业务
【进程14076】开始办理业务
【进程5412】结束办理业务
【进程8816】开始办理业务
【进程2876】结束办理业务
【进程14076】结束办理业务
【进程7900】开始办理业务
【进程7860】开始办理业务
【进程8816】结束办理业务
【进程16252】开始办理业务
【进程7860】结束办理业务
【进程7900】结束办理业务
【进程972】开始办理业务
【进程16252】结束办理业务
【进程972】结束办理业务

Chapter 2 网络编程