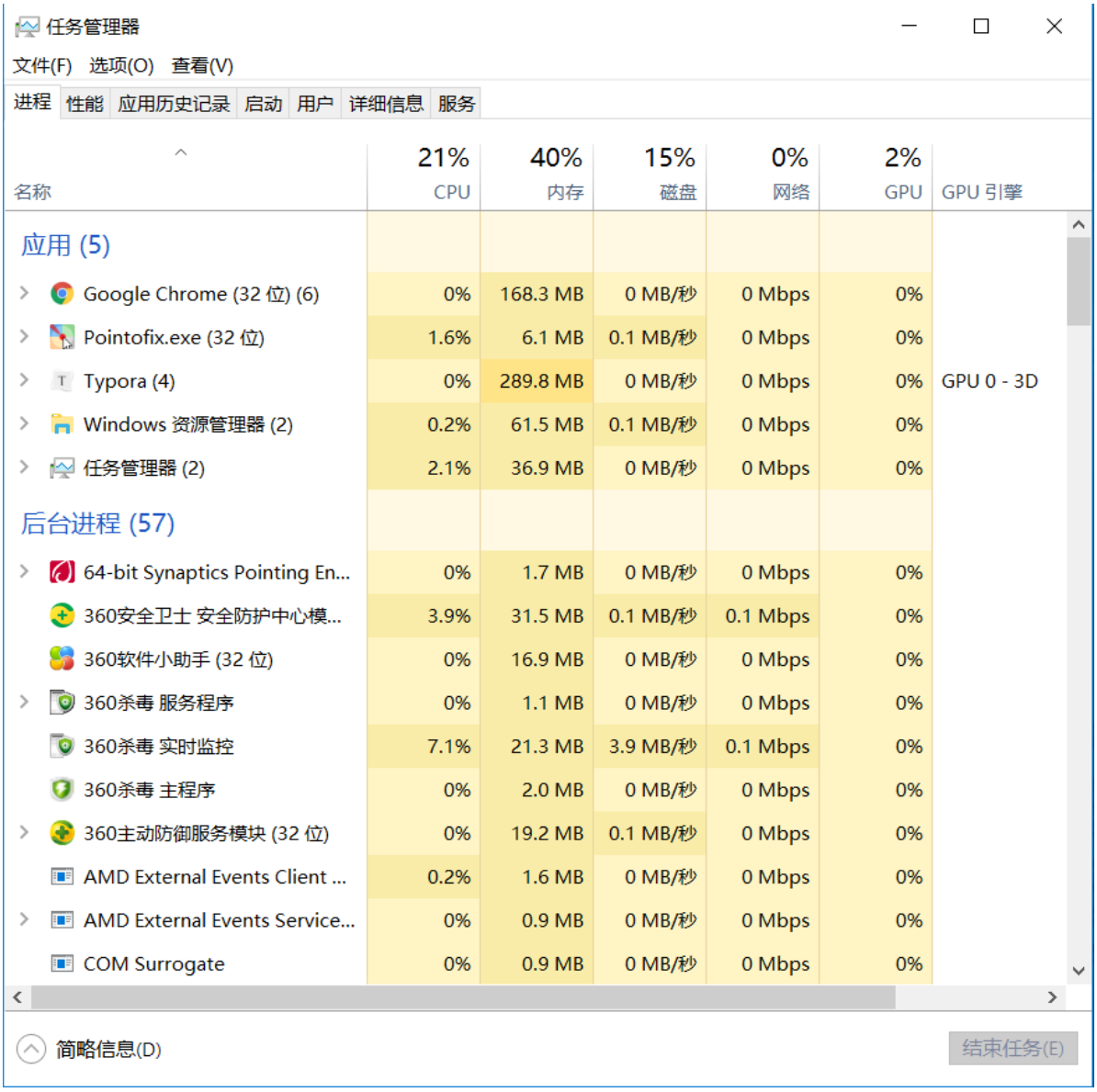
3. 多任务编程

3.1 多任务概述

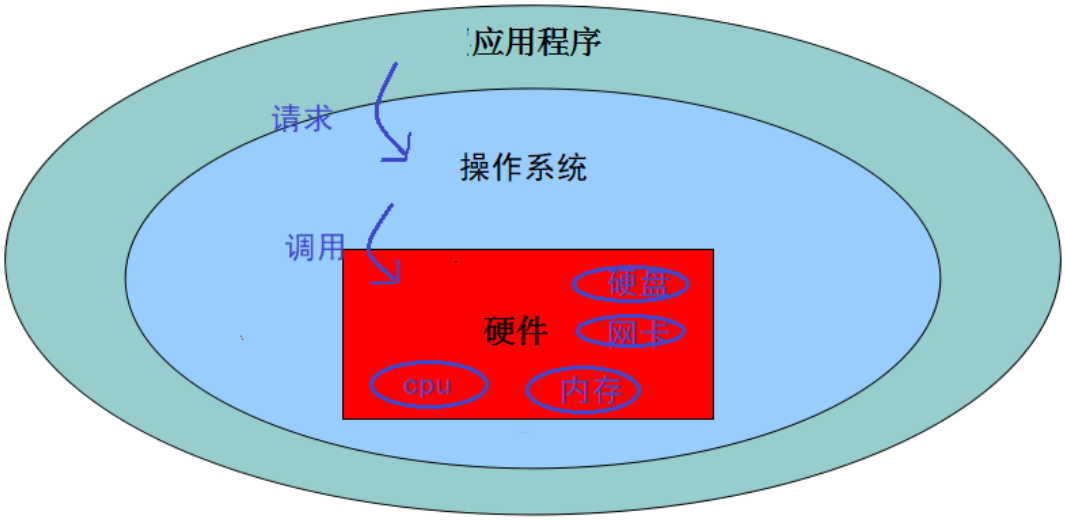
- 多任务

  即操作系统中可以同时运行多个任务。比如我们可以同时挂着qq、听音乐，并浏览网页。这是我们看得到的任务，在系统中还有很多系统任务在执行，现在的操作系统基本都是多任务的，具备运行多任务的能力。

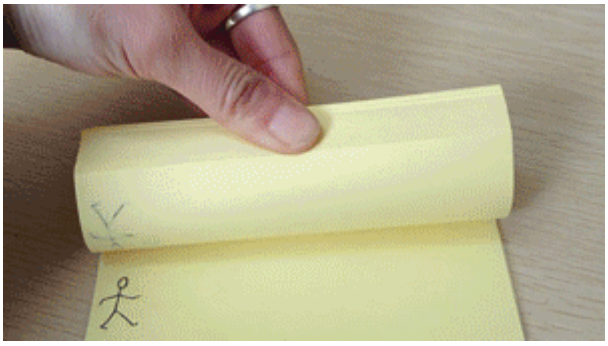


- 计算机原理
  - CPU：计算机硬件的核心部件，用于对任务执行运算

- 操作系统调用CPU执行任务



- CPU轮询机制：CPU在多个任务之间快速地切换执行，切换速度在微秒级别，其实CPU在同一时间点只执行一个任务，但是因为切换太快了，从应用层看好像有多个任务同时在执行。



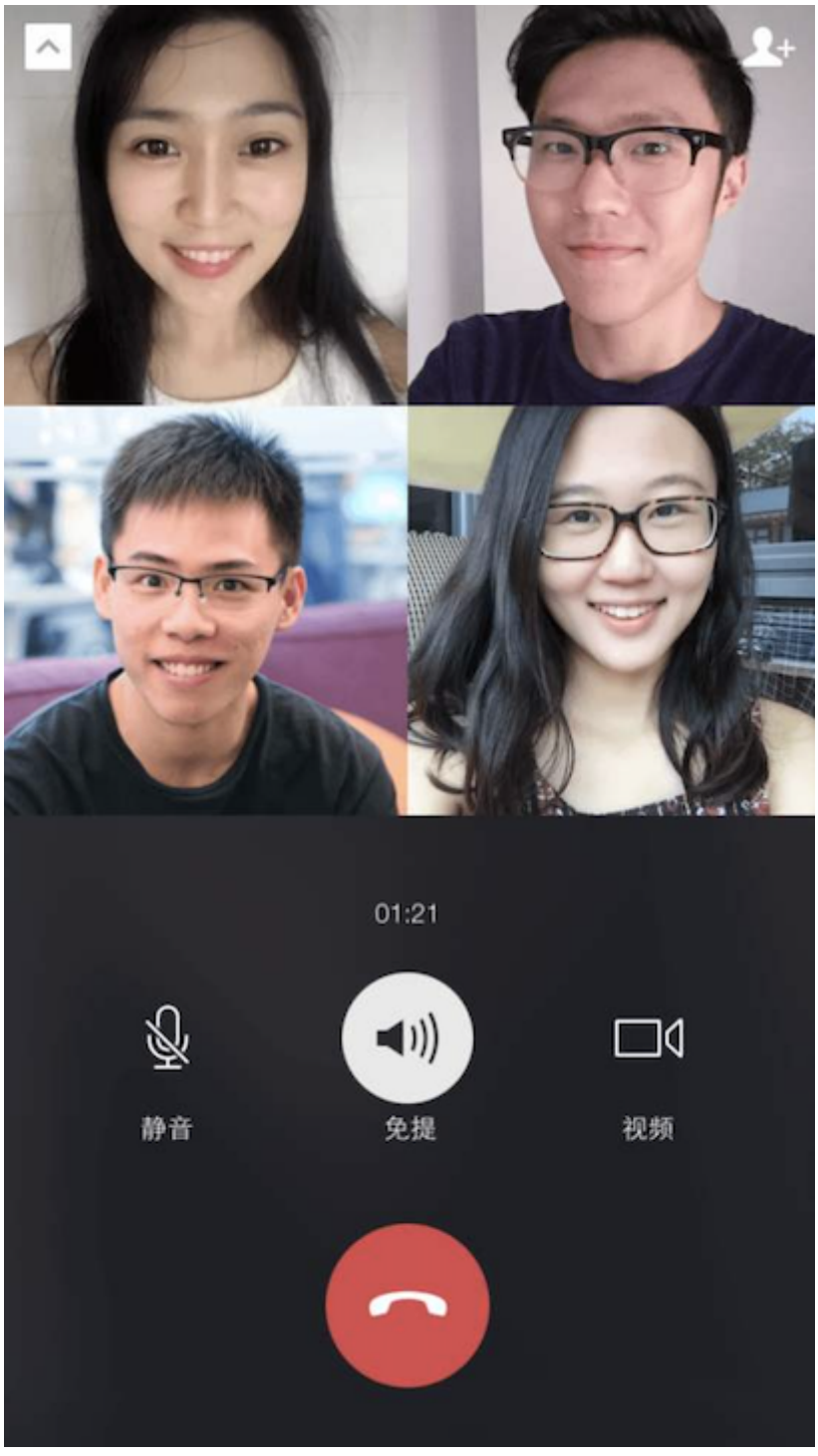
- 多核CPU：现在的计算机一般都是多核CPU，比如四核、八核，可以理解为多个单核CPU的集合。这时在执行任务时就有了选择，可以将多个任务分配给某一个CPU核心，也可以将多个任务分配给多个CPU核心，操作系统会自动根据任务的复杂程度选择最优的分配方案。
  - 并发： 多个任务如果被分配给了一个CPU内核，那么这多个任务之间就是并发关系，并发关系的多个任务之间并不是真正的"同时"。
  - 并行： 多个任务如果被分配给了不同的CPU内核，那么这多个任务之间执行时就是并行关系，并行关系的多个任务是真正的

“同时”执行。

- 什么是多任务编程

  多任务编程即一个程序中编写多个任务，在程序运行时让这多个任务一起运行，而不是一个一个的顺序执行。

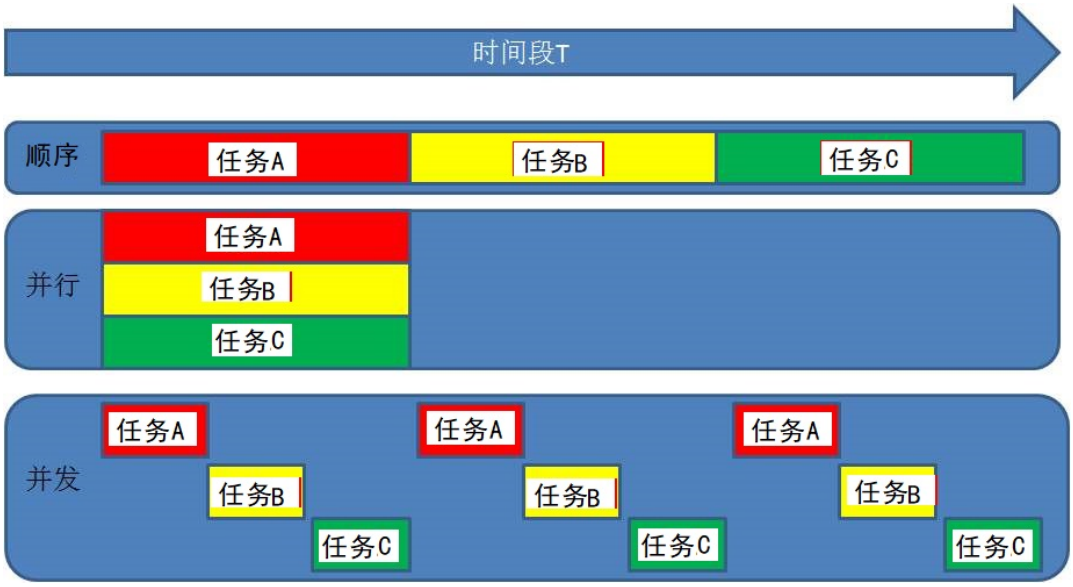
  比如微信视频聊天，在微信运行过程中既用到视频任务也用到音频任务，甚至同时还能发消息。这就是典型的多任务，而实际的开发过程中这样的情况比比皆是。



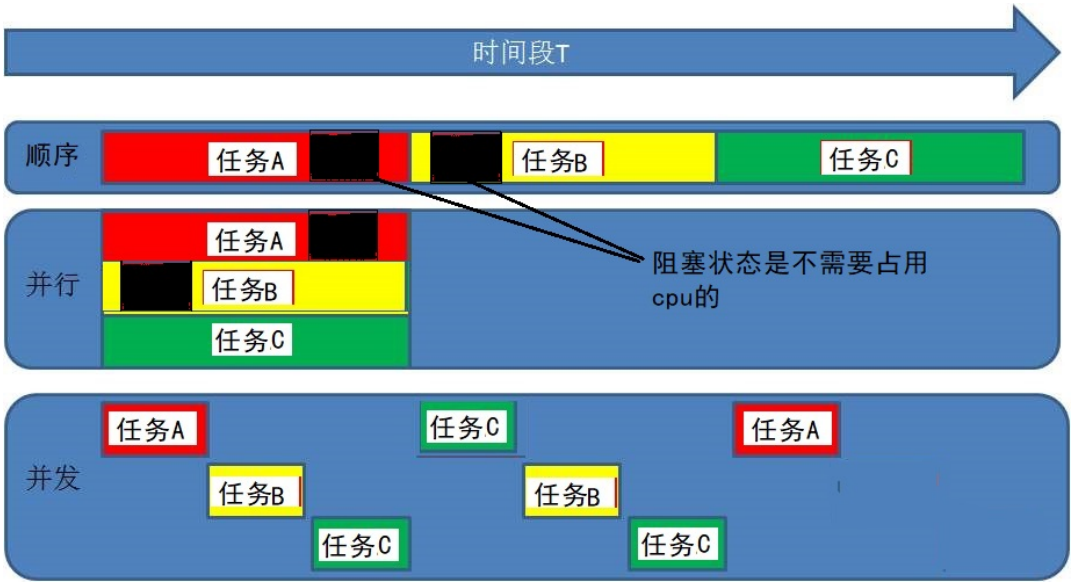
- 实现多任务编程的方法： **多进程编程，多线程编程**

- 多任务意义
  - 提高了任务之间的配合，可以根据运行情况进行任务创建。

    比如： 你也不知道用户在微信使用中是否会进行视频聊天，总不能提前启动起来吧，这是需要根据用户的行为启动新任务。
  - 充分利用计算机资源，提高了任务的执行效率。
    - 在任务中无阻塞时只有并行状态才能提高效率



    - 在任务中有阻塞时并行并发都能提高效率



3.2 进程（Process）

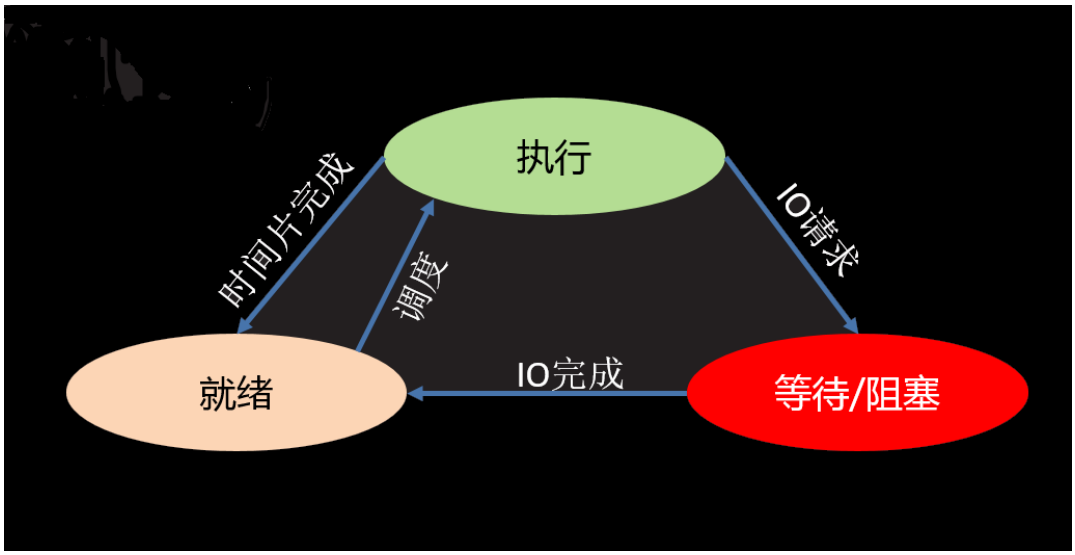
3.2.1 进程概述

- 定义： 程序在计算机中的一次执行过程
  - 程序是一个可执行的文件，是静态的占有磁盘
  - 进程是一个动态的过程描述，占有计算机运行资源，有一定的生命周期

- 进程状态
  - 三态

    就绪态： 进程具备执行条件，等待系统调度分配CPU资源

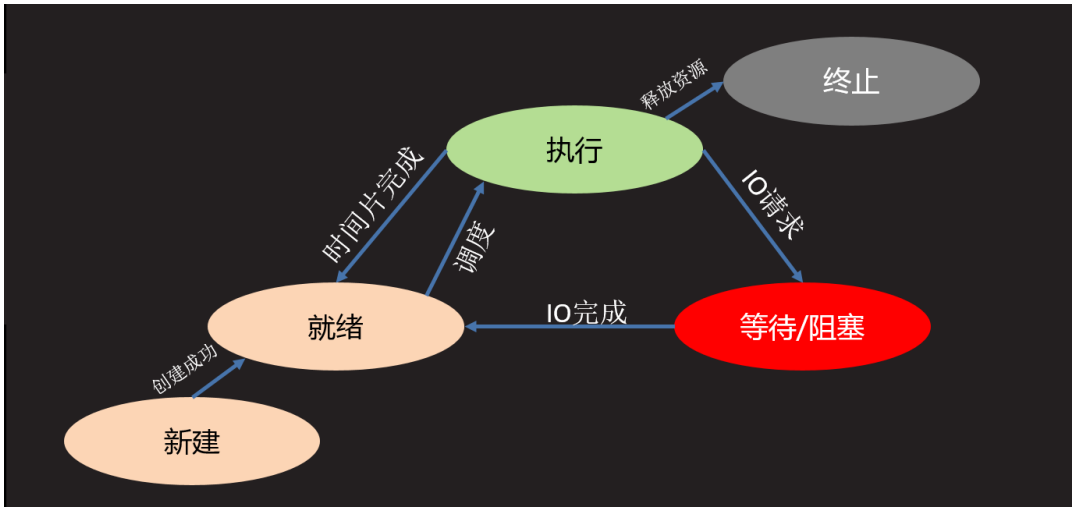
    ```
    运行态 ： 进程占有CPU正在运行

    等待态 ： 进程阻塞等待，此时会让出CPU
    ```



  - 五态 (在三态基础上增加新建和终止)

    新建： 创建一个进程，获取资源的过程

```
终止 ： 进程结束，释放资源的过程
```



- 进程命令
  - 查看进程信息 【windows：netstat -ano】

```
ps -aux
kill  -9   pid    可以用于杀死一个进程
```

```
    Robin     :~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.7  0.3 159896  9272 ?        Ss   18:36   0:02 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    18:36   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   18:36   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   18:36   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I    18:36   0:00 [kworker/0:0-eve]
root           6  0.0  0.0      0     0 ?        I<   18:36   0:00 [kworker/0:0H-kb]
root           7  0.0  0.0      0     0 ?        I    18:36   0:00 [kworker/0:1-cgr]
```

    - USER：进程的创建者
    - PID ：操作系统分配给进程的编号，大于0的整数，系统中每个进程的PID都不重复，PID是区分进程的重要标志。
    - %CPU、%MEM：占有的CPU和内存百分百
    - STAT：进程状态信息，R：运行状态、S：睡眠状态、T：暂停状态、Z：僵尸状态、s：包含子进程、l：多线程、+：前台显示
    - START：进程启动时间
    - COMMAND：通过什么程序启动的进程

  - 进程树形结构

```
pstree
```

- 父子进程：在Linux操作系统中，进程形成树形关系，任务上一级进程是下一级的父进程，下一级进程是上一级的子进程。

3.2.2 多进程编程

- 使用模块： multiprocessing
- 创建流程

  【1】 将需要新进程执行的事件封装为函数

  【2】 通过模块的Process类创建进程对象，关联函数

  【3】 通过进程对象调用start启动进程

- 主要类和函数使用

```
Process()
功能 ： 创建进程对象
参数 ： target 绑定要执行的目标函数
      args 元组，用于给target函数位置传参
      kwargs 字典，用于给target函数键值传参
      daemon  bool值，让子进程随父进程退出
```

```
p.start()
功能 ： 启动进程
```

> 注意 : 启动进程时target绑定函数开始执行，该函数作为新进程的执行内容，此时进程真正被创建

```
p.join([timeout])
功能：阻塞等待子进程退出
参数：最长等待时间
```

进程创建示例：

```
"""
进程创建示例 01
"""
from multiprocessing import Process
from time import sleep
```

```
a = 1
# 1. 创建新进程要执行的函数
def my_func():
    print('新进程开始执行')
    for i in range(101):
        sleep(0.1)
        print(f'新进程，i:{i}')
    global a
    print(f'在新进程中，a初始值：{a}')
    a = 100
    print(f'在新进程中，a：{a}')
    print('新进程执行结束')


if __name__ == '__main__':
    # 2. 创建进程对象，并与函数相绑定
    p = Process(target=my_func)
    # 3. 启动新进程
    p.start()

    print('主进程也做些事情')
    for i in range(101):
        sleep(0.1)
        print(f'主进程，i:{i}')

    sleep(2)
    print(f'在主进程中，a：{a}')
    print('主进程执行结束')
```

```
"""
进程创建示例02 ：  含有参数的进程函数
"""
from multiprocessing import Process
from time import sleep


def worker(second,name):
    print('新进程执行')
    sleep(second)
```

```
    print(f'name:{name}')


if __name__ == '__main__':
    # p = Process(target=worker, args=(2, 'Tom'))
    # p = Process(target=worker, args=(2,), kwargs=
{'name':'Tom'})
    # 子进程随父进程结束而结束
    p = Process(target=worker, args=(2,), kwargs={'name':
'Tom'}, daemon=True)
    p.start()
    # sleep(3)

    # 父进程等待子进程结束，再继续执行
    p.join()
```

1. **target 参数**： `target=worker` 指定了要在新进程中运行的目标函数或方法。这里假设 `worker` 是一个函数或方法的名称。
2. **args 参数**： `args=(2,)` 是一个元组，包含传递给 `worker` 函数的位置参数。在这个例子中， `worker` 函数被调用时将接收一个位置参数 `2` 。
3. **kwargs 参数**： `kwargs={'name': 'Tom'}` 是一个字典，包含传递给 `worker` 函数的关键字参数。在这个例子中， `worker` 函数被调用时将接收一个关键字参数 `name` ，其值为 `'Tom'` 。
4. **daemon 参数**： `daemon=True` 表示将新创建的进程标记为守护进程。守护进程会随着主进程的结束而结束，而非守护进程则会在主进程结束后继续运行。

- 进程执行现象理解 （难点）
  - 新的进程是原有进程的子进程，子进程复制父进程全部内存空间代码段，一个进程可以创建多个子进程
  - 子进程只执行指定的函数，其余内容均是父进程执行内容，但是子进程也拥有其他父进程资源
  - 各个进程在执行上互不影响，也没有先后顺序关系
  - 进程创建后，各个进程空间独立，相互没有影响
  - multiprocessing 创建的子进程中无法使用标准输入（即无法使用 input）

3.2.3 进程处理细节

- 进程相关函数

```
os.getpid()
功能： 获取一个进程的PID值
返回值： 返回当前进程的PID
```

```
os.getppid()
功能： 获取父进程的PID号
返回值： 返回父进程PID
```

```
sys.exit(info)
功能：退出进程
参数：字符串表示退出时打印内容
```

```
'''
创建多个子进程
'''
from multiprocessing import Process
from time import sleep
import os, sys
from random import randint

# def func01():
#     print('兔子开始跑步')
#     sleep(1)
#     print(f'PID:{os.getpid()},PPID:{os.getppid()}')
#     print('兔子跑步结束')
#
# def func02():
#     print('乌龟开始跑步')
#     sleep(2)
#     print(f'PID:{os.getpid()},PPID:{os.getppid()}')
#     print('乌龟跑步结束')
#
# def func03():
#     print('老牛开始跑步')
#     sleep(1)
```

```
#     print(f'PID:{os.getpid()},PPID:{os.getppid()}')
#     print('老牛跑步结束')
#
#
# if __name__ == '__main__':
#     print('预备，跑！')
#     jobs = []
#     for func in [func01, func02, func03]:
#         p = Process(target=func)
#         jobs.append(p)
#         p.start()
#
#     for job in jobs:
#         job.join()
#     print('比赛结束！')


def race(second, name):
    if name == '老牛':
        sys.exit('老牛太累了，退出比赛')

    print(f'{name}开始跑步')
    sleep(second)
    print(f'PID:{os.getpid()},PPID:{os.getppid()}')
    print(f'{name}跑步{second}秒结束')


if __name__ == '__main__':
    print('预备，跑！')
    jobs = []
    for arg in [(randint(1,5),'兔子'),(randint(1,5),'乌龟'),
(randint(1,5),'老牛')]:
        p = Process(target=race, args=arg)
        jobs.append(p)
        p.start()

    for job in jobs:
        job.join()
    print('比赛结束！')
```

随堂练习：大文件拆分

有一个大文件，将其拆分成上下两个部分 （按照字节大小），要求两个部分拆分要同步进行
plus ： 假设文件很大不要一次read读取全部

提示 ： os.path.getsize() 获取文件大小
创建两个子进程分别拆上下两个部分

```
import os
from multiprocessing import Process

filename = "./dict.txt"
size = os.path.getsize(filename)

# 如果父进程打开子进程直接用则会共用一个文件偏移量
# fr = open(filename,'rb')

def top():
    fr = open(filename,'rb')
    fw = open('top.txt','wb')
    n = size // 2
    while n >= 1024:
        fw.write(fr.read(1024))
        n -= 1024
    else:
        fw.write(fr.read(n))
    fr.close()
    fw.close()

def bot():
    fr = open(filename, 'rb')
    fw = open('bot.txt', 'wb')
    fr.seek(size//2,0)  # 文件偏移量到中间
    while True:
        data = fr.read(1024)
        if not data:
            break
        fw.write(data)
    fr.close()
    fw.close()

jobs=[]
```

```
for item in [top,bot]:
    p = Process(target=item)
    jobs.append(p)
    p.start()

[i.join() for i in jobs]
print("拆分完成")
```

3.2.4 创建进程类

进程的基本创建方法是将子进程执行的内容封装为函数，如果我们更热衷于面向对象的编程思想，也可以使用类来封装进程内容。

- 创建步骤

  【1】 继承Process类

  【2】 重写 `__init__` 方法添加自己的属性，使用super()加载父类属性

  【3】 重写run()方法

- 使用方法

  【1】 实例化对象

  【2】 调用start自动执行run方法

```
"""
自定义进程类
"""
from multiprocessing import Process
from time import sleep


class MyProcess(Process):
    def __init__(self, value):
        self.value = value
        super().__init__()  # 调用父类的init

    # 重写run作为进程的执行内容
```

```
    def run(self):
        for i in range(self.value):
            sleep(2)
            print("自定义进程类。。。。")

if __name__ == '__main__':
    p = MyProcess(3)
    p.start() # 将run方法作为进程执行
```

随堂练习：
1. 求100000以内质数之和，并且计算这个求和过程的时间
2. 将100000分成10份，创建10个进程，每个进程求其中一份的质数之和，统计10个进程执行完的时间

提示：
质数：只能被1和其本身整除的整数 >1

```
import time
from multiprocessing import Process

# def is_prime(n):
#     if n <= 1:
#         return False
#     for i in range(2,n // 2 + 1):
#         if n % i == 0:
#             return False
#     return True
#
# def prime_sum():
#     prime = [] # 存放所有质数
#     for i in range(100001):
#         if is_prime(i):
#             prime.append(i)  # 存入列表
#     print(sum(prime))
#
#
# begin = time.time()
# prime_sum()
# end = time.time()
# print(f"用时：{end - begin}")
```

```python
# 求begin -- end 之间的质数之和
class Prime(Process):
    @staticmethod
    def is_prime(n):
        if n <= 1:
            return False
        for i in range(2, n // 2 + 1):
            if n % i == 0:
                return False
        return True

    def __init__(self,begin,end):
        self.begin = begin # 起始数字
        self.end = end # 结尾数字
        super().__init__()

    def run(self):
        prime = []  # 存放所有质数
        for i in range(self.begin,self.end):
            if Prime.is_prime(i):
                prime.append(i)  # 存入列表
        print(sum(prime))

if __name__ == '__main__':
    jobs = []
    begin = time.time()
    for i in range(1,100001,10000):
        p = Prime(i,i + 10000)
        jobs.append(p)
        p.start()
    [i.join() for i in jobs]
    end = time.time()
    print(f"用时:{end - begin}")
```

3.2.5 进程间通信

- 必要性： 进程间空间独立，资源不共享，需要在进程间数据传输时就要用特定的手段进行数据通信。
- 常用进程间通信方法：消息队列、套接字等。
- 消息队列使用
  - 通信原理： 在内存中开辟空间，建立队列模型，进程通过将消息存入队列，从队列取出完成通信。
  - 实现方法

```
from multiprocessing import Queue

q = Queue(maxsize=0)
功能： 创建队列对象
参数：最多存放消息个数
返回值：队列对象

q.put(data)
功能：向队列存入消息
参数：data   要存入的内容

q.get()
功能：从队列取出消息
返回值：  返回获取到的内容

q.full()    判断队列是否为满
q.empty()   判断队列是否为空
q.qsize()   获取队列中消息个数
q.close()   关闭队列
```

```
进程间通信示例：
from multiprocessing import Process,Queue

# 创建消息队列
q = Queue(5)

# 子进程函数
def handle():
    while True:
```

```
        cmd = q.get() # 取出指令
        if cmd == "1":
            print("\n完成指令1")
        elif cmd == "2":
            print("\n完成指令2")

# 创建进程
p = Process(target=handle,daemon=True)
p.start()

while True:
    cmd = input("指令: ")
    if not cmd:
        break
    q.put(cmd) # 通过队列给子进程
```

群聊聊天室

> 功能： 类似qq群功能
>
> 【1】有人进入聊天室需要输入姓名，姓名不能重复
>
> 【2】有人进入聊天室时，其他人会收到通知：Lucy **进入了聊天室**
>
> 【3】一个人发消息，其他人会收到： Lucy： **一起出去玩啊**
>
> 【4】有人退出聊天室，则其他人也会收到通知：Lucy **退出了聊天室**
>
> 【5】扩展功能：服务器可以向所有用户发送公告：**管理员消息： 大家好，欢迎进入聊天室**

```
################ 服务端参考代码 ###################
"""
群聊聊天室服务器端
在Linux上运行
"""
```

```python
from socket import *
from multiprocessing import Process

# 服务器地址
HOST = "0.0.0.0"
PORT = 8888
ADDR = (HOST, PORT)

# 存储用户信息 {name:address}
user = {}


# 处理进入聊天室
def login(sock, name, address):
    print("server login")
    if name in user or "管理" in name:
        sock.sendto(b"FAIL", address)
    else:
        sock.sendto(b"OK", address)
        # 告知其他人
        msg = f"欢迎 {name} 进入聊天室"
        for addr in user.values():
            sock.sendto(msg.encode(), addr)
        user[name] = address  # 存储用户
        # print(user)  # 测试


# 处理聊天
def chat(sock, name, content):
    msg = f"{name} : {content}"
    for key, value in user.items():
        # 不是本人就发送
        if key != name:
            sock.sendto(msg.encode(), value)


# 处理退出
def exit(sock, name):
    if name in user:
        del user[name]  # 删除该用户
    # 通知其他用户
```

```
    msg = f"{name} 退出聊天室"
    for addr in user.values():
        sock.sendto(msg.encode(), addr)


def handle(sock):
    # 不断接收请求，分情况讨论
    while True:
        request, addr = sock.recvfrom(1024)
        tmp = request.decode().split(" ", 2)
        # 分情况讨论
        if tmp[0] == "LOGIN":
            # tmp ->[LOGIN,name]
            login(sock, tmp[1], addr)
        elif tmp[0] == "CHAT":
            # tmp ->[CHAT,name,content]
            chat(sock, tmp[1], tmp[2])
        elif tmp[0] == "EXIT":
            # tmp ->[EXIT,name]
            exit(sock, tmp[1])


# 程序入口函数
def main():
    # 创建udp
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(ADDR)

    # 接收请求，分类处理
    p = Process(target=handle, args=(sock,), daemon=True)
    p.start()

    while True:
        content = input("管理员消息:")
        if not content:
            break
        msg = "CHAT 管理员消息 " + content
        # 从父进程发送到子进程
        sock.sendto(msg.encode(), ADDR)
```

```python
if __name__ == '__main__':
    main()


################## 客户端参考代码 ##################
"""
群聊聊天室客户端
在Linux上运行
"""
from socket import *
from multiprocessing import Process
import sys

# 服务器地址
SERVER_ADDR = ("127.0.0.1", 8888)


def login(sock):
    while True:
        name = input("请输入昵称:")
        # 组织请求
        msg = "LOGIN " + name
        sock.sendto(msg.encode(), SERVER_ADDR)
        result, addr = sock.recvfrom(1024)
        if result == b"OK":
            print("进入聊天室")
            return name
        else:
            print("该昵称已存在")


# 子进程接收函数
def recv_msg(sock):
    while True:
        data, addr = sock.recvfrom(1024 * 10)
        # 格式处理
        content = f"\n{data.decode()}\n发言: "
        print(content, end="")


#   父进程发送函数
```

```
def send_msg(sock, name):
    while True:
        try:
            content = input("发言: ")
        except KeyboardInterrupt:
            content = "exit"
        # 表示退出
        if content == 'exit':
            msg = "EXIT " + name
            sock.sendto(msg.encode(), SERVER_ADDR)
            sys.exit("您已退出聊天室")
        msg = f"CHAT {name} {content}"
        sock.sendto(msg.encode(), SERVER_ADDR)


def main():
    sock = socket(AF_INET, SOCK_DGRAM)
    name = login(sock)  # 请求进入聊天室
    # 子进程负责接收
    p = Process(target=recv_msg, args=(sock,), daemon=True)
    p.start()
    send_msg(sock, name)  # 发送消息


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

3.3 线程 (Thread)

3.3.1 线程概述

- 什么是线程

  【1】 线程被称为轻量级的进程，也是多任务编程方式

  【2】 也可以利用计算机的多CPU资源

  【3】 线程可以理解为进程中再开辟的分支任务

- 线程特征

【1】 一个进程中可以包含多个线程

【2】 线程也是一个运行行为，消耗计算机资源

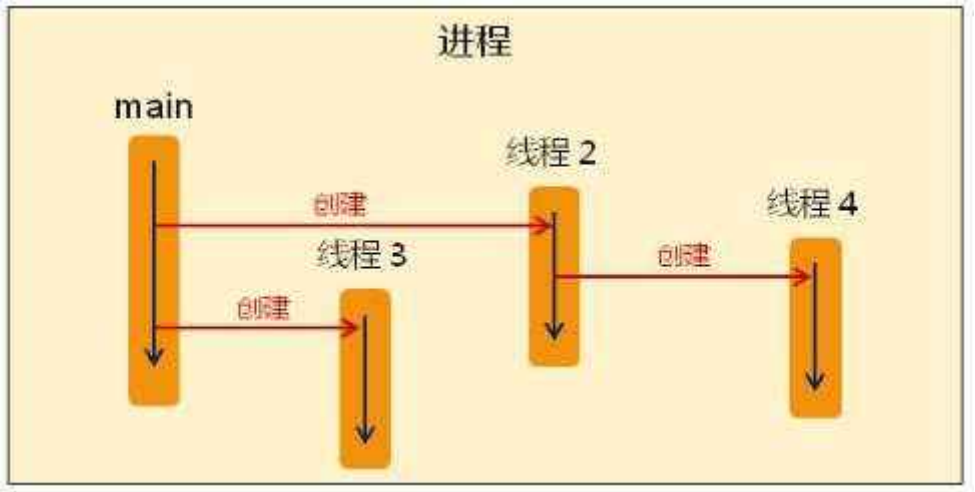
【3】 一个进程中的所有线程共享这个进程的资源

【4】 多个线程之间的运行同样互不影响各自运行

【5】 线程的创建和销毁消耗资源远小于进程

3.3.2 多线程编程

- 线程模块： threading



- 创建方法

【1】 创建线程对象

```
from threading import Thread

t = Thread()
功能：创建线程对象
参数：target 绑定线程函数
      args   元组 给线程函数位置传参
      kwargs 字典 给线程函数键值传参
      daemon bool值，主线程退出时该分支线程也退出
```

【2】启动线程

```
 t.start()
```

【3】等待分支线程结束

```
t.join([timeout])
功能：阻塞等待分支线程退出
参数：最长等待时间
```

```
"""
    线程示例01：
"""

import threading
from time import sleep
import os

a = 1

#  线程函数
def music():
    global a
    print("a =",a)
    a = 10000
    for i in range(3):
        sleep(2)
        print(os.getpid(),"播放:黄河大合唱")

# 实例化线程对象
thread = threading.Thread(target=music)
# 启动线程 线程存在
```

```
thread.start()

for i in range(4):
    sleep(1)
    print(os.getpid(),"播放:葫芦娃")

# 阻塞等待分支线程结束
thread.join()
print("a:",a)
```

```
"""
    线程示例02:
"""

from threading import Thread
from time import sleep

# 带有参数的线程函数
def func(sec,name):
    print("含有参数的线程来喽")
    sleep(sec)
    print(f"{name} 线程执行完毕")

# 循环创建线程
for i in range(5):
    t = Thread(target=func,
               args=(2,),
               kwargs={"name":f"T-{i}"},
               daemon=True)
    t.start()
    sleep(5)
```

- `target=func`：指定线程执行的目标函数为 `func`。
- `args=(2,)`：传递给 `func` 函数的位置参数是 `(2,)`，即 `sec` 参数为 `2`。
- `kwargs={"name": f"T-{i}"}`：传递给 `func` 函数的关键字参数 `name`，根据循环变量 `i` 动态生成不同的线程名称，例如 `T-0`、`T-1` 等。

- `daemon=True`：将线程设置为守护线程，即随着主线程的结束而结束。

3.3.3 创建线程类

1. 创建步骤

   【1】 继承Thread类

   【2】 重写`__init__`方法添加自己的属性，使用super()加载父类属性

   【3】 重写run()方法

2. 使用方法

   【1】 实例化对象

   【2】 调用start自动执行run方法

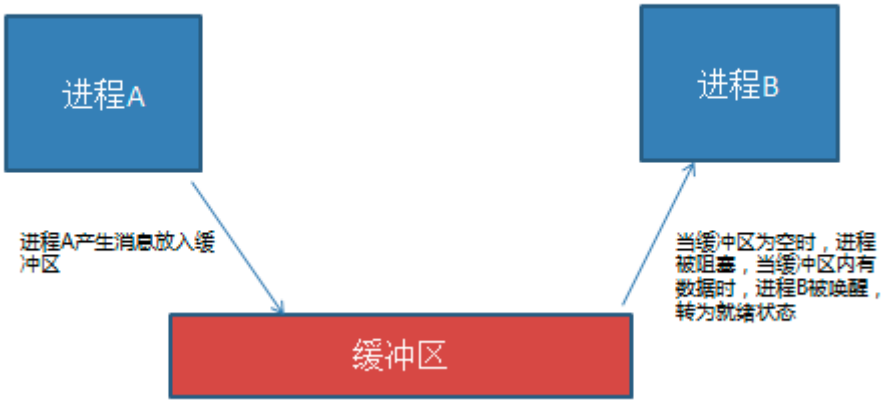
```
from threading import Thread
from time import sleep

class MyThread(Thread):
    def __init__(self,song):
        self.song = song
        super().__init__() # 得到父类内容

    # 线程要做的事情
    def run(self):
        for i in range(3):
            sleep(2)
            print("播放:",self.song)

t = MyThread("凉凉")
t.start() # 运行run
```

随堂练习：
现在有500张票，存在一个列表中 ["T1",....."T500"]，10个窗口同时卖这500张票 W1-W10

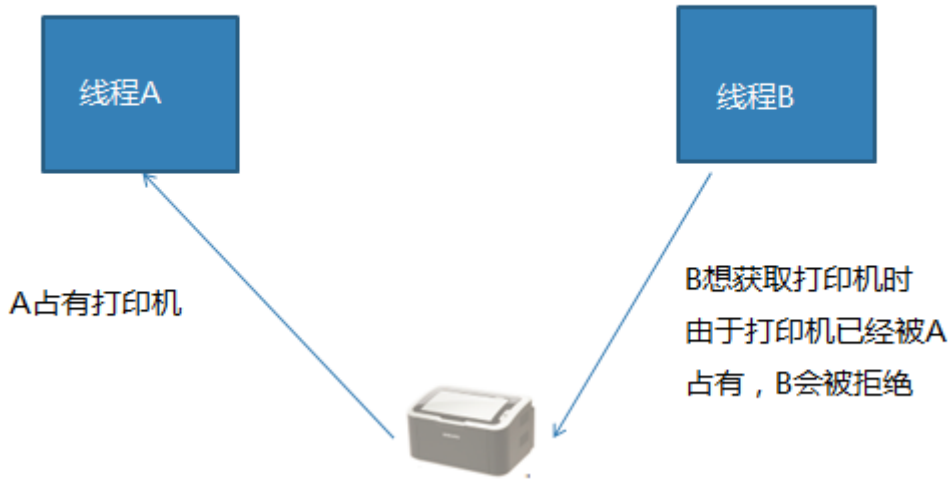
使用10个线程模拟这10个窗口，同时卖票，直到所有的票都卖出为止，每出一张票需要0.1秒，打印表示即可print("W1----T250")

3.3.4 线程同步互斥

- 线程通信方法： 线程间使用全局变量进行通信
- 共享资源争夺
  - 共享资源：多线程都可以操作的资源称为共享资源，对共享资源进行操作的代码段称为临界区
  - 影响：对共享资源的无序操作可能会带来数据的混乱，或操作错误，此时需要同步互斥机制协调操作顺序
- 同步互斥机制
  - 同步：同步是一种协作关系，为完成操作，线程间形成一种协调，按照必要的步骤有序执行操作



  - 互斥： 互斥是一种制约关系，当一个线程占有资源时会进行加锁处理，此时其他进程线程就无法操作该资源，直到解锁后才能操作



- 线程Event

```
from threading import Event

e = Event()  创建线程event对象

e.wait([timeout])  阻塞等待e被set

e.set()  设置e，使wait结束阻塞

e.clear() 使e回到未被设置状态

e.is_set()  查看当前e是否被设置
```

```
"""
    Event使用示例：
"""

from threading import Thread, Event

msg = None  # 通信变量
e = Event()  # 事件对象


def yzr():
    print("杨子荣前来拜山头")
    global msg
    msg = "天王盖地虎"
```

```
    e.set()  # 通知主线程可以判断


t = Thread(target=yzr)
t.start()

print("说对口令才是自己人")
e.wait()  # 阻塞等待通知
if msg == "天王盖地虎":
    print("宝塔镇河妖")
    print("确认过眼神你是对的人")
else:
    print("打死他.... 无情啊 哥哥....")
```

- 线程锁 Lock

```
from  threading import Lock

lock = Lock()  创建锁对象
lock.acquire() 上锁  如果lock已经上锁再调用会阻塞
lock.release() 解锁
```

```
'''
锁：Lock
'''

from threading import Thread, Lock

lock = Lock()
# 多线程共享资源
a = b = 1


def print_value():
    while True:
        lock.acquire()
        if a != b:
            print(f'a={a},b={b}')
        else:
            print('equal')
```

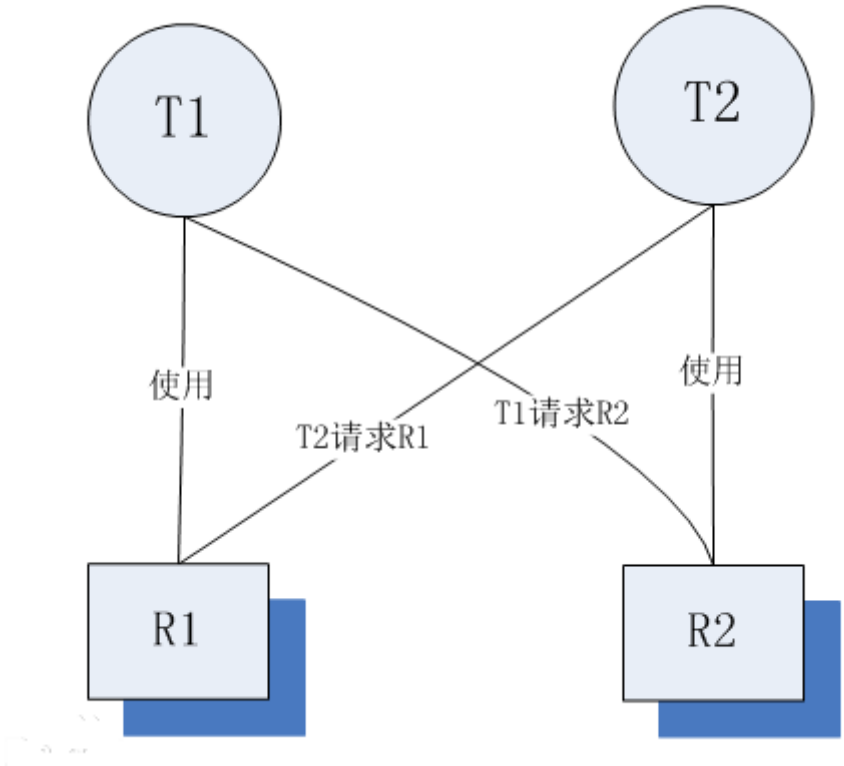
```
        lock.release()


if __name__ == '__main__':
    t = Thread(target=print_value)
    t.start()

    while True:
        try:
            lock.acquire()
            a += 1
            # ...cpu进行切换
            b += 1
            lock.release()
        except Exception as e:
            print(e)
```

随堂练习：
使用两个分支线程，一个线程打印1-52 这52个数字，另一个线程打印A-Z 这26个字母。要求同时执行两个线程，打印顺序为： 12A34B....5152Z

3.3.5 死锁

- 什么是死锁

  死锁是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。



- 死锁产生条件
  - 互斥条件：指线程使用了互斥方法，使用一个资源时其他线程无法使用
  - 请求和保持条件：指线程已经保持至少一个资源，但又提出了新的资源请求，在获取到新的资源前不会释放自己保持的资源
  - 不剥夺条件：不会受到线程外部的干扰，如系统强制终止线程等
  - 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，如 T0正在等待一个T1占用的资源；T1正在等待T2占用的资源，……，Tn正在等待已被T0占用的资源。
- 如何避免死锁
  - 逻辑清晰，不要同时出现上述死锁产生的四个条件
  - 通过测试工程师进行死锁检测

```
"""
死锁现象演示
"""
from time import sleep
from threading import Thread,Lock

# 账户类
class Account:
```

```
    def __init__(self,id,balance,lock):
        self._id = id
        self._balance = balance
        self.lock = lock

    # 取钱
    def withdraw(self,amount):
        self._balance -= amount

    # 存钱
    def deposit(self,amount):
        self._balance += amount

    # 查看余额
    def getBalance(self):
        return self._balance


# 转账函数
def transfer(from_,to,amount):
    from_.lock.acquire()
    from_.withdraw(amount) # from_钱减少
    from_.lock.release() # 不会产生死锁
    sleep(0.1) # 网络延迟

    to.lock.acquire()
    to.deposit(amount) # to钱增加

    # from_.lock.release() # 产生死锁
    to.lock.release()

if __name__ == '__main__':
    tom = Account("Tom",5000,Lock())
    abby = Account("abby",8000,Lock())

    t1 = Thread(target=transfer,args=(tom,abby,2000))
    t2 = Thread(target=transfer,args=(abby,tom,3000))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

```
    print("Tom:",tom.getBalance())
    print("Abby:",abby.getBalance())
```

3.3.6 GIL问题

- 什么是GIL（Global Interpreter Lock）问题 （全局解释器锁）

  由于python解释器设计中加入了解释器锁，导致python解释器同一时刻只能解释执行一个线程，大大降低了线程的执行效率

- 导致后果
  因为遇到阻塞时线程会主动让出解释器，去解释其他线程，所以python多线程在执行多阻塞任务时可以提升程序效率，其他情况并不能对效率有所提升
- 关于GIL问题的处理

> 尽量使用进程完成无阻塞的并发行为
>
> 不使用C语言版解释器 （可以用Java、C#）
> Guido的声明： **http://www.artima.com/forums/flat.jsp?forum=106&thread=214235**

- 结论
  - GIL问题与Python语言本身并没什么关系，属于解释器设计的历史问题
  - 在无阻塞状态下，多线程程序程序执行效率并不高，甚至还不如单线程效率
  - Python多线程只适用于执行有阻塞延迟的任务情形

线程效率对比进程实验：

```
class Prime(Thread):
    # 判断一个数是否为质数
    @staticmethod
    def is_prime(n):
        if n <= 1:
            return False
        for i in range(2,n // 2 + 1):
```

```
            if n % i == 0:
                return False
        return True

    def __init__(self,begin,end):
        self.__begin = begin
        self.__end = end
        super().__init__()

    def run(self):
        prime = [] # 存放所有质数
        for i in range(self.__begin,self.__end):
            if Prime.is_prime(i):
                prime.append(i)
        print(sum(prime))

@timeis
def process_10():
    jobs = []
    for i in range(1,100001,10000):
        t = Prime(i,i + 10000)
        jobs.append(t)
        t.start()
    for i in jobs:
        i.join()


if __name__ == '__main__':
    process_10()
```

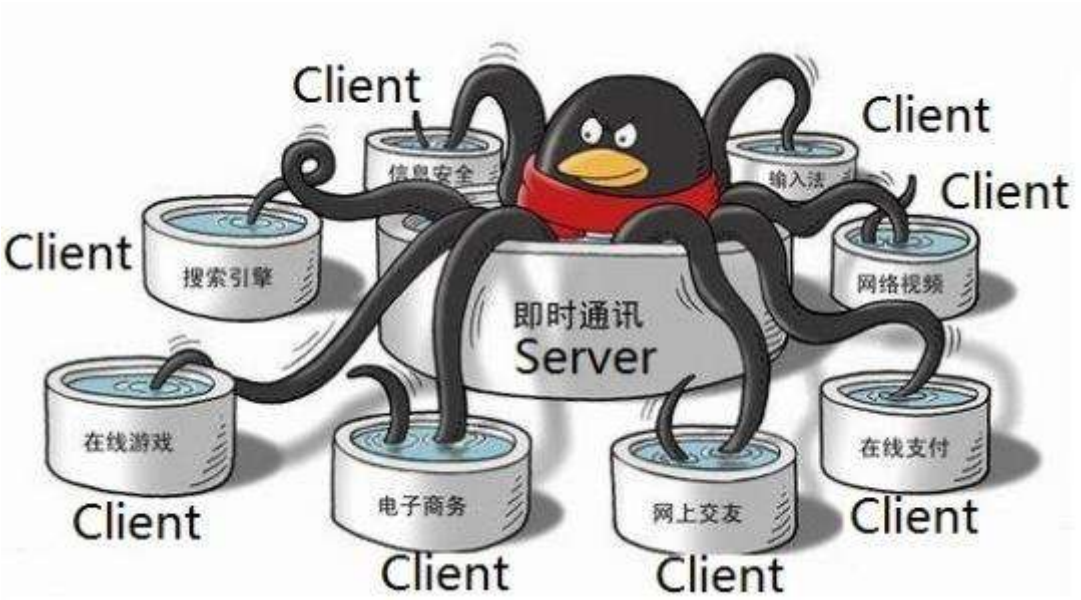
3.3.7 进程线程的区别联系

- 区别联系

1. 两者都是多任务编程方式，都能使用计算机多核资源
2. 进程的创建删除消耗的计算机资源比线程多
3. 进程空间独立，数据互不干扰，有专门通信方法；线程使用全局变量通信
4. 一个进程可以有多个分支线程，两者有包含关系

5. 多个线程共享进程资源，在共享资源操作时往往需要同步互斥处理
6. Python线程存在GIL问题，但是进程没有

- 使用场景



1. 任务场景：一个大型服务，往往包含多个独立的任务模块，每个任务模块又有多个小独立任务构成，此时整个项目可能有多个进程，每个进程又有多个线程
2. 编程语言：Java、C#之类的编程语言在执行多任务时，一般都是用线程完成，因为线程资源消耗少；而Python由于GIL问题往往使用多进程