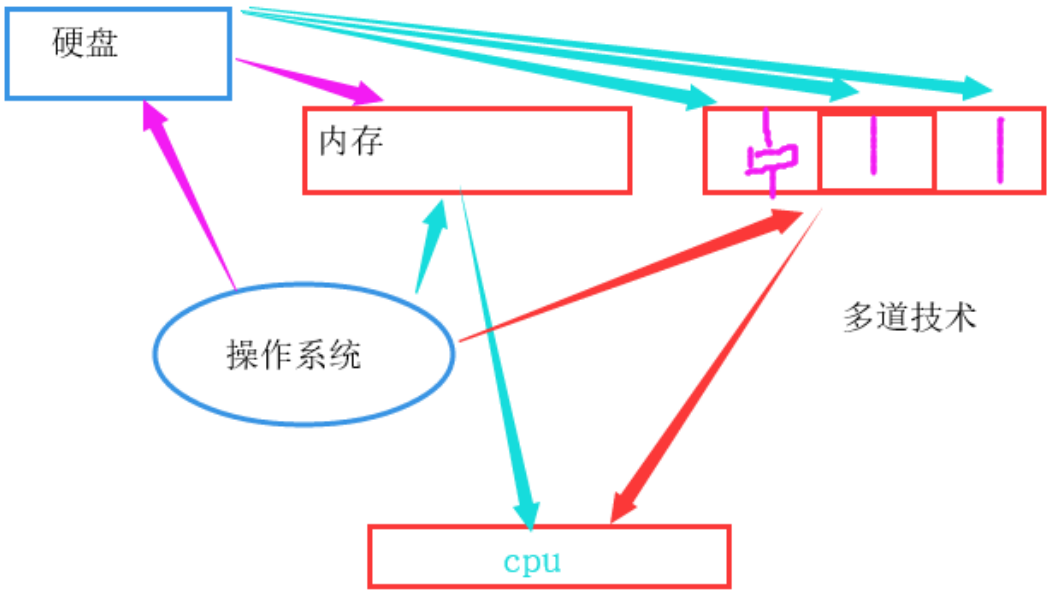
# 并发编程

## ---- 进程

### 1.什么是进程

#### 1.多道技术与分时系统




- 最初 -- 操作系统将硬盘中的应用程序读到内存，此时程序是一个个的被读到内存中，操作系统将内存中的应用程序交给CPU去执行，当遇到IO即等待
- 多道技术 -- 此时在硬盘上的应用程序（比如有三个）就会全部读到内存中，操作系统为其划分不同的内存空间，此时不同内存空间中的应用程序，内存空间，以及操作系统对其的调度称为进程
- 由于将不同的应用程序放到不同的内存空间中，因此多道技术实现了空间复用，而当第一个被调度的应用程序出现IO，操作系统将其切换到另外一个进程，将应用程序交给CPU去执行，由此实现了时间复用
- 当此时进行的程序没由IO，则无法对其他的进程进行调度，由此出现了分时系统，每隔一定的时间，就切换进程，但是这样的切换工作其实并没有提高cpu的效率，反而使得计算机的效率降低了。因为CPU需要切换，并且记录每次切换程序执行到了哪里，以便下次再切换回来的时候能够继续之前的程序，虽然我们牺牲了一点效率，但是却实现了多个程序共同执行的效果

#### 2.进程的概念

- 进程是指应用程序，内存空间，操作系统的调度称为一个进程
- 进程是竞争计算机系统有限资源的基本单位，也是进行处理机调度的基本单位
- 进程是程序的基本执行实体

#### 3.并发与并行

- 并发：是伪并行，即看起来是同时运行。单个cpu+多道技术就可以实现并发
- 并行：同时运行，只有具备多个cpu才能实现并行

#### 4.同步 、异步 | 阻塞、非阻塞

- 同步 、异步: 是指任务提交的方式 同步:提交任务得等上一个任务提交完毕,才能提交 ,如串行 异步:任务的提交互不影响
- 阻塞 、非阻塞 : 指程序执行中的运行状态 阻塞: 出现io 非阻塞: 没有出现 io
- 异步非阻塞: 程序在执行过程中没有出现io,任务提交是异步,无需等待

#### 5.进程的创建与结束

- 进程的创建
  - 系统初始化（查看进程linux中用ps命令，windows中用任务管理器，前台进程负责与用户交互，后台运行的进程与用户无关，运行在后台并且只在需要时才唤醒的进程，称为守护进程，如电子邮件、web页面、新闻、打印）
  - 一个进程在运行过程中开启了子进程（如nginx开启多进程，os.fork,subprocess.Popen等）
  - 用户的交互式请求，而创建一个新进程（如用户双击暴风影音）
  - 一个批处理作业的初始化（只在大型机的批处理系统中应用）
- 进程的结束
  - 正常退出（自愿，如用户点击交互式页面的叉号，或程序执行完毕调用发起系统调用正常退出，在linux中用exit，在windows中用ExitProcess）
  - 出错退出（自愿，python a.py中a.py不存在）
  - 严重错误（非自愿，执行非法指令，如引用不存在的内存，1/0等，可以捕捉异常，try...except...）
  - 被其他进程杀死（非自愿，如kill -9）

## 2.multiprocess模块

#### 1.创建进程部分

**1.process模块:**

```
#当前文件名称为test.py
# from multiprocessing import Process
#
# def func():
#     print(12345)
#
# if __name__ == '__main__': #windows 下才需要写这个，这和系统创建进程的机制有关系，不用深究，记着windows下要写就好啦
#     #首先我运行当前这个test.py文件，运行这个文件的程序，那么就产生了进程，这个进程我们称为主进程
#
#     p = Process(target=func,) #将函数注册到一个进程中，p是一个进程对象，此时还没有启动进程，只是创建了一个进程对象。并且func是不加括号的，因为加上括号这个函数就直接运行了对吧。
#     p.start() #告诉操作系统，给我开启一个进程，func这个函数就被我们新开的这个进程执行了，而这个进程是我主进程运行过程中创建出来的，所以称这个新创建的进程为主进程的子进程，而主进程又可以称为这个新进程的父进程。
                #而这个子进程中执行的程序，相当于将现在这个test.py文件中的程序copy到一个你看不到的python文件中去执行了，就相当于当前这个文件，被另外一个py文件import过去并执行了。
                #start并不是直接就去执行了，我们知道进程有三个状态，进程会进入进程的三个状态，就绪，（被调度，也就是时间片切换到它的时候）执行，阻塞，并且在这个三个状态之间不断的转换，等待cpu执行时间片到了。
#     print('*' * 10) #这是主进程的程序，上面开启的子进程的程序是和主进程的程序同时运行的，我们称为异步
```

```
    #我们通过init方法可以传参数，如果只写一个run方法，那么没法传参数，因为创建对象的是传参就是在init方法里面，面向对象的时候，我们是不是学过
```

```
    def __init__(self,person):
        super().__init__()
        self.person=person
    def run(self):
        print(os.getpid())
        print(self.pid)
        print(self.pid)
        print('%s 正在和女主播聊天' %self.person)
    # def start(self):
    #     #如果你非要写一个start方法，可以这样写，并且在run方法前后，可以写一些其他的逻辑
    #     self.run()
if __name__ == '__main__':
    p1=MyProcess('Jedan')
    p2=MyProcess('太白')
    p3=MyProcess('alexDSB')

    p1.start() #start内部会自动调用run方法
    p2.start()
    # p2.run()
    p3.start()


    p1.join()
    p2.join()
    p3.join()
```

**2.进程同步部分**

进程之间数据不共享,但是共享同一套文件系统,所以访问同一个文件,或同一个打印终端,是没有问题的，而共享带来的是竞争，竞争带来的结果就是错乱，如何控制，就是加锁处理。

```
#由并发变成了串行,牺牲了运行效率,但避免了竞争
from multiprocessing import Process,Lock
import os,time
def work(n,lock):
    #加锁，保证每次只有一个进程在执行锁里面的程序，这一段程序对于所有写上这个锁的进程，大家都变成了串行
    lock.acquire()
    print('%s: %s is running' %(n,os.getpid()))
    time.sleep(1)
    print('%s:%s is done' %(n,os.getpid()))
    #解锁，解锁之后其他进程才能去执行自己的程序
    lock.release()
if __name__ == '__main__':
    lock=Lock()
    for i in range(5):
        p=Process(target=work,args=(i,lock))
        p.start()
```

**3. 进程池部分**

**1.进程池的概念**

- 定义一个池子，在里面放上固定数量的进程，有需求来了，就拿一个池中的进程来处理任务，等到处理完毕，进程并不关闭，而是将进程再放回进程池中继续等待任务。
- 如果有很多任务需要执行，池中的进程数量不够，任务就要等待之前的进程执行任务完毕归来，拿到空闲进程才能继续执行。
- 也就是说，池中进程的数量是固定的，那么同一时间最多有固定数量的进程在运行。这样不会增加操作系统的调度难度，还节省了开闭进程的时间，也一定程度上能够实现并发效果

**2.multiprocess.Pool模块**

```
# -------- 进程池 -----------
#为什么要使用进程池:由于重复的开启和销毁一个进程的开销比较大,定义一个池子，在里面放上固定数量的进程，有需求来了，就拿一个池中的进程来处理任务，等到处理完毕，进程并不关闭，而是将进程再放回进程池中继续等待任务

# -------------------------apply ------------------------
import os,time
from multiprocessing import Pool

def work(n):
    print('%s run' %os.getpid())
    time.sleep(1)
    return n**2

if __name__ == '__main__':
    p=Pool(3) #进程池中从无到有创建三个进程,以后一直是这三个进程在执行任务
    res_l=[]
    for i in range(10):
        res=p.apply(work,args=(i,)) # 同步调用，直到本次任务执行完毕拿到res，等待任务work执行的过程中可能有阻塞也可能没有阻塞
                                    # 但不管该任务是否存在阻塞，同步调用都会在原地等着
        res_l.append(res)
    print(res_l)
# ------------------------map---------------------------------

#if __name__ == '__main__':
   # poll = Pool(5) # 创建含有5个进程的进程池
    # poll.map(func,range(100)) #异步调用进程，开启100个任务,map自带join的功能
    poll.map(work,[(1,2),'alex']) #异步调用进程，开启100个任务,map自带join的功能
    # poll.map(func2,range(100))  #如果想让进程池完成不同的任务，可以直接这样搞
    #map只限于接收一个可迭代的数据类型参数，列表啊，元祖啊等等，如果想做其他的参数之类的操作，需要用后面我们要学的方法。

# ------------------------apply_async-----------------
if __name__ == '__main__':
    p=Pool(3) #进程池中从无到有创建三个进程,以后一直是这三个进程在执行任务
    res_l=[]
    for i in range(10):
        res=p.apply_async(work,args=(i,))
        # 异步运行，根据进程池中有的进程数，每次最多3个子进程在异步执行，并且可以执行不同的任务，传送任意的参数了。
        # 返回结果之后，将结果放入列表，归还进程，之后再执行新的任务
        # 需要注意的是，进程池中的三个进程不会同时开启或者同时结束
        # 而是执行完一个就释放一个进程，这个进程就去接收新的任务。
        res_l.append(res)
```

```
    # 异步apply_async用法：如果使用异步提交的任务，主进程需要使用join，等待进程池内任务都处理完，然后可以用get收集结果
    # 否则，主进程结束，进程池可能还没来得及执行，也就跟着一起结束了
    p.close() #不是关闭进程池，而是结束进程池接收任务，确保没有新任务再提交过来。
    p.join()   #感知进程池中的任务已经执行结束，只有当没有新的任务添加进来的时候，才能感知到任务结束了，所以在join之前必须加上close方法
    for res in res_l:
        print(res.get()) #使用get来获取apply_aync的结果,如果是apply,则没有get方法,因为apply是同步执行,立刻获取结果,也根本无需get
```

**4.进程之间数据共享 - 进程之间的通信**

**1.基于Queue的通信**

```
from multiprocessing import Queue
q = Queue(3)         #创建一个队列对象,队列长度为3

q.put(1)          #往队列中添加数据
q.put(2)
q.put(3)
# q.put(4)            #如果队列已满,程序就会停在这里,等待数据被别人取走,再将数据放入队列
                      #如果队列中的数据一直不被取走,程序就会永远停在这
try:
    q.put_nowait(3)    #可以使用put_nowait,如果队列满了不会阻塞,但是会因为队列满了就报错.
except:                #因此,我们用一个try语句来处理这个错误,这样程序不会一直阻塞下去,但是会丢掉该消息
    print("队列已经满了")

#因此,我们在放入数据之前,可以先看一下队列的状态,如果已经满了就不在put了
print(q.full())             #查看是否满了,满了返回True,不满返回False

print(q.get())
print(q.get())
print(q.get())
# print(q.get())               #同put方法一样,如果队列已经空了,那么读取就会阻塞
try:
    q.get_nowait(3) # 可以使用get_nowait，如果队列满了不会阻塞，但是会因为没取到值而报错。
except:             # 因此我们可以用一个try语句来处理这个错误。这样程序不会一直阻塞下去。
    print('队列已经空了')

# -------关于q.empty()--------
#在空队列上放置对象之后，在队列的p.empty()方法返回False之前，可能有无限小的延迟,导致返回的结果是True

#基于队列的生产者消费者模型:问题,就是当队列空了之后,程序会阻塞,进程不会结束,原因是：生产者p在生产完后就结束了，但是消费者c在取空了q之后，则一直处于死循环中且卡在q.get()这一步。
```

**2.JoinableQueue**

```
#JoinableQueue([maxsize]):这就像是一个Queue对象，但队列允许项目的使用者通知生成者项目已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。

#参数介绍:
  maxsize是队列中允许最大项数，省略则无大小限制。
#方法介绍:
  JoinableQueue的实例p除了与Queue对象相同的方法之外还具有:
  q.task_done():消费者使用此方法发出信号，表示q.get()的返回项目已经被处理。如果调用此方法的次数大于从队列中删除项目的数量，将引发ValueError异常
  q.join():生产者调用此方法进行阻塞，直到队列中所有的项目均被处理。阻塞将持续到队列中的每个项目均调用q.task_done () 方法为止，也就是队列中的数据全部被get拿走了。

#在该模型中,对消费者设置守护进程 c.deamon = True ,对生产者设置p.join()  -> 当主进程结束意味着:
    #生产者中的 q.join()接收到队列中所有的task_done信号,队列中的任务都被处理完,生产者的子进程结束,主 进程随之结束
```

3.管道

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send("Hello 妹妹") #子进程发送了消息 什么数据类型都可以
    conn.close()
if __name__ == '__main__':
    parent_conn, child_conn = Pipe() #建立管道，拿到管道的两端，双工通信方式，两端都可以收发消息
    p = Process(target=f, args=(child_conn,)) #将管道的一段给子进程
    p.start() #开启子进程
    print(parent_conn.recv()) #主进程接受了消息
    p.join()
#管道通信不安全:
#由Pipe方法返回的两个连接对象表示管道的两端。每个连接对象都有send和recv方法（除其他之外）。注意，如果两个进程（或线程）试图同时从管道的同一端读取或写入数据，那么管道中的数据可能会损坏
```

4.信号量

5.事件

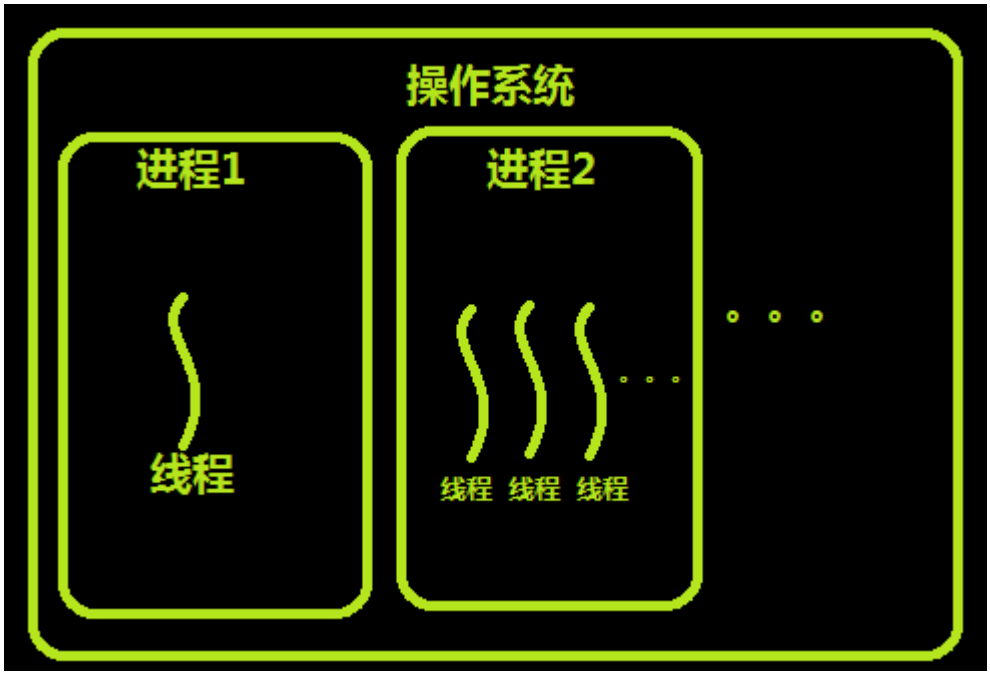
## 3.一些概念

# ---- 线程

## 1.什么是线程

进程想要执行任务需要依赖线程，换句话说就是进程中的最小执行单位就是线程，并且一个进程中至少有一个线程。

### 1.进程与线程的关系




- 地址空间和其它资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见
- 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。（就类似进程中的锁的作用）
- 调度和切换：线程上下文切换比进程上下文切换要快得多。
- 在多线程操作系统中（现在咱们用的系统基本都是多线程的操作系统），进程不是一个可执行的实体，真正去执行程序的不是进程，是线程，你可以理解进程就是一个线程的容器。

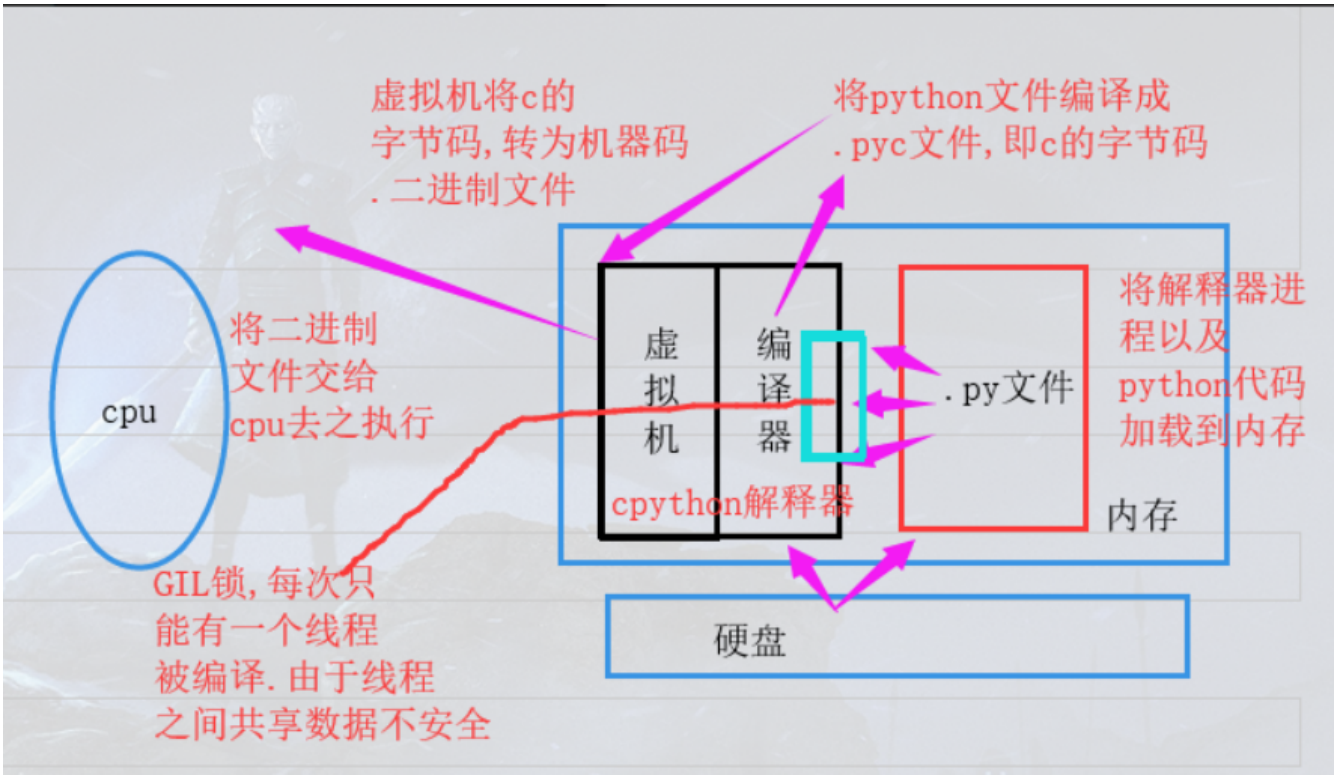
**2.多线程**

多线程（即多个控制线程）的概念是，在一个进程中存在多个控制线程，多个控制线程共享该进程的地址空间，相当于一个车间内有多条流水线，都共用一个车间的资源。

## 2.python与线程

**1.全局解释器锁GIL**

Python代码的执行由Python虚拟机(也叫解释器主循环)来控制。Python在设计之初就考虑到要在主循环中，同时只有一个线程在执行。虽然 Python 解释器中可以“运行”多个线程，但在任意时刻只有一个线程在解释器中运行。




在多线程环境中，Python 虚拟机按以下方式执行：

a、设置 GIL；

b、切换到一个线程去运行；

c、运行指定数量的字节码指令或者线程主动让出控制(可以调用 time.sleep(0))；

d、把线程设置为睡眠状态；

e、解锁 GIL；

d、再次重复以上所有步骤。

## 2.threading模块

### 1.线程的创建

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('太白',))
    t.start()
    print('主线程')
```

```
import time
from threading import Thread
class Sayhi(Thread):
    def __init__(self,name):
```

```
        super().__init__()
        self.name=name
    def run(self):
        time.sleep(2)
        print('%s say hello' % self.name)


if __name__ == '__main__':
    t = Sayhi('太白')
    t.start()
    print('主线程')
```

**2.线程同步 - 锁**

**1. GIL锁存在的意义**

- GIL锁相当于执行权限，它并保护线程之间共享的数据
- 拿到执行权限后才能拿到互斥锁Lock，其他线程也可以抢到GIL，但如果发现Lock仍然没有被释放则阻塞，即便是拿到执行权限GIL也要立刻交出来

**2.互斥锁**

锁通常被用来实现对共享资源的同步访问。为每一个共享资源创建一个Lock对象，当你需要访问该资源时，调用acquire方法来获取锁对象（如果其它线程已经获得了该锁，则当前线程需等待其被释放），待资源访问完后，再调用release方法释放锁：

```
import threading

R=threading.Lock()

R.acquire() #
#R.acquire()如果这里还有一个acquire，你会发现，程序就阻塞在这里了，因为上面的锁已经被拿到了并且还没有释放的情况下，再去拿就阻塞住了
'''
对公共数据的操作
'''
R.release()
```

**3.死锁与递归锁**

所谓死锁： 是指两个或两个以上的进程或线程在执行过程中，因争夺资源而造成的一种互相等待的现象，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，这些永远在互相等待的进程称为死锁进程，如下就是死锁

```
# ------死锁 -------
from threading import Thread,Lock,RLock
import time
mutexA=Lock()
mutexB=Lock()
# mutexA = mutexB = RLock()
class MyThread(Thread):
    def run(self):
        self.func1()
```

```
            self.func2()
        def func1(self):
            mutexA.acquire()
            print('%s 拿到A锁>>>' %self.name)
            mutexB.acquire()
            print('%s 拿到B锁>>>' %self.name)
            mutexB.release()
            mutexA.release()

        def func2(self):
            mutexB.acquire()
            print('%s 拿到B锁???' %self.name)
            time.sleep(0.5)

            mutexA.acquire()
            print('%s 拿到A锁???' %self.name)
            mutexA.release()

            mutexB.release()

if __name__ == '__main__':
        for i in range(10):
            t=MyThread()
            t.start()

#上述程序形成死锁现象:数据分析,数据库出现比较多,双方拿着对方想要抢的锁
#解决方法，递归锁，在Python中为了支持在同一线程中多次请求同一资源，python提供了可重入锁RLock。

#这个RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次require。直到一个线程所有的acquire都被release，其他的线程才能获得资源。上面的例子如果使用RLock代替Lock，则不会发生死锁：
```

**3.线程之间的通信 - 线程队列**

```
import queue #不需要通过threading模块里面导入，直接import queue就可以了，这是python自带的
#用法基本和我们进程multiprocess中的queue是一样的
q=queue.Queue()
q.put('first')
q.put('second')
q.put('third')
# q.put_nowait() #没有数据就报错，可以通过try来搞
print(q.get())
print(q.get())
print(q.get())
# q.get_nowait() #没有数据就报错，可以通过try来搞
'''
结果(先进先出):
first
second
third
```

```
import queue
```

```
q=queue.LifoQueue() #队列，类似于栈，栈我们提过吗，是不是先进后出的顺序啊
q.put('first')
q.put('second')
q.put('third')
# q.put_nowait()

print(q.get())
print(q.get())
print(q.get())
# q.get_nowait()
'''
结果(后进先出):
third
second
first
```

```
import queue

q=queue.PriorityQueue()
#put进入一个元组,元组的第一个元素是优先级(通常是数字,也可以是非数字之间的比较),数字越小优先级越高
q.put((-10,'a'))
q.put((-5,'a'))  #负数也可以
# q.put((20,'ws'))  #如果两个值的优先级一样，那么按照后面的值的acsii码顺序来排序,如果字符串第一个数元素相同，比较第二个元素的acsii码顺序
# q.put((20,'wd'))
# q.put((20,{'a':11})) #TypeError: unorderable types: dict() < dict() 不能是字典
# q.put((20,('w',1)))  #优先级相同的两个数据，他们后面的值必须是相同的数据类型才能比较，可以是元祖，也是通过元素的ascii码顺序来排序

q.put((20,'b'))
q.put((20,'a'))
q.put((0,'b'))
q.put((30,'c'))

print(q.get())
print(q.get())
print(q.get())
print(q.get())
print(q.get())
print(q.get())
'''
结果(数字越小优先级越高,优先级高的优先出队):
'''
```

**4.线程池**

```
concurrent.futures模块提供了高度封装的异步调用接口
ThreadPoolExecutor：线程池，提供异步调用
ProcessPoolExecutor: 进程池，提供异步调用
Both implement the same interface, which is defined by the abstract Executor class.
```

```
#2 基本方法
#submit(fn, *args, **kwargs)
异步提交任务

#map(func, *iterables, timeout=None, chunksize=1)
取代for循环submit的操作

#shutdown(wait=True)
相当于进程池的pool.close()+pool.join()操作
wait=True，等待池内所有任务执行完毕回收完资源后才继续
wait=False，立即返回，并不会等待池内的任务执行完毕
但不管wait参数为何值，整个程序都会等到所有任务执行完毕
submit和map必须在shutdown之前

#result(timeout=None)
取得结果

#add_done_callback(fn)
回调函数
```

### 4.一些概念

# ---- 协程

## 1.什么是协程

协程本质上就是一个线程，以前线程任务的切换是由操作系统控制的，遇到I/O自动切换，现在我们用协程的目的就是较少操作系统切换的开销（开关线程，创建寄存器、堆栈等，在他们之间进行切换等），在我们自己的程序里面来控制任务的切换。

协程是一种用户态的轻量级线程，即线程是由用户程序自己控制调度的

#### 1.基于yield实现一个协程

##### 1.可迭代对象与迭代器

- 可迭代的对象：实现了`__iter__`方法的对象
- 迭代器：具有next方法的对象都是迭代器。在调用next方法时，迭代器会返回它的下一个值，如果next方法被调用，但迭代器没有值可以返回，就会引发一个StopIteration异常

##### 2.生成器

任何包含yield语句的函数都称为生成器，带有 yield 的函数不再是一个普通函数，而是一个生成器generator，可用于迭代。生成器就是一个迭代器

##### 3. yield的作用:

- 它和return差不多的用法，只是拥有它的语法结构最后是返回了一个生成器。
- 当你调用yield所在的那个函数或者生成器表达式的时候，那个函数并没有运行，只会返回一个生成器的对象。
- 当你第一次在for中调用生成器的的对象，它将会运行你函数中的代码从最开始一直到到碰到了yield的关键字，然后它会返回循环中的第一个值。然后每一次其他的调用将会运行你在这个函数中所写的循环多一次(`第二次循环从第一次返回的yield位置后面开始到遇到下一个yield`)，并且返回下一个值，直到没有值可以返回了,此时迭代器遍历完成。

#### 4.与生成器(实际上是迭代器)相关的next()和send()方法

- 对于普通的生成器，第一个next调用，相当于启动生成器，会从生成器函数的第一行代码开始执行，直到第一次执行完yield语句
- send(msg)与next()都有返回值，它们的返回值是当前迭代遇到yield时，yield后面表达式的值，其实就是当前迭代循环()中yield后面的参数
- 第一次调用时必须先next()或send(None)，否则会报错，send后之所以为None是因为这时候没有上一个yield(根据第8条)。可以认为，next()等同于send(None)
- send可以强行修改上一个yield表达式值, 多了一次赋值的动作。 send语句伴随着类似`n1 = yield ret`的结构, 旨在从循环外传入数据而影响循环

#### 5. yield实现协程 - 任务之间的切换加保存状态

```
#基于yield并发执行，多任务之间来回切换，这就是个简单的协程的体现，但是他能够节省I/O时间吗？不能
import time
def consumer():
    '''任务1:接收数据,处理数据'''
    while True:
        x=yield
        # time.sleep(1) #发现什么？只是进行了切换，但是并没有节省I/O时间
        print('处理了数据: ',x)
def producer():
    '''任务2:生产数据'''
    g=consumer()
    next(g)  #找到了consumer函数的yield位置
    for i in range(3):
    # for i in range(10000000):
        g.send(i)  #给yield传值，然后再循环给下一个yield传值，并且多了切换的程序，比直接串行执行还多了一些步骤，导致执行效率反而更低了。
        print('发送了数据: ',i)
start=time.time()
#基于yield保存状态,实现两个任务直接来回切换,即并发的效果
#PS:如果每个任务中都加上打印,那么明显地看到两个任务的打印是你一次我一次,即并发执行的.
producer() #我在当前线程中只执行了这个函数，但是通过这个函数里面的send切换了另外一个任务
stop=time.time()
```

并没有实现IO切换，只是切换了任务 + 保存状态，并没有实现提高效率

#### 2.Greenlet

- 如果我们在单个线程内有20个任务，要想实现在多个任务之间切换，使用yield生成器的方式过于麻烦（需要先得到初始化一次的生成器，然后再调用send。。。非常麻烦），而使用greenlet模块可以非常简单地实现这20个任务直接的切换
- greenlet只是提供了一种比generator更加便捷的切换方式，当切到一个任务执行时如果遇到io，那就原地阻塞，仍然是没有解决遇到IO自动切换来提升效率的问题。

#### 3.Gevent

Gevent 是一个第三方库，可以轻松通过gevent实现并发同步或异步编程，在gevent中用到的主要模式是Greenlet，它是以C扩展模块形式接入Python的轻量级协程。 Greenlet全部运行在主程序操作系统进程的内部，但它们被协作式地调度。

```
#用法
g1=gevent.spawn(func,1,2,3,x=4,y=5)创建一个协程对象g1，spawn括号内第一个参数是函数名，如eat，后面可以有多个参数，可以是位置实参或关键字实参，都是传给函数eat的，spawn是异步提交任务

g2=gevent.spawn(func2)

g1.join() #等待g1结束

g2.join() #等待g2结束  有人测试的时候会发现，不写第二个join也能执行g2，是的，协程帮你切换执行了，但是你会发现，如果g2里面的任务执行的时间长，但是不写join的话，就不会执行完等到g2剩下的任务了

#或者上述两步合作一步：gevent.joinall([g1,g2])

g1.value#拿到func1的返回值
```

遇到IO阻塞时会自动切换任务

```
import gevent
def eat(name):
    print('%s eat 1' %name)
    gevent.sleep(2)
    print('%s eat 2' %name)

def play(name):
    print('%s play 1' %name)
    gevent.sleep(1)
    print('%s play 2' %name)


g1=gevent.spawn(eat,'egon')
g2=gevent.spawn(play,name='egon')
g1.join()
g2.join()
#或者gevent.joinall([g1,g2])
print('主')
```

- 上例gevent.sleep(2)模拟的是gevent可以识别的io阻塞
- 而time.sleep(2)或其他的阻塞,gevent是不能直接识别的需要用下面一行代码,打补丁,就可以识别了
- from gevent import monkey;monkey.patch_all()必须放到被打补丁者的前面，如time，socket模块之前
- 或者我们干脆记忆成：要用gevent，需要将from gevent import monkey;monkey.patch_all()放到文件的开头

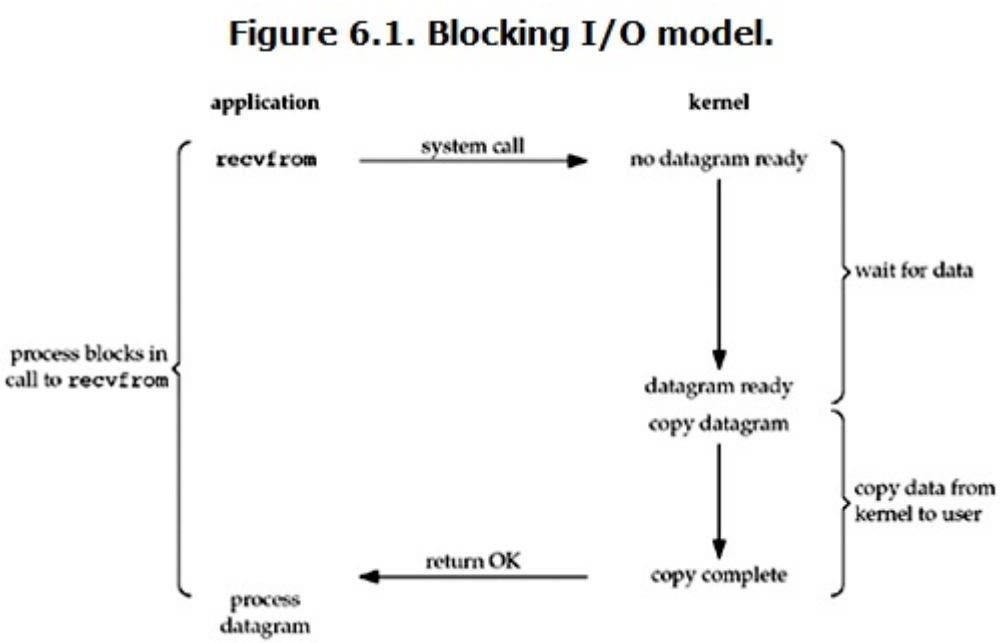
**4.Gevent之同步异步**

```
from gevent import spawn,joinall,monkey;monkey.patch_all()

import time
def task(pid):
    """
    Some non-deterministic task
    """
    time.sleep(0.5)
    print('Task %s done' % pid)
```

```
def synchronous():
    for i in range(10):
        task(i)

def asynchronous():
    g_l=[spawn(task,i) for i in range(10)]
    joinall(g_l)

if __name__ == '__main__':
    print('Synchronous:')
    synchronous()

    print('Asynchronous:')
    asynchronous()
#上面程序的重要部分是将task函数封装到Greenlet内部线程的gevent.spawn。 初始化的greenlet列表存放在数组threads中，此数组被传给gevent.joinall 函数，后者阻塞当前流程，并执行所有给定的greenlet。执行流程只会在所有greenlet执行完后才会继续向下走。
```

## ---- IO模型

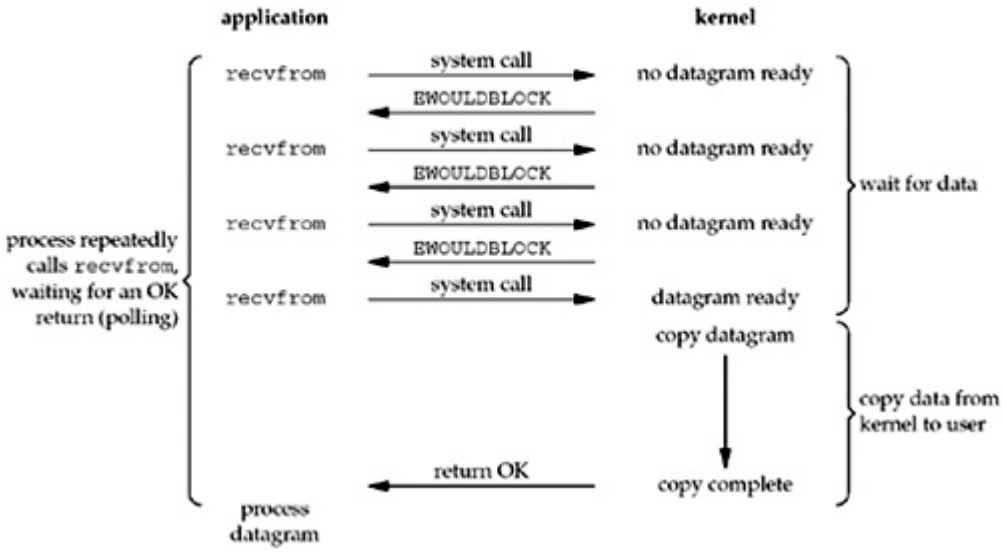
**1.阻塞IO**




**blocking IO的特点就是在IO执行的两个阶段（等待数据和拷贝数据两个阶段）都被block了**

**2.非阻塞IO模型**

Figure 6.2. Nonblocking I/O model.

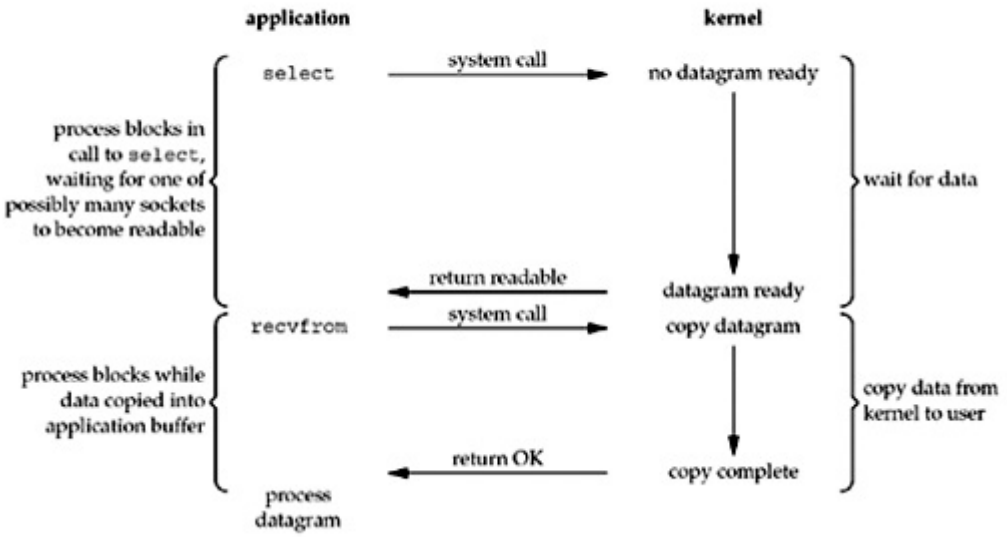



**nonblocking IO的特点是用户进程需要**不断的主动询问**kernel数据好了没有。**

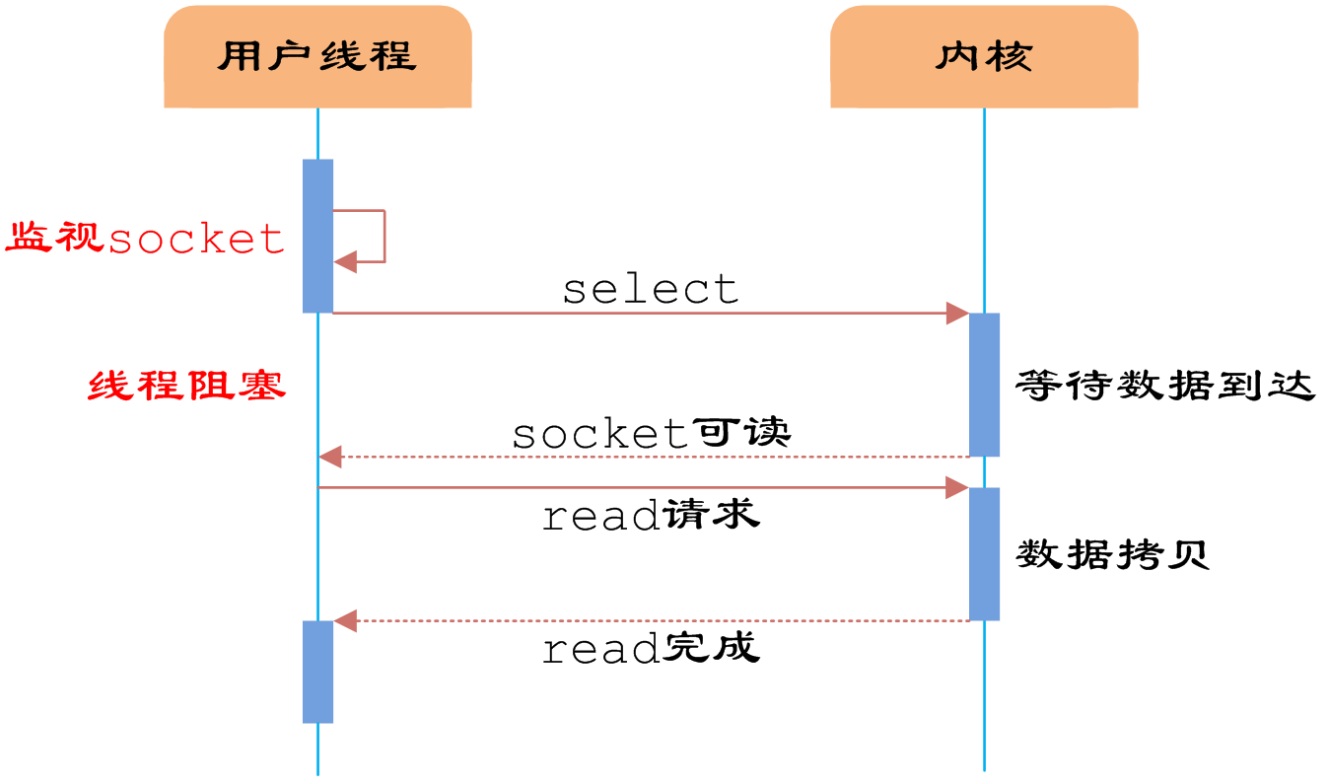
**3. IO多路复用**

- IO multiplexing就是我们说的select，poll，epoll，有些地方也称这种IO方式为event driven IO
- select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。
- 它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

Figure 6.3. I/O multiplexing model.




**1.多路分离函数select**




- 用户首先将需要进行IO操作的socket添加到select中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。
- 从流程上来看，使用select函数进行IO请求和同步阻塞模型没有太大的区别，甚至还多了添加监视socket，以及调用select函数的额外操作，效率更差。
- 但是，使用select以后最大的优势是用户可以在一个线程内同时处理多个socket的IO请求。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在**同一个线程内同时处理多个IO请求的目的**。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。
- 然而，使用select函数的优点并不仅限于此。虽然上述方式允许单线程内处理多个IO请求，但是每个IO请求的过程还是阻塞的（在select函数上阻塞），平均时间甚至比同步阻塞IO模型还要长。如果用户线程只注册自己感兴趣的socket或者IO请求，然后去做自己的事情，等到数据到来时再进行处理，则可以提高CPU的利用率。

**2.poll==>时间复杂度O(n)**

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

**3.epoll==>时间复杂度O(1)**

设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发?

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

**epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：**

- 调用**epoll_create()**建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)
- 调用**epoll_ctl**向epoll对象中添加这100万个连接的套接字
- 调用**epoll_wait**收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll_wait的效率也非常高，因为调用epoll_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

**Linux内核具体的epoll机制实现思路**

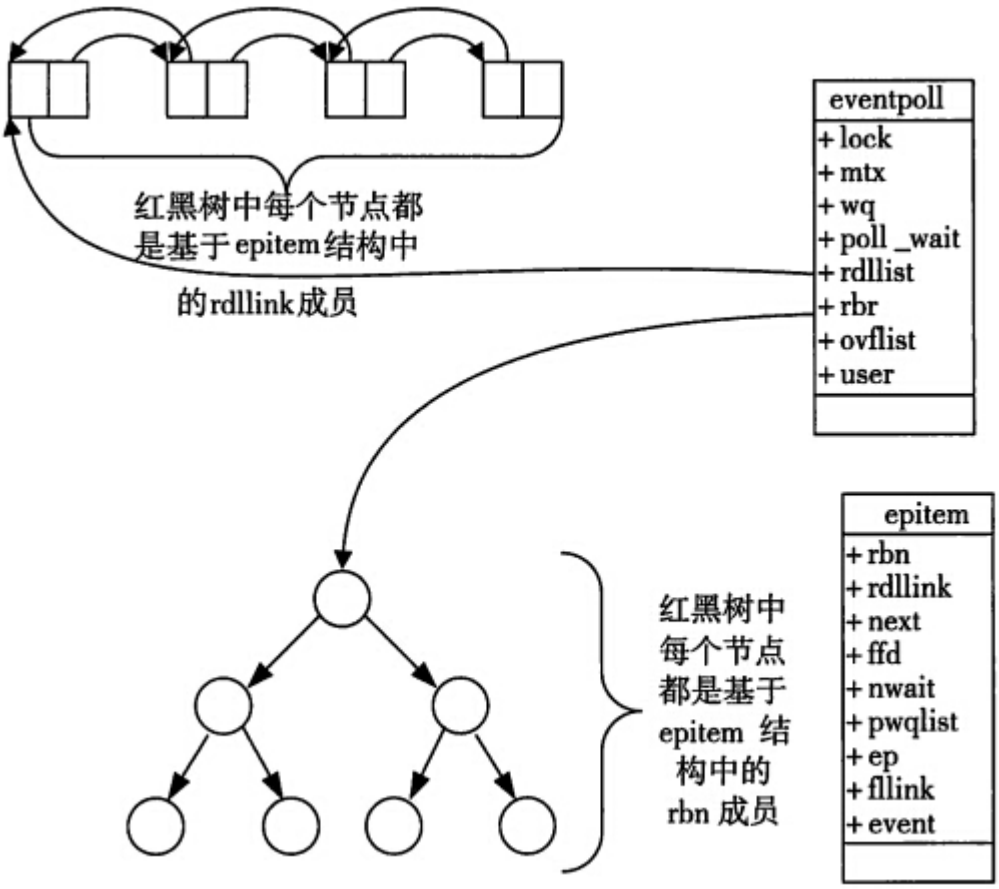
- 某一进程调用epoll_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：

```
struct eventpoll{
    ....
    /红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件/
    struct rb_root  rbr;
    /双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件/
    struct list_head rdlist;
    ....
};
```

- 每一个epoll对象都有一个独立的**eventpoll**结构体，用于存放通过epoll_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。
- 而**所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系**，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫**ep_poll_callback**,它会**将发生的事件添加到rdlist双链表**中。
- 在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：

```
struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head    rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}
```

当**调用epoll_wait检查是否有事件发生**时，只需要**检查eventpoll对象中的rdlist双链表中是否有epitem元素**即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。




epoll数据结构示意图

**通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。**

OK，讲解完了Epoll的机理，我们便能很容易掌握epoll的用法了。一句话描述就是：三步曲。

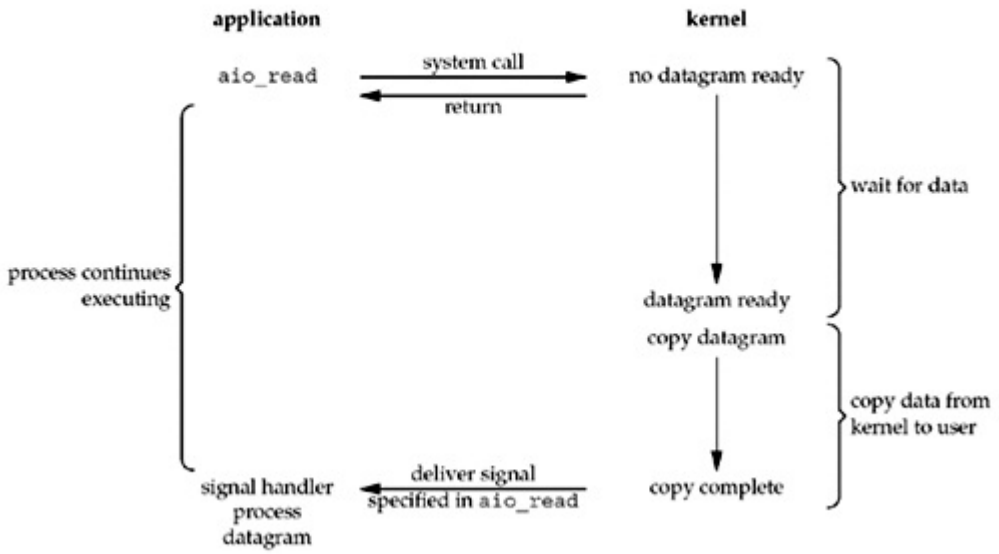
第一步：epoll_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣

第三步：epoll_wait()系统调用。通过此调用收集收集在epoll监控中已经发生的事件

**4.异步IO**

**Figure 6.5. Asynchronous I/O model.**




户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel操作系统会等待数据（阻塞）准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了