电网SCADA原型开发

（后台通信）

课堂练习（1） 模拟FTP

第1步， 客户端 put 文件， 服务端接收文件

```python
def put(sock, file_name):
    data_len = os.path.getsize(file_name)
    data_head = {"name": file_name, "size": data_len, "cmd": "put"}
    json_head = json.dumps(data_head).encode('utf8')

    # 发送报文头
    json_head_lens = struct.pack("i", len(json_head))
    sock.send(json_head_lens)
    sock.send(json_head)

    # 发送文件内容
    with open(file_name, "rb") as fp:
        send_len = 0
        while True:
            data = fp.read(10240)
            send_len += len(data)
            if not data:
                break
            process = int(send_len * 100.0 / data_len)
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')
            sock.send(data)
    print("上传成功")
```

client.py

```python
try:
    while True:
        head_recv = conn.recv(4)
        head_len = struct.unpack("i", head_recv)[0]

        head_recv = conn.recv(head_len)
        head_data = json.loads(head_recv)

        print("recv", head_data)

        file_cmd = head_data['cmd']

        func = globals().get(file_cmd, None)
        if func is None:
            raise ValueError(f"cmd error: {head_recv}")

        func(conn, head_data)

except Exception as e:
    print("Error:", e)
```

```python
def put(conn, head_data):
    file_name, file_len = head_data['name'], head_data['size']
    with open(file_name, "wb") as fp:
        recv_len = 0
        while recv_len < file_len:
            this_len = 10240 if file_len - recv_len > 10240 else file_len - recv_len
            data = conn.recv(this_len)
            fp.write(data)
            recv_len += this_len
```

server.py

课堂练习（1） 模拟FTP

第2步， 客户端 get 文件， 服务端发送文件

```python
def get(sock, file_name):
    data_head = {"name": file_name, "cmd": "get"}
    json_head = json.dumps(data_head).encode('utf8')

    # 发送报文头
    json_head_lens = struct.pack("i", len(json_head))
    sock.send(json_head_lens)
    sock.send(json_head)

    # 接受报文头
    head_recv = sock.recv(4)
    head_len = struct.unpack("i", head_recv)[0]
    head_recv = sock.recv(head_len)
    head_data = json.loads(head_recv)
    file_name, file_len = head_data['name'], head_data['size']

    # 接受文件信息
    with open(file_name, "wb") as fp:
        recv_len = 0
        while recv_len < file_len:
            this_len = 10240 if file_len - recv_len > 10240 else file_len - recv_len
            data = sock.recv(this_len)
            fp.write(data)
            fp.flush()
            recv_len += this_len
            process = int(recv_len * 100.0 / file_len)
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')

    print("\n下载成功")
```

client.py

```python
try:
    while True:
        head_recv = conn.recv(4)
        head_len = struct.unpack("i", head_recv)[0]

        head_recv = conn.recv(head_len)
        head_data = json.loads(head_recv)

        print("recv", head_data)

        file_cmd = head_data['cmd']

        func = globals().get(file_cmd, None)
        if func is None:
            raise ValueError(f"cmd error: {head_recv}")

        func(conn, head_data)

except Exception as e:
    print("Error:", e)
```

```python
def get(conn, head_data):
    # 获取文件信息
    file_name = head_data['name']
    file_len = os.path.getsize(file_name)
    send_head = {"name": file_name, "size": file_len, "cmd": "get"}
    json_head = json.dumps(send_head).encode('utf8')

    # 发送报文头
    json_head_lens = struct.pack("i", len(json_head))
    conn.send(json_head_lens)
    conn.send(json_head)

    # 发送文件内容
    with open(file_name, "rb") as fp:
        send_len = 0
        while True:
            data = fp.read(10240)
            send_len += len(data)
            process = int(send_len * 100.0 / file_len)
            print(f"\r{'>' * process}{'-' * (100 - process)}{process}%", end='')
            if not data:
                break
            conn.send(data)
    print("上传成功")
```

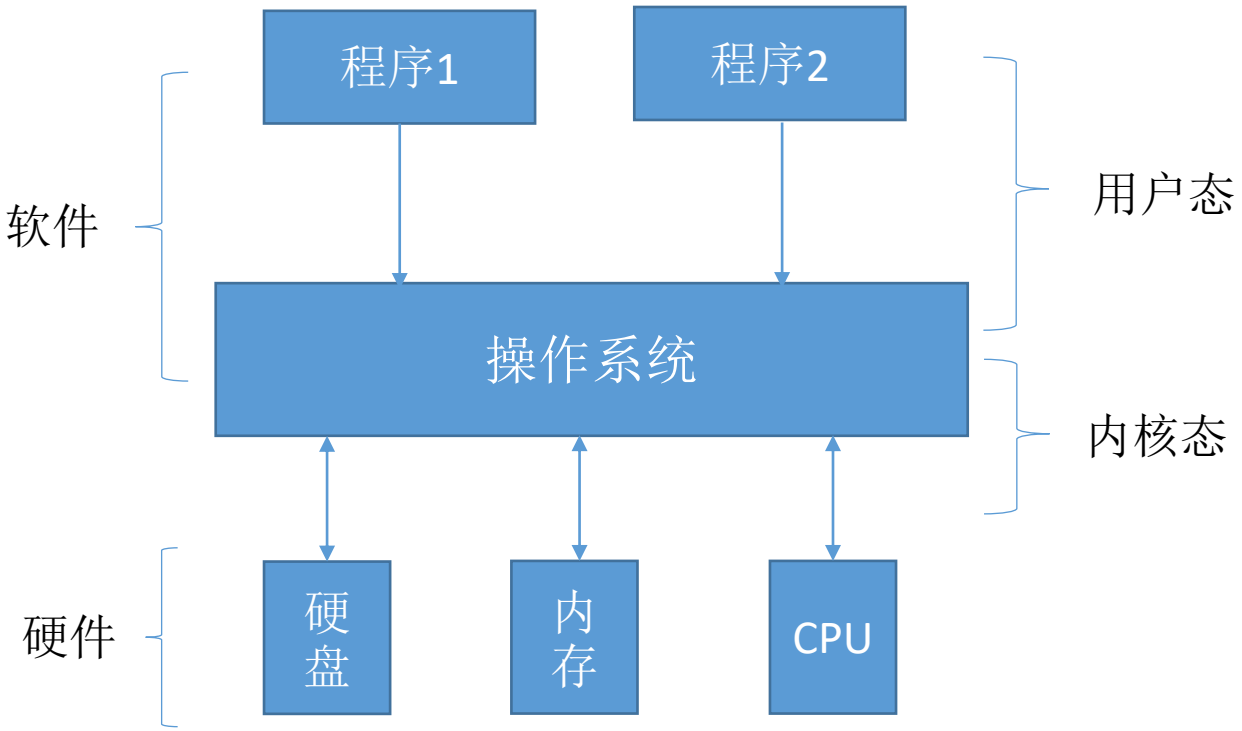
server.py

多线程编程

并发编程的重点：

- 了解操作系统的基本概念及发展历史。
- 理解进程的概念，熟练掌握python进程
- 理解线程的概念，熟练掌握python线程。
- 理解进程与线程的主要区别。

操作系统的基本概念



操作系统位于计算机硬件与应用软件之间，本质也是一个软件。操作系统由操作系统的内核（运行于内核态，管理硬件资源）以及系统调用（运行于用户态，为应用程序员写的应用程序提供系统调用接口）两部分组成。

主要功能：

- 隐藏了丑陋的硬件调用接口，为应用程序员提供调用硬件资源的更好，更简单，更清>晰的模型（系统调用接口）。应用程序员有了这些接口后，就不用再考虑操作硬件的细节，专心开发自己的应用程序即可。
- 将应用程序对硬件资源的竞态请求变得有序化。

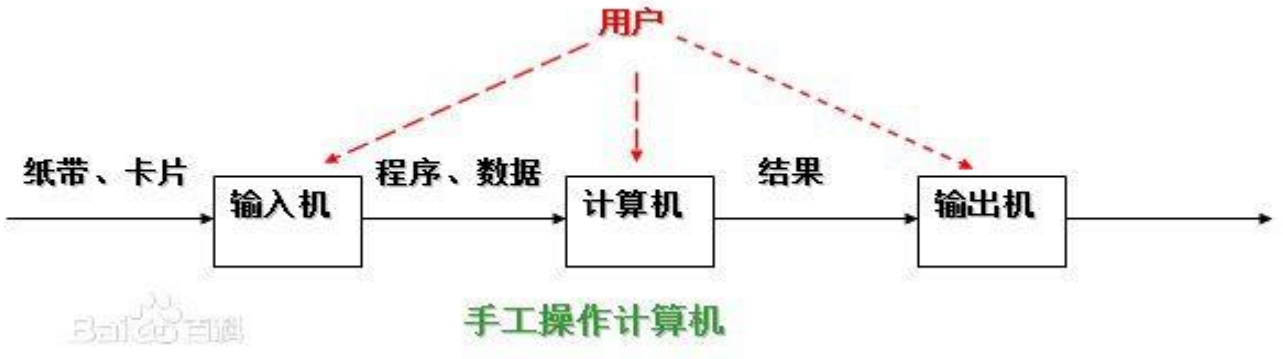
与普通软件的区别：

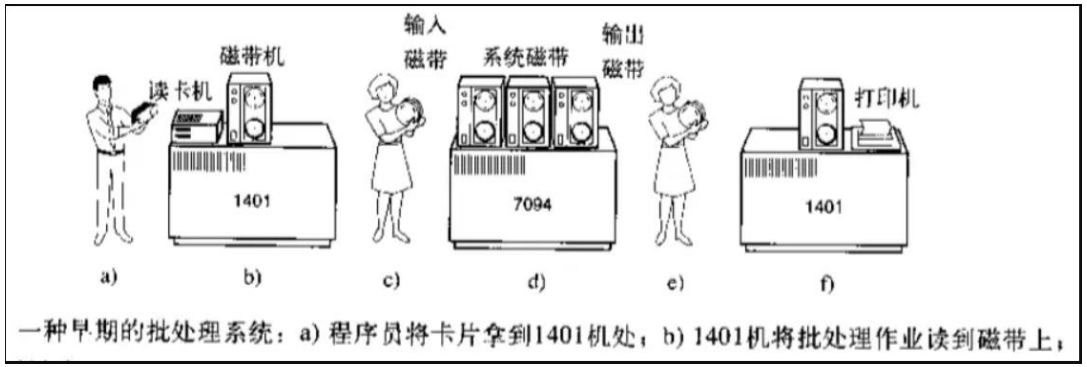
- 直接运行在“裸机”上的最基本软件，其他软件都必须在操作系统的支持下才能运行，由硬件保护，不能被用户修改。
- 协调管理计算机系统中各种独立的硬件，其他软件可不需要顾及到底层每个硬件是如何工作。

操作系统发展历史

1、20世纪50年代中期之前，采用手工操作方式：

- 用户独占全机。不会出现因资源已被其他用户占用而等待的现象，但资源的利用率低。
- CPU 等待手工操作。CPU的利用不充分，手工操作的慢速度和计算机的高速度之间形成了尖锐矛盾。



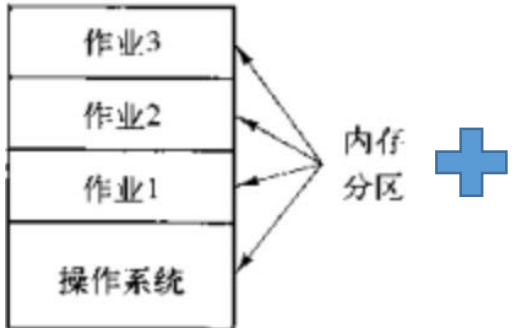


一种早期的批处理系统：a) 程序员将卡片拿到1401机处；b) 1401机将批处理作业读到磁带上，

2、20世纪50年代起，为了CPU使用效率，人们提出了批处理操作系统。用户（生成磁带）+计算机操作员（导出磁带），主要问题：

- 用户程序和监控程序的交替执行，使得一部分内存要交付给监控程序使用，监控程序消耗了一部分时间。
- 每次只能执行一道程序，I/O速度较处理器速度太慢，导致处理器空闲。

3、批处理多道技术，引入多进程系统，通过多路利用解决多进程竞争或者说共享同一个资源（比如cpu）的有序调度问题，多路复用分为空间复用（内存分片）和时间上复用（进程切换）。这里，多道即为多进程。

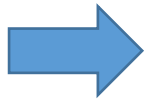


进程之间去争抢cpu的执行权限。这种切换不仅会在一个进程遇到io时进行，一个进程占用cpu时间过长也会被操作系统夺走cpu的执行权限

操作系统发展历史

第3代技术（批处理多道），存在的问题：

- 程序之间的内存必须分割，否则存在安全性和鲁棒稳定性的问题。这种分割需要在硬件层面实现，必须由操作系统控制。。
- 程序员原来独享一段时间的计算机，现在必须被统一规划到一批作业中，等待结果和重新调试的过程都需要等同批次的其他程序都运作完才可以（这极大的影响了程序的开发效率，无法及时调试程序）



分时操作系统：

- 多个联机终端
- 多道技术

分时操作系统将系统处理器时间与内存空间按一定的时间间隔，轮流地切换给各终端用户的程序使用。由于时间间隔很短，每个用户的感觉就像他独占计算机一样。分时操作系统的特点是可有效增加资源的使用率。

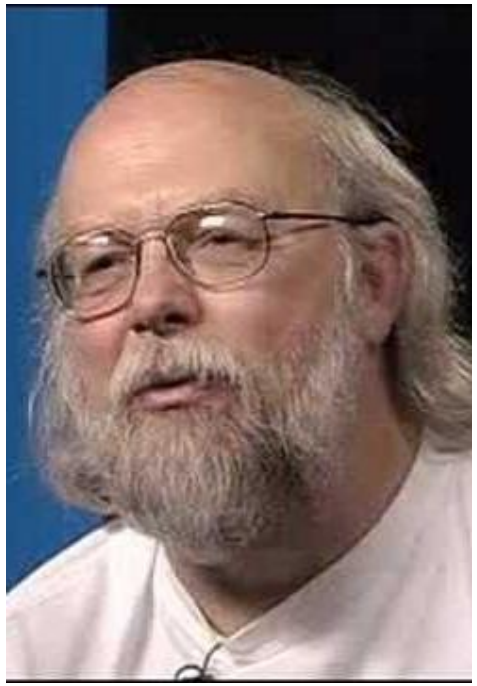
- 1961年，CTSS兼容分时系统， IBM 709（最后一款电子管机）→ IBM 7094
- 1965年，贝尔实验室和通用电气开发MULTICS，目标是满足波士顿地区所有用户需求，最终失败。
- 1969年，贝尔实验室Ken Thompson(UNIX之父) 简化了MULTIC，形成UNIX系统，并衍生相当多版本（IBM-AIX，HP-UX, SUN-Soliaris，APPLE-MAC）
- 1985年，美国微软公司研发的WINDOWS操作系统，多用户多任务操作系统。
- 1991年，芬兰学生Linus（Linux之父）在minix(UNIX变种)的基础上，编写了Linux，开源的成功典型。

操作系统发展历史

Ken Thompson
UNIX之父
（1943-）

Linus
Linux之父
（1969年-）

Dennis Ritchie
C语言发明者
（1941-2011）



James Gosling
JAVA之父
（1955-）

Guido Van Rossum
Python之父
（1956-）

论程序员职业选择与头发之间的关系。。。。

操作系统知识小结

即使可以利用的cpu只有一个（早期的计算机确实如此），也能保证支持（伪）并发的能力。将一个单独的cpu变成多个虚拟的cpu（多道技术：时间多路复用和空间多路复用+硬件上支持隔离），没有进程的抽象，现代计算机将不复存在。

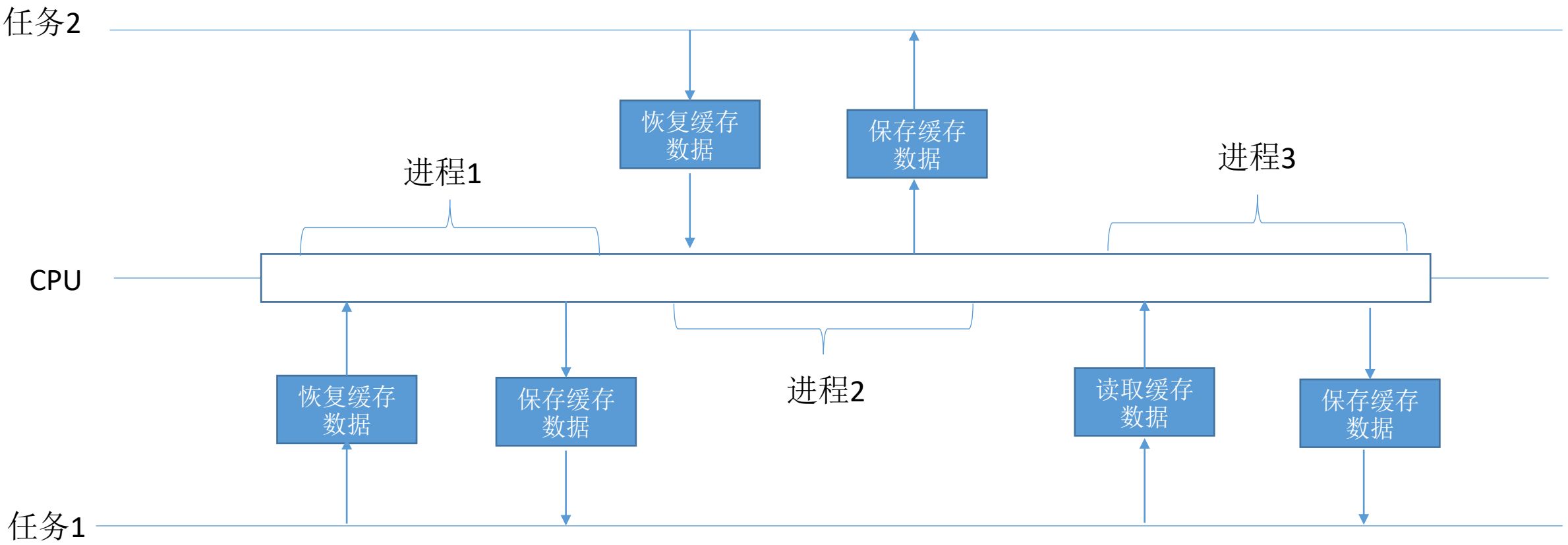
- 操作系统的作用：
  - ✓ 隐藏丑陋复杂的硬件接口，提供良好的抽象接口
  - ✓ 管理、调度进程，并且将多个进程对硬件的竞争变得有序
- 多道技术：
  - ✓ 产生背景：针对单核，实现并发。
  - ✓ 现在的主机一般是多核，那么每个核都会利用多道技术。
  - ✓ 空间上的复用：如内存中同时有多道程序
  - ✓ 时间上的复用：复用一个cpu的时间片，遇到io切，占用cpu时间过长也切，核心在于切之前将进程的状态保存下来，这样才能保证下次切换回来时，能基于上次切走的位置继续运行

并发编程-进程

- 进程与程序的区别：
  - ✓ 程序：一堆代码的集合，进程是操作系统调用程序执行的运行过程。
  - ✓ 进程：正在进行的一个过程或者说一个任务。而负责执行任务则是cpu。
  - ✓ 需要强调的是：同一个程序执行两次，那也是两个进程，比如打开暴风影音，虽然都是同一个软件，但是一个可以播放本地视频，一个可以播放网络视频。
- 并发与并行的区别：
  - ✓ 并发：伪并行，看起来是同时运行。单个cpu+多道技术就可以实现并发，实际上一个cpu在同一时刻只能执行一个任务。
  - ✓ 并行：同时运行，只有具备多个cpu才能实现并行，
  - ✓ 无论是并行还是并发，在用户看来都是‘同时’运行的，不管是进程还是线程，都只是一个任务而已，真是干活的是cpu，cpu来做这些任务，而一个cpu同一时刻只能执行一个任务。

思考：通过并行计算，一定可以提高计算速度吗？

并发编程-实现原理



进程并发：操作系统维护一张表格，即进程表（process table），每个进程占用一个进程表项（或进程控制块），存放进程状态信息：程序计数器、堆栈指针、内存分配状况、所有打开文件的状态，以及其他必须保存的信息，当状态切出时保存进程实时状态；当状态切入时再加载进程控制块，就像从未被中断过一样。

并发编程-进程

进程的创建：由一个已经存在的进程执行了系统调用命令而创建的新进程：

- 操作系统初始化
- 一个进程在运行过程中开启了子进程（如：os.system,subprocess.Popen等）
- 通过用户的交互式请求创建一个新进程：
  - ✓ 双击调用。
  - ✓ cmd窗口命令调用。

不同操作系统的进程调用方法：

- UNIX：fork。
- windows：CreateProcess，CreateProcess。

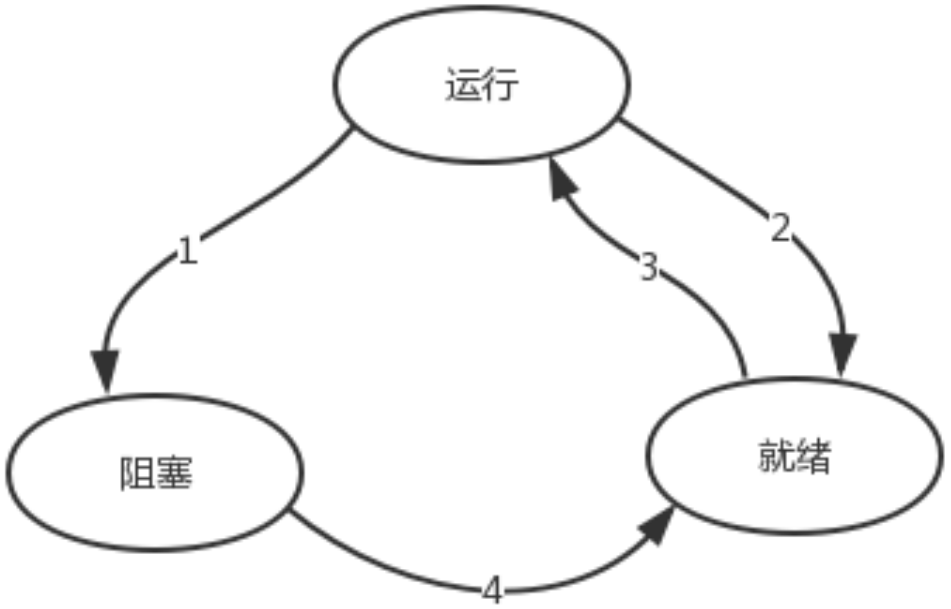
进程创建后，父进程和子进程有各自不同的地址空间（多道技术要求物理层面实现进程之间内存的隔离），任何一个进程的在其地址空间中的修改都不会影响到另外一个进程。

并发编程-进程

进程的中止：

- 正常退出（如用户点击交互式页面的叉号，或程序执行完毕后退出，linux：exit，windows：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）

进程的状态：



1、IO阻塞或调用sleep()

4、阻塞结束，转入就绪

3、操作系统切就绪态进程到CPU

2、进程用时过长而被操作系统切换

并发编程- multiprocessing模块-Process方法

multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数）。

需要再次强调的一点是：进程没有任何共享状态，进程修改的数据，改动仅限于该进程内。

```python
import time
import random
from multiprocessing import Process
def Sleep(name):
    print('%s %s 睡觉中。。。。。' % (__name__,name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床' % name)

if __name__ == '__main__':
    # 实例化得到四个对象
    p1 = Process(target=Sleep, args=('p1',))  # 必须加,号
    p2 = Process(target=Sleep, args=('p2',))
    p3 = Process(target=Sleep, args=('p3',))
    p4 = Process(target=Sleep, args=('p4',))

    # 调用对象下的方法，开启四个进程
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    print('主')
```

```python
import time
import random
from multiprocessing import Process

val = 0
def Sleep(name):
    global val
    val = val + 1
    print('%s %s 睡觉中。。。。' % (__name__,name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床 %s' % (name, val))

if __name__ == '__main__':
    # 实例化得到四个对象
    p1 = Process(target=Sleep, args=('p1',))
    p2 = Process(target=Sleep, args=('p2',))
    p3 = Process(target=Sleep, args=('p3',))
    li = [p1, p2, p3]

    for p in li:
        p.start()
    print('主')

    for p in li:
        p.join()
    print('end 主')
```

并发编程- multiprocessing模块

1、思考开启进程的方式一和方式二各开启了几个进程？

2、进程之间的内存空间是共享的还是隔离的？下述代码的执行结果是什么？

```
from multiprocessing import Process
n=100 #在windows系统中应该把全局变量定义在if __name__ == '__main__'之上就可以了
def work():
    global n
    n=0
    print('子进程内: ',n)
if __name__ == '__main__':
    p=Process(target=work)
    p.start()
    print('主进程内: ',n)
```

课堂练习（2） 用多进程去封装模拟FTP

```python
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        try:
            while True:
                head_recv = conn.recv(4)
                head_len = struct.unpack("i", head_recv)[0]
                head_recv = conn.recv(head_len)
                head_data = json.loads(head_recv)
                print("recv", head_data)
                file_cmd = head_data['cmd']
                func = globals().get(file_cmd, None)
                if func is None:
                    raise ValueError(f"cmd error: {head_recv}")
                func(conn, head_data)

        except Exception as e:
            print("Error:", e)

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

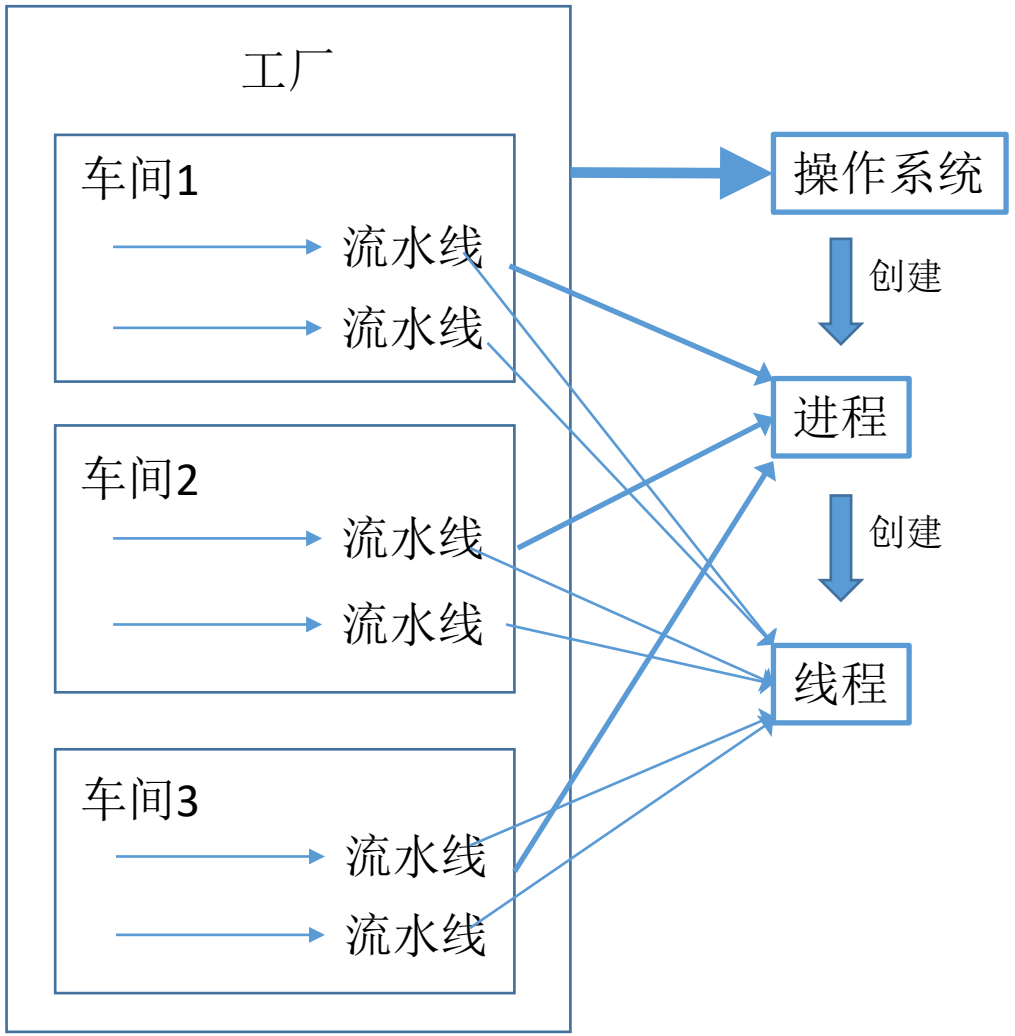
```python
from multiprocessing import Process
def conn_thread(conn):
    print("new thread", conn)
    try:
        while True:
            head_recv = conn.recv(4)
            head_len = struct.unpack("i", head_recv)[0]
            head_recv = conn.recv(head_len)
            head_data = json.loads(head_recv)
            print("recv", head_data)
            file_cmd = head_data['cmd']
            func = globals().get(file_cmd, None)
            if func is None:
                raise ValueError(f"cmd error: {head_recv}")
            func(conn, head_data)

    except Exception as e:
        print("Error:", e)

def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        process = Process(target=conn_thread, args=(conn,))
        process.start()

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

并发编程-线程



线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

每个进程都有自己的地址空间，在高并发场景下，为每一个请求都创建一个进程显然行不通（系统开销大响应用户请求效率低），为此才引入线程：

- 线程是执行单位，不是资源单位，申请线程不需要向操作系统额外申请地址空间（启动快）。
- 每一个进程至少有一个主线程，单线程进程宏观来看也是线性执行过程，微观上只有单一的执行过程。多线程进程宏观是线性的，微观上多个执行操作。

并发编程-线程与进程的区别：

	进程	线程
地址空间	独立	共享进程内地址空间
资源拥有	独立	共享本进程资源，如内存、I/O、cpu等
健壮性	独立	线程崩溃将整个进程故障
生成与切换速度	消耗资源大，效率低	消耗资源小，效率高
执行过程	操作系统调用进程	进程调用线程
处理器调度	资源单位	执行单位
并发执行	可以	可以

并发编程-threading模块

用法与multiprocessing模块相同

```
import time
import random
from multiprocessing import Process
from threading import  Thread
def Sleep(name):
    print('%s %s 睡觉中。。。。' % (__name__,name))
    time.sleep(random.randrange(1, 5))
    print('%s 起床' % name)

if __name__ == '__main__':
# 实例化得到四个对象

    p1 = Thread(target=Sleep, args=('p1',))
    p2 = Thread(target=Sleep, args=('p2',))
    p3 = Thread(target=Sleep, args=('p3',))
    p4 = Thread(target=Sleep, args=('p4',))
# 调用对象下的方法，开启四个进程

    p1.start()
    p2.start()
    p3.start()
    p4.start()
    print('主')
```

```
import time
import random
from multiprocessing import Process
from threading import Thread
class Sleep(Thread):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        print('%s %s 睡觉中。。。。' % (__name__, self.name))
        time.sleep(random.randrange(1, 5))
        print('%s 起床' % self.name)

# if __name__ == '__main__':
# 实例化得到四个对象
p1 = Sleep('p1')
p2 = Sleep('p2')
p3 = Sleep('p3')
p4 = Sleep('p4')
p_list = [p1, p2, p3, p4]
for p in p_list:
    p.start()
print('主')
```

并发编程-threading模块-其他参数

- Thread实例对象的方法

  # isAlive(): 返回线程是否活动的。

  # getName(): 返回线程名。

  # setName(): 设置线程名。

- threading模块提供的一些方法：

  # threading.currentThread(): 返回当前的线程变量。

  # threading.enumerate(): 返回一个包含正在运行的线程的list，不包括启动前和终止后的线程。

  # threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

```
import threading
import time
def work():
    time.sleep(3)
    print(threading.current_thread().getName())
```

```
t = threading.Thread(target=work)
t.start()
print(threading.current_thread().getName())
print(threading.current_thread())  # 主线程
print(threading.enumerate())  # 连同主线程在内有两个运行的线程
print(threading.active_count())
print('主线程/主进程')
```

并发编程-threading模块-其他参数

主系统等待子线程结束

```
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=('egon',))
    t.start()
    t.join()
    print('主线程')
    print(t.is_alive())
```

主系统不等待子线程结束

```
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=('egon',))
    t.daemon = True
    t.start()
    # t.join()
    print('主线程')
    print(t.is_alive())
```

思考执行结果？

```
from threading import Thread
import time
def foo():
    print(123)
    time.sleep(3)
    print("end123")
def bar():
    print(456)
    time.sleep(1)
    print("end456")
if __name__ == '__main__':
    t1=Thread(target=foo)
    t2=Thread(target=bar)
    t1.daemon=True
    t1.start()
    t2.start()
    print("main-------")
```

课堂练习（3） 用多线程去封装模拟FTP

```python
def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        try:
            while True:
                head_recv = conn.recv(4)
                head_len = struct.unpack("i", head_recv)[0]
                head_recv = conn.recv(head_len)
                head_data = json.loads(head_recv)
                print("recv", head_data)
                file_cmd = head_data['cmd']
                func = globals().get(file_cmd, None)
                if func is None:
                    raise ValueError(f"cmd error: {head_recv}")
                func(conn, head_data)

        except Exception as e:
            print("Error:", e)

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

```python
from threading import Thread
def conn_thread(conn):
    print("new thread", conn)
    try:
        while True:
            head_recv = conn.recv(4)
            head_len = struct.unpack("i", head_recv)[0]
            head_recv = conn.recv(head_len)
            head_data = json.loads(head_recv)
            print("recv", head_data)
            file_cmd = head_data['cmd']
            func = globals().get(file_cmd, None)
            if func is None:
                raise ValueError(f"cmd error: {head_recv}")
            func(conn, head_data)

    except Exception as e:
        print("Error:", e)

def main():
    sock = socket.socket()
    sock.bind(("127.0.0.1", 8889))
    sock.listen(10)
    while True:
        conn, status = sock.accept()
        process = Thread(target=conn_thread, args=(conn,))
        process.start()

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