银行家算法实验报告

课程名称： 操作系统 **指导老师：** _____ **完成时间：** 2022.11.28
班　级　： _____ **学　　号　：** 2 **姓　　名** _____

一、实验内容

用高级编程语言实现银行家算法对进程资源调度的处理和死锁的判断。

二、实验目的

1、了解计算机为进程分配所需资源的过程；

2、认识死锁产生的原因及其解决办法，了解银行家算法的原理；

3、用高级编程语言实现模拟动态资源分配的银行家算法，并检查死锁是否产生，确保系统的安全性。

三、实验原理

1、检验系统安全性算法

检查系统的安全性，一般以系统是否会在当前的状态下发生死锁作为判断依据。在检查系统安全性的算法当中，应当已知：

（1）该计算机系统可用的资源数组 Available，其中 Available[i]代表系统中第 i 个资源的数量；

（2）该计算机系统中各进程所需的资源数二维数组 Need,其中 Need[i]表示第 i 个进程所需的资源数量数组，Need[i][j]表示第 i 个进程所需第 j 个资源的数量；

（3）该计算机系统中已经为各进程所分配的资源数量二维数组 Allocation，其中 Allocation[i]表示第 i 个进程已分配的资源数量数组，Allocation[i][j]表示第 i 个进程已分配的第 j 个资源的数量；

此外，应设置一个数组 Finish 用以记录每个进程的完成状况，Finish[i]表示第 i 个进程是否已经完成。

检验系统的安全性时，应在所有进程当中找到一个能够使用当前数量的系统资源完成运行过程的进程（即 Need 中的所需数量不大于 Available

中已有的资源数量），并将其在 Finish 数组中的对应位置更改为已完成状态，再将其 Allocation 中已分配的资源回收到系统当中，系统的 Available 数组作相应的增加。如果无法找到一个能够使用当前数量的系统资源完成运行过程的进程，那么死锁发生，系统不安全。

2、动态资源分配的银行家算法

银行家算法是用来评估是否可以给某一进程分配指定数量的系统资源的一种算法，其核心思想是评估在为指定进程分配指定数量的系统资源后，系统是否会发生死锁。

在执行银行家算法的时候，应当具有检查系统安全性所需要的相关信息，此外还需要：

（1） work 数组，检查安全性的时候以此数组作为 available 数组；

（2） Request 数组，表示要为某一应用程序请求的资源的数量，如 Request[i]表示该应用程序所请求的第 i 种资源的数量。

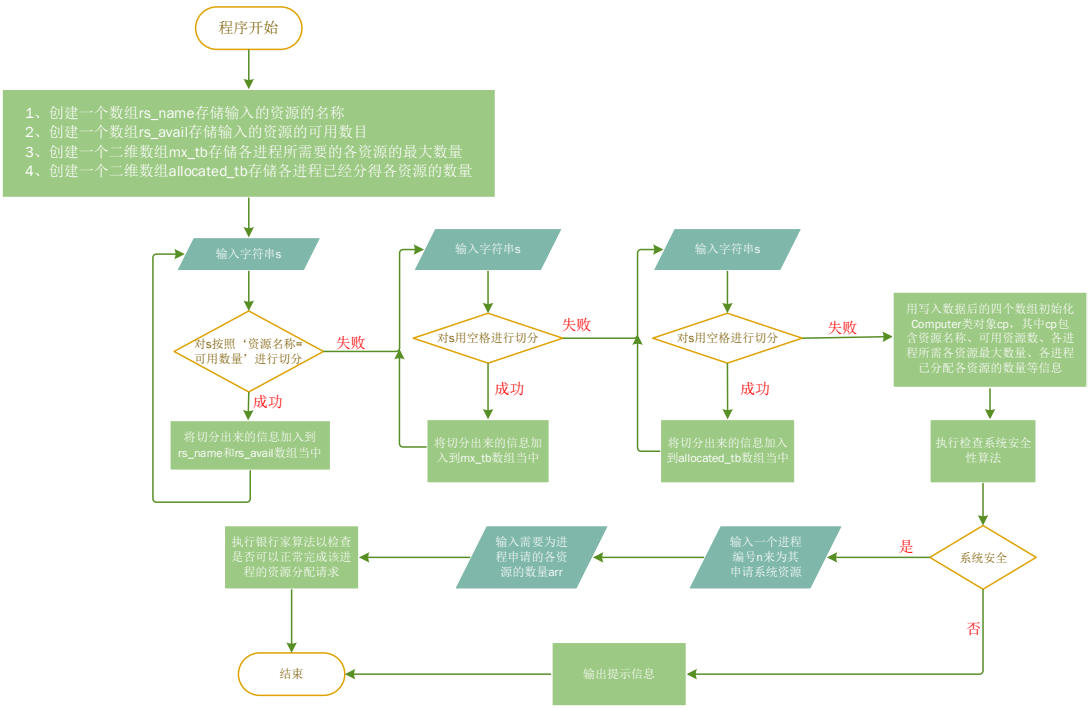
银行家算法的流程为，先对所请求的资源进行预分配，再按照预分配后的系统资源数量情况和程序所需资源的情况进行安全性检查，若出现死锁则证明分配后系统不安全，当不进行资源的分配，将程序状态和系统状态返回到分配前。

在执行银行家算法的时候，需要对输入的要请求的资源数量进行检查，若出现已分配加请求值大于声明的最大需求数量则认为输入有误，不予分配请求的资源；若输入的请求需要分配的资源的数量大于当前系统所具有的资源的数量，则系统不予分配请求的资源的数量。

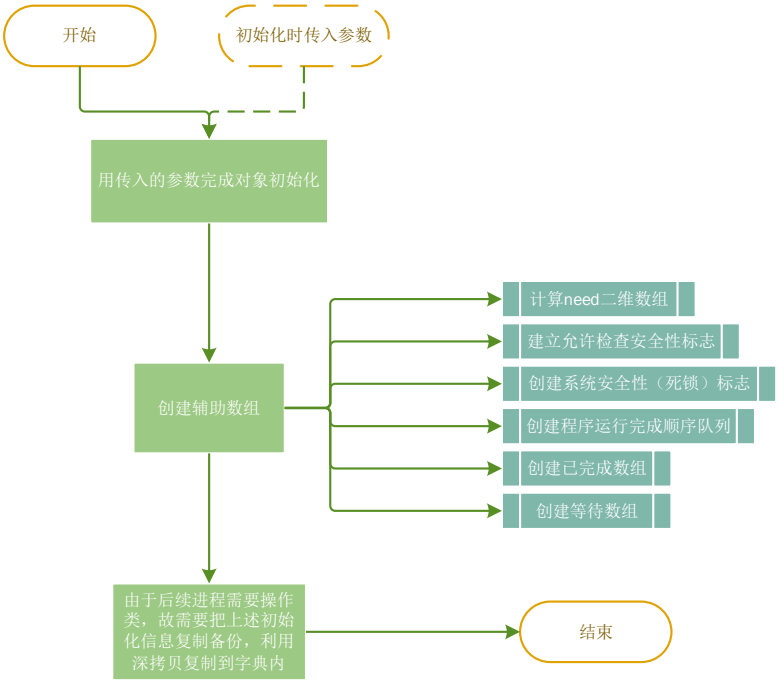
四、实验报告

1、算法流程图

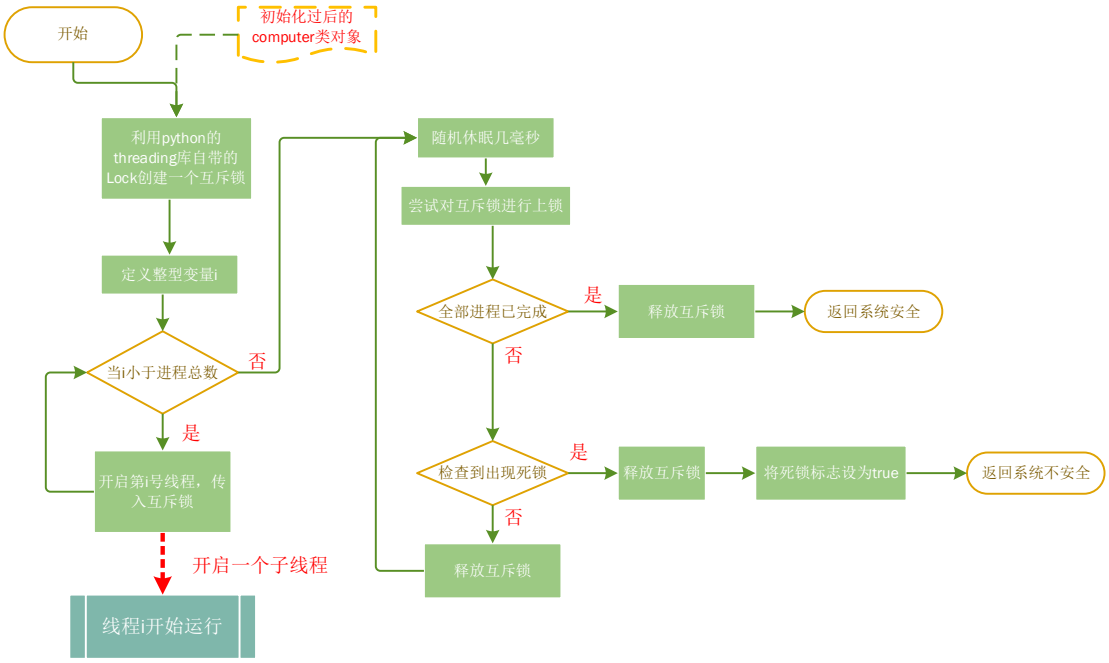
（1）、程序整体运行过程



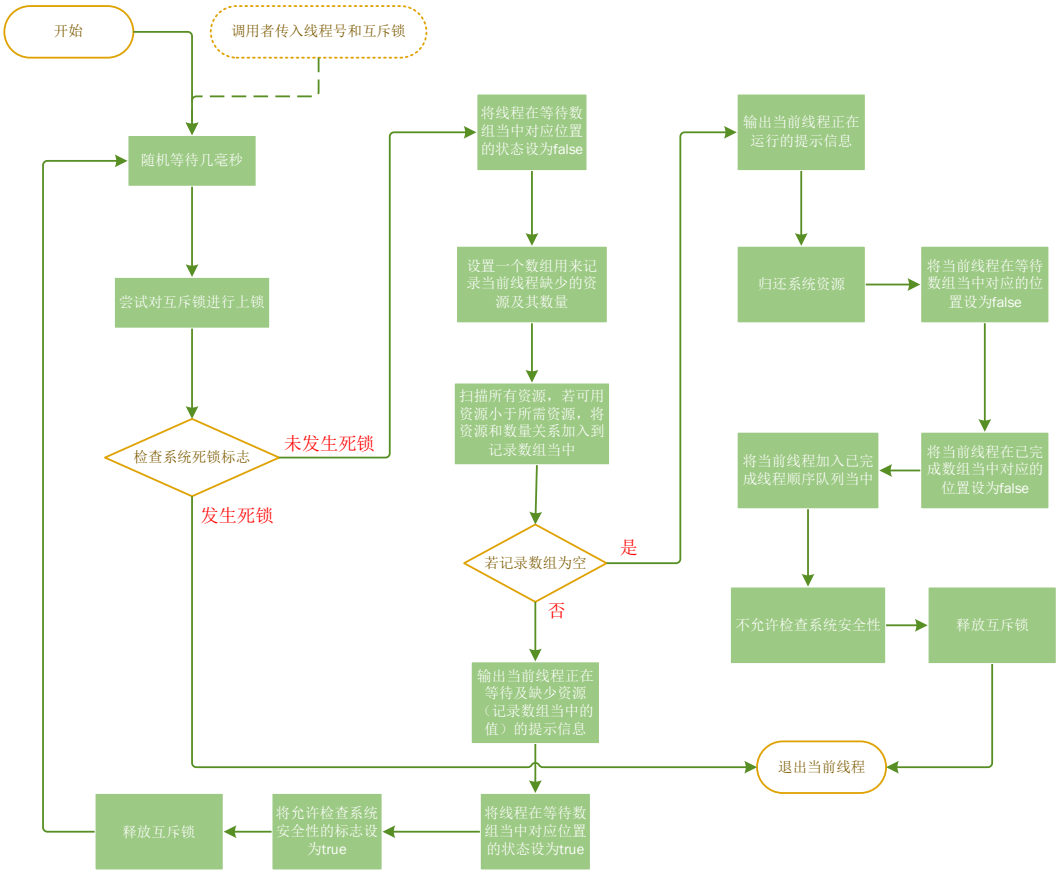
（2）、初始化 computer 对象



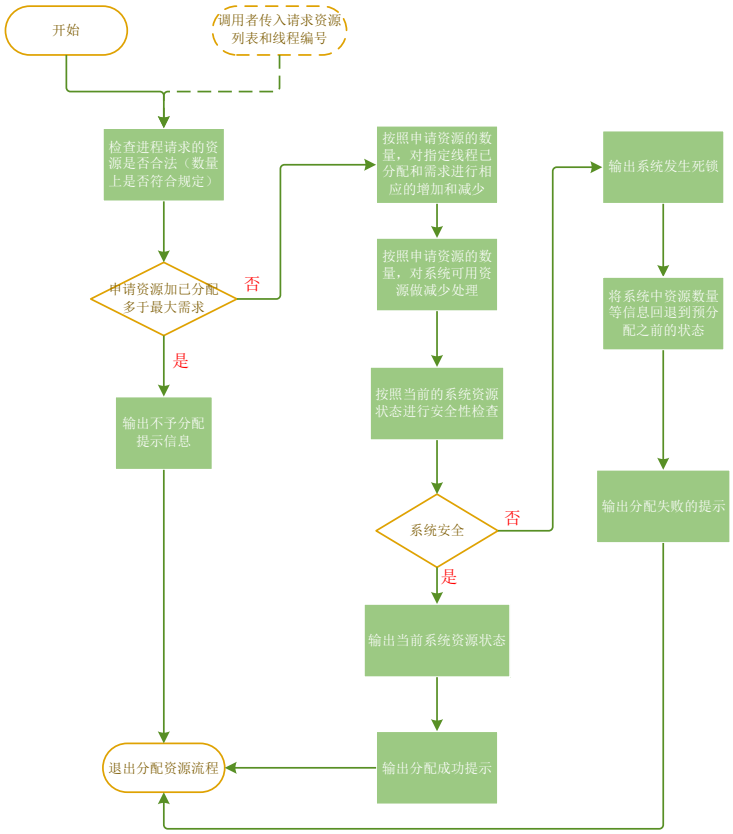
（3）、检查系统安全性



（4）、子线程的运行



（5）、为线程申请资源



2、程序中使用到的数据结构

类 computer

```
class Computer:
    def __init__(self, resources: list, available: list, maximum: typing.List[list],
allocated: typing.List[list]):
            self.resource_total_num: int = resources.__len__()
            self.resources = resources
            self.available = available
            self.progress_total_num = maximum.__len__()
            self.finished: typing.List[bool] = [False for _ in
range(self.progress_total_num)]
            self.waiting: typing.List[bool] = [False for _ in
range(self.progress_total_num)]
            self.solve_queue: typing.List[int] = []
            self.deadlock: bool = False
            self.allow_check_sys_safe: bool = False
...
            self.maximum = maximum
            self.allocated = allocated
            self.need: typing.List[typing.List[int]] = [[0 for __ in
range(self.resource_total_num)] for _ in range(self.progress_total_num)]
            for i in range(self.progress_total_num):
```

```
                for j in range(self.resource_total_num):
                    self.need[i][j] = self.maximum[i][j] - self.allocated[i][j]
            self.bk = {
                'need': copy.deepcopy(self.need),
                'available': copy.deepcopy(self.available),
                'allocate': copy.deepcopy(self.allocated)

    def progress(self, progress_index: int, lock: threading.Lock):
        ...

    def check_died_lock(self) -> bool:
        ...

    def check_progress_finish(self) -> bool:
        ...
    def check_system_is_not_safety(self) -> bool:
        ...
    def apply_resource(self, progress_index, apply_array):
        ...
```

如所给出的部分代码，computer 类使用传入的资源数组、最大需求数组、可用资源数数组和已分配资源数组对 computer 对象进行初始化。

类中各成员变量：

（1）、由传入的数据可以简单的分析得到进程的数量和资源的数量；同时，设置了已完成数组来记录每个进程是否已完成，其初始化为长度为进程数量的数组，值全为 false；

（2）、设置等待数组记录每个进程是否正在等待系统资源，其初始化为长度为进程数量的数组，值全为 false；

（3）、设置完成队列，为一个空列表，将进程按照完成的先后顺序添加到其中；

（4）、设置系统已经发生死锁的标志，当主线程检查到已经发生死锁后将其修改为 true，子线程每一次运行检查到其值为 true 后便停止执行；

（5）、设置允许检查系统安全性的标志，当有子线程处于等待时将其修改为 true，这时主线程可以检查系统安全性，若有子线程刚好完成，将其修改为 false，不允许检查系统的安全性，因为已经有进程完成了，即便此时检查系统的安全性也一定是安全的，即：有子线程完成意味着一

定没有死锁，有子线程等待则可能发生死锁，需要检查；

（6）、通过传入的已分配数组和最大需求数组可以计算出各进程对各资源的需求，作为二维数组存放到其中；

（7）、为了便于后面还原系统的状态，在初始化的时候即可将信息储存到一个字典当中。

类中各成员函数：

（1）、__init__()

该函数为 python 中类的初始化函数，成员变量的初始化在其中完成；

（2）、progress()

该函数的运行对应着一个线程，其中需要传递要运行的线程的编号和与其他线程共享的互斥锁；函数通过线程编号操作 computer 类的对象当中对应的数据；

（3）、check_died_lock()

检查死锁是否发生的函数，检查死锁的标准为：是否所有的未完成的线程都处于等待状态。执行时扫描一遍等待数组，若全部等待数组中为 true 且完成数组中为 false 则发生了死锁，返回 true，否则返回 false；

（4）、check_progress_finish()

检查是否所有的线程都已完成，只需统计完成数组当中的 false 的个数是否为 0 即可；

（5）、check_system_is_not_safety()

检查当前的系统状态是不是处于不安全状态。整体思路为：开启全部的子线程，让子线程一起运行，父线程开始统计所有的子线程是否完成，若都完成则为安全，返回 false；若没有完成则检查死锁，若发生死锁则不安全，返回 true；若没有完成且也没有死锁则父线程等待几十毫秒，再次检查安全性；

（6）、apply_resource()

为子线程申请一定数量的资源。若申请的资源数量加已分配大于最大需求，则申请无效，不予分配；若申请有效则进行预分配，按照申请的数量为相应的线程已分配增加相应的资源数，需求减少相应的资源数，系统可用资源减少相应的资源数。按照预分配后系统的状态检查系统的安全性，若不安全则分配无效，按照预分配前对相关资源数量的备份进行回退；若安全，则分配有效，输出分配后的系统资源信息。

3、程序源代码

在实现银行家算法的过程当中，使用了 python 的 threading 库实现了多线程，用不同的线程模拟每一个进程，通过主线程对子线程的不断检测来判断系统的安全性，其中使用到了诸如互斥信号量等进程/线程同步方法。

```
import threading
import time
import typing
import random
import copy


class Computer:
    # 初始化 computer 类
    def __init__(self, resources: list, available: list, maximum:
typing.List[list], allocated: typing.List[list]):
        try:
            # 由资源列表的长度得到资源数量
            self.resource_total_num: int = resources.__len__()
            self.resources = resources
            # 系统可用资源数量的长度必须等于资源数，否则抛出异常
            if available.__len__() != self.resource_total_num:
                raise Exception("Available array does not match resources array!")
            else:
                self.available = available
            # 通过最大需求矩阵得到进程的数量
            self.progress_total_num = maximum.__len__()
            # 创建完成和等待记录的数组，成员为 bool 型，长度为进程的数量
            self.finished: typing.List[bool] = [False for _ in
range(self.progress_total_num)]
            self.waiting: typing.List[bool] = [False for _ in
range(self.progress_total_num)]
            # 完成顺序队列，方便后面输出完成的顺序
            self.solve_queue: typing.List[int] = []
```

```
            # 死锁标志
            self.deadlock: bool = False
            # 运行检查死锁标志
            self.allow_check_sys_safe: bool = False

            # 最大需求矩阵的宽度需要都等于资源数，否则抛出异常
            if maximum[0].__len__() != self.resource_total_num:
                raise Exception("Maximum array does not match resources array!")
            else:
                self.maximum = maximum
            # 已分配矩阵的长度需要等于进程总数，否则抛出异常
            if allocated.__len__() != self.progress_total_num:
                raise Exception("Allocated array does not match progress num!")
            else:
                # 已分配矩阵的宽度需要等于资源数量，否则抛出异常
                if allocated[0].__len__() != self.resource_total_num:
                    raise Exception("Allocated array does not match resources
num!")
                else:
                    self.allocated = allocated
            # 计算需求矩阵
            self.need: typing.List[typing.List[int]] = [[0 for __ in
range(self.resource_total_num)] for _ in

range(self.progress_total_num)]
            for i in range(self.progress_total_num):
                for j in range(self.resource_total_num):
                    self.need[i][j] = self.maximum[i][j] - self.allocated[i][j]
            # 备份当前初始化值到字典
            self.bk = {
                'need': copy.deepcopy(self.need),
                'available': copy.deepcopy(self.available),
                'allocate': copy.deepcopy(self.allocated)
            }
        # 捕捉上述异常，作为新的异常抛出到顶层
        except IndexError as e:
            raise IndexError("Must be maximum or allocated array out of range", e)
        except Exception as e:
            print(e)

    # 方法：子线程运行
    def progress(self, progress_index: int, lock: threading.Lock):
        # 一直循环
        while True:
```

```
                # 先随机休眠几个毫秒，为别的线程提供上锁的机会
                time.sleep(random.randint(10, 1000)/100000)
                # 尝试为互斥锁上锁
                lock.acquire()
                # 如果系统已经死锁，则释放锁，退出
                # 子线程不知道系统已经死锁，主线程检查到死锁后修改 self.deadlock 为
true，这样才能使已经死锁的子线程退出
                if self.deadlock:
                    lock.release()
                    return
                # 已经开始运行，把对应位置的等待值设为假
                self.waiting[progress_index] = False
                # 设置列表记录缺少的资源
                lack_resources = []
                # 扫描所有资源
                for i in range(self.resource_total_num):
                    if self.need[progress_index][i] > self.available[i]:
                        # 若缺少相关资源，则把相关信息写入到上表
                        lack_resources.append([self.resources[i], f"need =
{self.need[progress_index][i]}",
                                               f"available = {self.available[i]}"])
                # 如果缺少的资源列表非空
                # 非空则证明有资源数量不够，那么这个线程就需要继续等待
                if lack_resources:
                    print(f"Progress {progress_index} is waiting for {'&'.join(str(i)
for i in lack_resources)} now")
                    self.waiting[progress_index] = True
                    # 发生了继续等待，系统有可能死锁，运行主线程进行安全性检查
                    self.allow_check_sys_safe = True
                # 都已经等待了，释放锁，开始下一次循环
                if self.waiting[progress_index]:
                    lock.release()
                    continue
                # 到这里的一定是能够有足够的资源运行完成的线程
                # 输出运行提示信息
                print(f"Progress {progress_index} is running now")
                # 顺序完成队列加入本线程
                self.solve_queue.append(progress_index)
                # 已完成，则不再等待
                self.waiting[progress_index] = False
                self.finished[progress_index] = True
                # 将已分配的资源退还系统
                for i in range(self.resource_total_num):
                    self.available[i] += self.allocated[progress_index][i]
```

```
            print(f"Progress {progress_index} finished,
available={self.available}")
            # 这里有线程运行结束了，无需检查系统安全性，此时一定是没有死锁的
            self.allow_check_sys_safe = False
            # 释放锁，退出循环，线程结束
            lock.release()
            break

    # 方法：检查死锁的发生
    def check_died_lock(self) -> bool:
        # 若所有的没有完成的线程都在等待，则一定死锁
        for i in range(self.progress_total_num):
            if self.finished[i]:
                continue
            flag = True
            for j in range(self.resource_total_num):
                if self.need[i][j] > self.available[j]:
                    flag = False
            if flag:
                return flag
        return False

    # 方法：检查所有的线程都已完成
    def check_progress_finish(self) -> bool:
        # 已完成数组中 false 的个数为 0
        return self.finished.count(False) == 0

    # 方法：检查当前系统是否不安全
    def check_system_is_not_safety(self) -> bool:
        # 从 threading 库创建一个互斥锁
        lock = threading.Lock()
        # 开启全部的子线程
        for i in range(self.progress_total_num):
            # 参数传入：线程号，同一个互斥锁
            process = threading.Thread(target=self.progress, args=(i, lock, ))
            process.start()
        # 一直循环
        while True:
            # 主线程随机休眠一定时间（略大于子线程的休眠时间）
            time.sleep(random.randint(130, 160)/10000)
            # 尝试上锁
            lock.acquire()
            # 如果检查到全部已完成，那么系统安全，返回 false
            if self.check_progress_finish():
```

```
                print(f"solve order: {' --> '.join(str(_) for _ in
self.solve_queue)}")
                lock.release()
                return False
            # 如果检查到死锁，那么系统不安全，返回 true
            if self.allow_check_sys_safe:
                if not self.check_died_lock():
                    self.deadlock = True
                    lock.release()
                    return True
            lock.release()

    # 方法：重置信息
    def reset_bk_inf(self):
        # 将备份到字典 bk 当中的信息用深拷贝的方法重新赋值给相关类成员变量
        self.available = copy.deepcopy(self.bk['available'])
        self.need = copy.deepcopy(self.bk['need'])
        self.allocated = copy.deepcopy(self.bk['allocate'])
        self.waiting = [False for _ in range(self.progress_total_num)]
        self.finished = [False for _ in range(self.progress_total_num)]
        self.solve_queue = []
        self.allow_check_sys_safe = False
        self.deadlock = False

    # 方法：为指定的进程申请指定的数量的资源
    def apply_resource(self, progress_index, apply_array):
        # 先暂存相关信息
        bak_available = copy.deepcopy(self.available)
        bak_allocated = copy.deepcopy(self.allocated)
        bak_need = copy.deepcopy(self.need)
        # 检查申请的资源数是否合法
        flag = True
        for i in range(self.resource_total_num):
            # 若需求的资源数小于申请的资源数，则不可分配，退出方法
            if self.need[progress_index][i] < apply_array[i]:
                flag = False
                break
        if not flag:
            print(f"Allocate resource to progress {progress_index} failed, apply
more than need!")
            return
        # 进行预分配，将指定的资源数量从系统可用、已分配、需求当中加上或者减去
        for i in range(self.resource_total_num):
            self.allocated[progress_index][i] += apply_array[i]
```

```
                self.need[progress_index][i] -= apply_array[i]
                self.available[i] -= apply_array[i]
            print("-----try to allocate and check died lock-----")
            print("after prepare allocate, the system situation: ")
            print(f"available array={self.available}\nallocated
table={self.allocated}\nneed table={self.need}")
            # 用预分配后的系统状态进行安全性检查
            if self.check_system_is_not_safety():
                # 若不安全，输出分配失败的提示信息，再将备份的信息进行回退
                print("Allocate failed: died lock will occur")
                print("return to last state")
                self.available = bak_available
                self.need = bak_need
                self.allocated = bak_allocated
            else:
                # 若安全，则分配成功
                print("Allocate success")
                print(f"formal allocate resource{apply_array} to progress
No.{progress_index}")
                self.available = bak_available
                self.allocated = bak_allocated
                self.need = bak_need
                for i in range(self.resource_total_num):
                    self.available[i] -= apply_array[i]
                    self.allocated[progress_index][i] += apply_array[i]
                    self.need[progress_index][i] -= apply_array[i]
            # 等待一点时间
            # 原因：让没有跑完的子线程运行完毕，以免修改了后面的数据
            time.sleep(0.3)
            # 恢复系统状态
            self.waiting = [False for _ in range(self.progress_total_num)]
            self.finished = [False for _ in range(self.progress_total_num)]
            self.solve_queue = []
            self.deadlock = False
            self.allow_check_sys_safe = False
            print(f"available array={self.available}\nallocated
table={self.allocated}\nneed table={self.need}")


# 函数：程序入口
def main():
    try:
        # 输入资源名称和数量，在一行内完成，像 CPU=10 memory=13
        rs = list(input("Input all the resource and the available num, like R=1,
```

```
split with space:").split(' '))
        rs_name = []
        rs_avail = []
        for i in rs:
            tmp = i.split('=')
            rs_name.append(tmp[0])
            rs_avail.append(int(tmp[-1]))
        # 输入最大需求矩阵，多行完成，若使用空格切片失败则进入下一内容的输入
        mx_tb: typing.List[list] = []
        count = 1
        while True:
            try:
                s = input(f"Input the No.{count} progress maximum table, split
with space, other to break:").split(' ')
                mx_tb.append(list(map(int, s)))
                count += 1
            except Exception:
                break
        # 输入已分配矩阵，多行输入，空格切片
        allocated_tb: typing.List[list] = []
        count = 1
        while True:
            try:
                s = input(f"Input the No.{count} progress allocated table, "
                          f"split with space, other to break:").split(' ')
                allocated_tb.append(list(map(int, s)))
                count += 1
            except Exception:
                break
        # 使用输入的信息创建 computer 类的对象 cp
        cp = Computer(rs_name, rs_avail, mx_tb, allocated_tb)
        # 首先执行检查安全性算法
        is_died_lock = cp.check_system_is_not_safety()
        print("check sys init died lock state:", is_died_lock)
        # 如果不安全，则退出
        if is_died_lock:
            print("Sys occurs died lock, could not apply resource for
progress...")
            print("exit...")
            return
        # 若安全，则进行后面部分
        # 休眠一定时间，使得子线程可以运行完毕，避免修改到后面的数据
        time.sleep(0.01)
        # 回退原有的信息（验证系统安全性的时候对相关数据有修改）
```

```
            cp.reset_bk_inf()
            # 输入要申请资源的线程的编号
            progress_num = int(input(f"Input an index(max: {cp.progress_total_num-1})
to apply resource and check by "
                                     f"banker algorithm:"))
            # 输入需要申请的资源的数量，一行，按照每个进程的个数，如0 1，空格切片
            apply_arr = list(map(int, input(f"Input the resources {cp.resources} want
to apply, split by space:").split(' ')))
            # 进行申请
            cp.apply_resource(progress_num, apply_arr)
        # 捕捉异常，输出
        except Exception as e:
            print(e)



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

4、程序运行时的过程

使用课堂上的例子：

主观题 10分

某一系统中，有四类资源R1、R2、R3、R4，以及五个并发进程，
要求：
1. 完成下面的资源分配矩阵，并判断当前系统的状态
2. 当P2申请资源为（0，1，0，0）时，系统是否安全

资源 / 进程	Max				Allocation				Need				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
													2	1	0	0
P0	0	0	1	2	0	0	1	2								
P1	2	7	5	0	2	0	0	0								
P2	6	6	5	6	0	0	3	4								
P3	4	3	5	6	2	3	5	4								
P4	0	6	5	2	0	3	3	2								

输入：

R1=2 R2=1 R3=0 R4=0

0 0 1 2

2 7 5 0

6 6 5 6

4 3 5 6

0 6 5 2

0 0 1 2

2 0 0 0

0 0 3 4

2 3 5 4

0 3 3 2

2

0 1 0 0

输出：

Progress 4 is waiting for ['R2', 'need = 3', 'available = 1']&['R3', 'need = 2', 'available = 0'] now

Progress 4 is waiting for ['R2', 'need = 3', 'available = 1']&['R3', 'need = 2', 'available = 0'] now

Progress 0 is running now

Progress 0 finished, available=[2, 1, 1, 2]

Progress 1 is waiting for ['R2', 'need = 7', 'available = 1']&['R3', 'need = 5', 'available = 1'] now

Progress 2 is waiting for ['R1', 'need = 6', 'available = 2']&['R2', 'need = 6', 'available = 1']&['R3', 'need = 2', 'available = 1'] now

Progress 3 is running now

Progress 3 finished, available=[4, 4, 6, 6]

Progress 1 is waiting for ['R2', 'need = 7', 'available = 4'] now

Progress 4 is running now

Progress 4 finished, available=[4, 7, 9, 8]

Progress 2 is waiting for ['R1', 'need = 6', 'available = 4'] now

Progress 1 is running now

Progress 1 finished, available=[6, 7, 9, 8]

Progress 2 is running now

Progress 2 finished, available=[6, 7, 12, 12]

solve order: 0 --> 3 --> 4 --> 1 --> 2

check sys init died lock state: False

Input an index(max: 4) to apply resource and check by banker algorithm:Input the resources ['R1', 'R2', 'R3', 'R4'] want to apply, split by space:-----try to allocate and check died lock-----

after prepare allocate, the system situation:

available array=[2, 0, 0, 0]

allocated table=[[0, 0, 1, 2], [2, 0, 0, 0], [0, 1, 3, 4], [2, 3, 5, 4], [0, 3, 3, 2]]

need table=[[0, 0, 0, 0], [0, 7, 5, 0], [6, 5, 2, 2], [2, 0, 0, 2], [0, 3, 2, 0]]

Progress 4 is waiting for ['R2', 'need = 3', 'available = 0']&['R3', 'need = 2', 'available = 0'] now

Progress 3 is waiting for ['R4', 'need = 2', 'available = 0'] now

Progress 1 is waiting for ['R2', 'need = 7', 'available = 0']&['R3', 'need = 5', 'available = 0'] now

Progress 2 is waiting for ['R1', 'need = 6', 'available = 2']&['R2', 'need = 5', 'available = 0']&['R3', 'need = 2', 'available = 0']&['R4', 'need = 2', 'available = 0'] now

Progress 0 is running now

Progress 0 finished, available=[2, 0, 1, 2]

Progress 2 is waiting for ['R1', 'need = 6', 'available = 2']&['R2', 'need = 5', 'available = 0']&['R3', 'need = 2', 'available = 1'] now

Progress 3 is running now

Progress 3 finished, available=[4, 3, 6, 6]

Progress 1 is waiting for ['R2', 'need = 7', 'available = 3'] now

Progress 4 is running now

Progress 4 finished, available=[4, 6, 9, 8]

Progress 1 is waiting for ['R2', 'need = 7', 'available = 6'] now

Progress 2 is waiting for ['R1', 'need = 6', 'available = 4'] now

Progress 1 is waiting for ['R2', 'need = 7', 'available = 6'] now

Progress 2 is waiting for ['R1', 'need = 6', 'available = 4'] now

Allocate failed: died lock will occur

return to last state

available array=[2, 1, 0, 0]

allocated table=[[0, 0, 1, 2], [2, 0, 0, 0], [0, 0, 3, 4], [2, 3, 5, 4], [0, 3, 3, 2]]

need table=[[0, 0, 0, 0], [0, 7, 5, 0], [6, 6, 2, 2], [2, 0, 0, 2], [0, 3, 2, 0]]

从输出可以看到，最开始的系统没有死锁，程序找出了一条可以完成的路径（见红色字体标出部分）；分配系统资源后，程序尝试寻找可以解决的路径失败，当尝试到 0-3-4 后便开始了死锁，遂将系统状态进行了回退。

由于线程之间的先后顺序不确定，故每一次运行的输出可能不尽相同。