一、实验目的

多道系统中，进程与进程之间存在同步与互斥关系。当就绪进程数大于处理机数时，需按照某种策略决定哪些进程先占用处理机。在可变分区管理方式下，采用**首次适应算法**实现主存空间的分配和回收。

本实验模拟**实现处理机调度**及**内存分配及回收**机制，以对处理机调度的工作原理以及内存管理的工作过程进行更深入的了解。

二、实验内容及要求

1.实验内容

（1）选择一个调度算法，实现处理机调度；

（2）结合（1）实现主存储器空间的分配和回收。

2.实验具体要求

（1）设计一个抢占式优先权调度算法实现多处理机调度的程序，并且实现在可变分区管理方式下，采用首次适应算法实现主存空间的分配和回收。

（2）PCB内容包括：进程名/PID；要求运行时间（单位时间）；优先权；状态；进程属性：独立进程、同步进程（前趋、后继）。

（3）可以随机输入若干进程，可随时添加进程，并按优先权排序；

（4）从就绪队首选进程运行：优先权-1；要求运行时间-1；要求运行时间为0时，撤销该进程；一个时间片结束后重新排序，进行下轮调度；

（5）考虑两个处理机，考虑同步进程的处理机分配问题，每次调度后，显示各进程状态，运行进程要显示在哪个处理机上执行。

（6）规定道数，设置后备队列和挂起状态。若内存中进程少于规定道数，可自动从后备队列调度一作业进入。被挂起进程入挂起队列，设置解挂功能用于将制定挂起进程解挂入就绪队列。

（7）结合实验一pcb增加所需主存大小，主存起始位置；采用首次适应算法分配主存空间。

（8）自行假设主存空间大小，预设操作系统所占大小并构造未分分区表。表目内容：起址、长度、状态（未分/空表目）。

（9）进程完成后，回收主存，并与相邻空闲分区合并。

（10）最好采用图形界面；

三、实验的步骤

（1）需求分析：了解基本原理，确定软件的基本功能，查找相关资料，画出基本的数据流图；

（2）总体设计：确定软件的总体结构、模块关系和总体流程；

（3）详细设计：确定模块内部的流程和实现算法。

（4）上机编码和调试；

（5）实际数据运行测试。

需求分析：

主体设计

先分模块设计好主存类，PCB总队列类（需要先设计好PCB块类），处理器类。然后设计操作系统类System，内含主存，PCB队列，处理器以及四种队列，在System联系主存，PCB队列，处理器和四种队列完成进程调度和内存管理并封装好对外的接口。

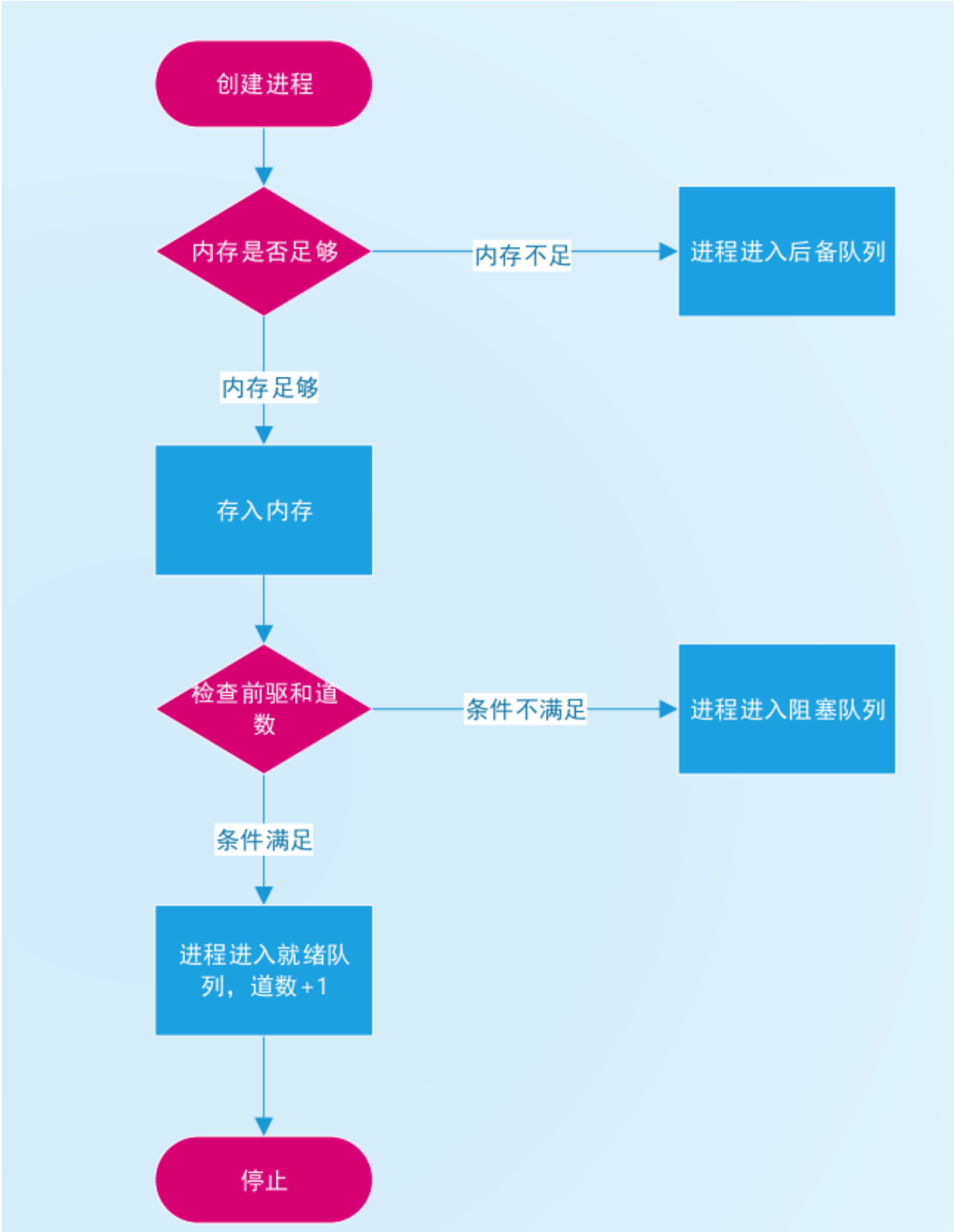
进程调度：

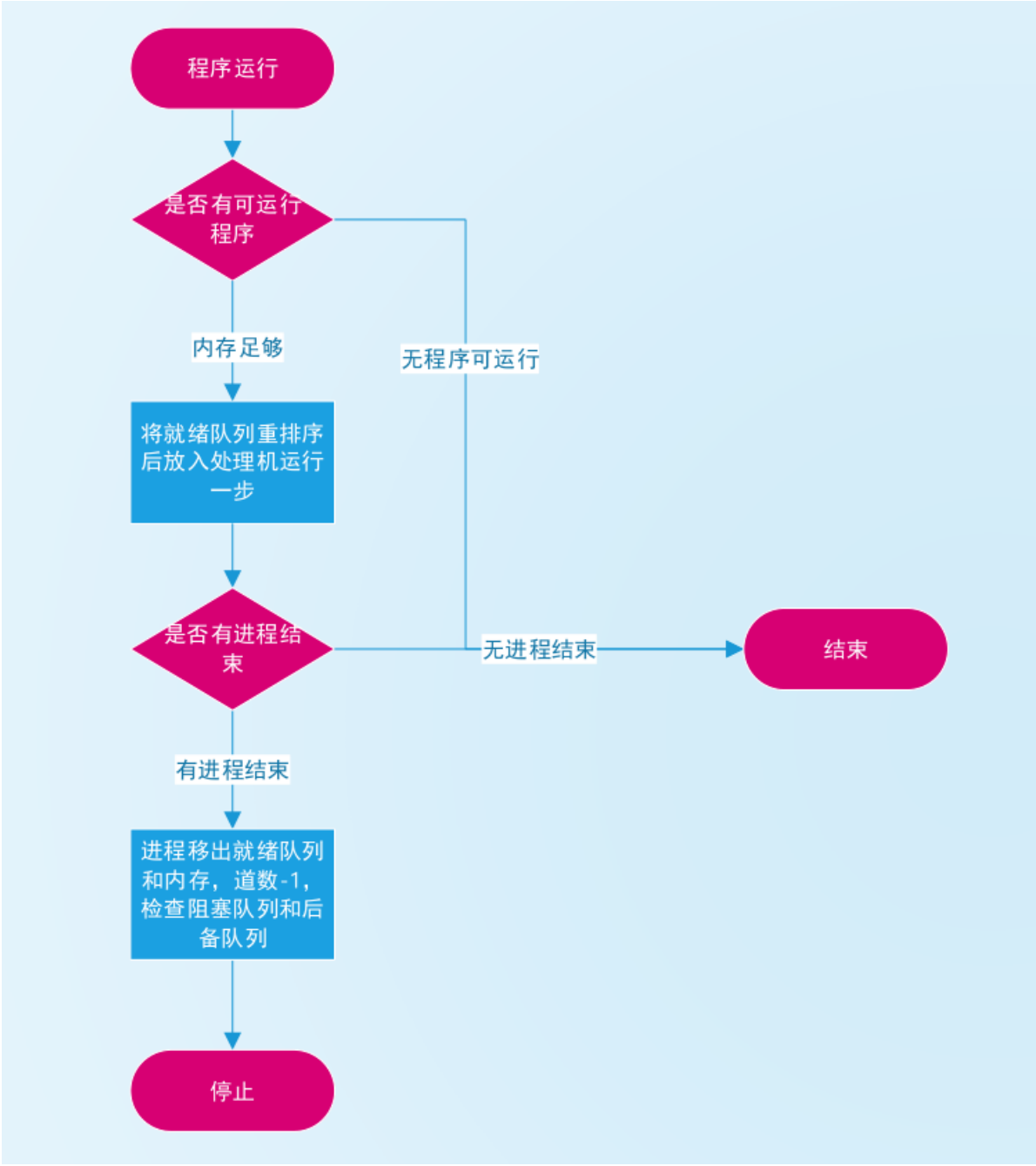
阻塞队列在等待前驱完成和道数两个条件，挂起队列等待解挂，后备队列等待内存。

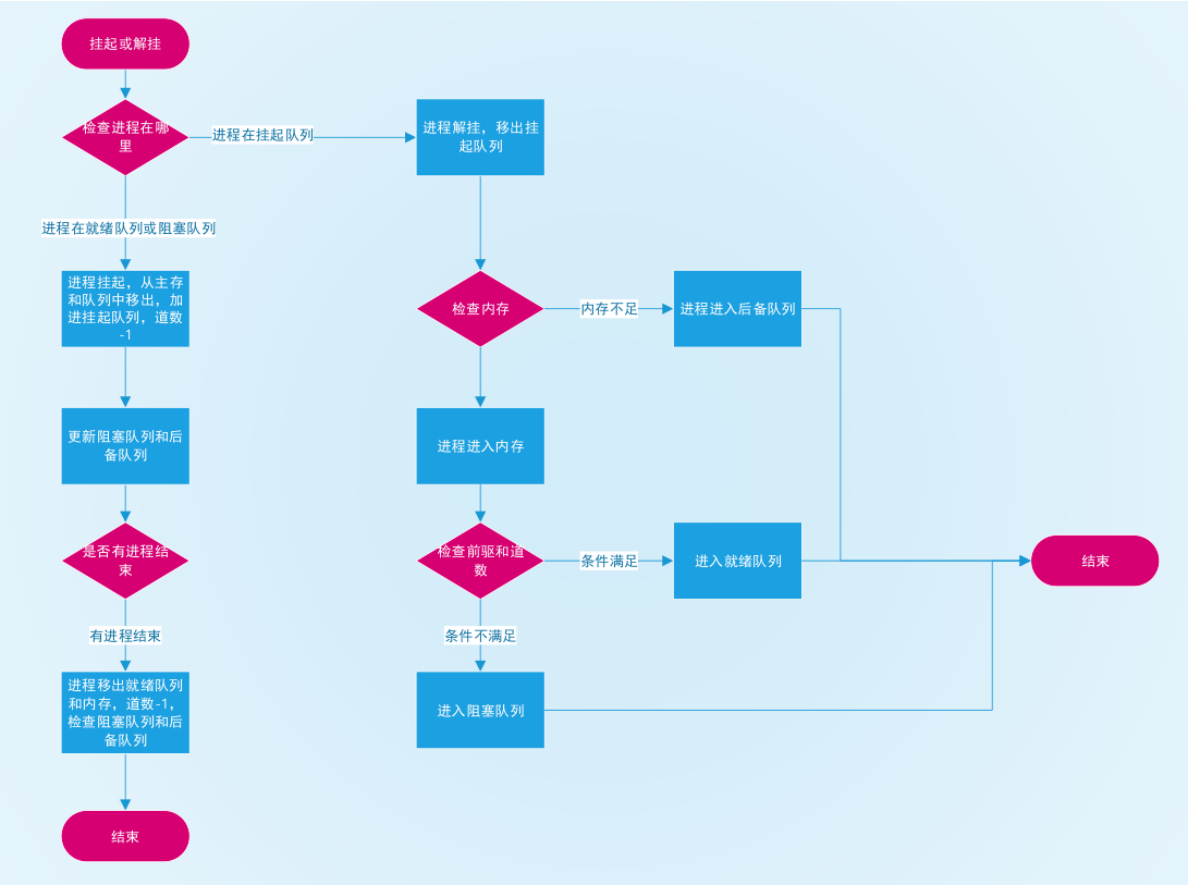
首先要先创建进程。如果内存不够，PCB存进**后备队列。**内存足够则存入内存，同时检查前驱和当前道数，满足则加入**就绪队列**，道数+1；否则加入**阻塞队列。**

模拟处理器处理一个时间片，将就绪队列重排序后放入处理器类进行处理一个时间单位，按要求时间和权值同时减1，结束后若有进程结束，则移出就绪队列和内存，道数-1。进程结束，道数减少，**阻塞队列**可能有进程能够进入就绪队列，需要更新；内存释放，**后备队列**可能有进程能够进入内存，需要更新。

当进程处于就绪队列或阻塞队列时可以进行进程挂起，挂起后需要将进程从主存中移除，从**就绪队列/阻塞队列**中移除并添加进**挂起队列。**进程挂起内存释放道数减少，同上需要更新**阻塞队列**和**后备队列。**当进程处于挂起队列时可以解挂，移除挂起队列，同时进行检查。如果内存不够，PCB存进**后备队列。**内存足够则存入内存，同时检查前驱和当前道数，满足则加入**就绪队列**，道数+1；否则加入**阻塞队列。**


创建进程
内存是否足够
内存不足
进程进入后备队列
内存足够
存入内存
检查前驱和道数
条件不满足
进程进入阻塞队列
条件满足
进程进入就绪队列，道数+1
停止


程序运行
是否有可运行程序
内存足够
无程序可运行
将就绪队列重排序后放入处理机运行一步
是否有进程结束
无进程结束
结束
有进程结束
进程移出就绪队列和内存，道数-1，检查阻塞队列和后备队列
停止



内存管理：

内存管理采用首次适应算法，方便起见没有另设空闲内存分区表，直接在内存分区表从头到尾遍历找到第一个匹配的空闲块即可。内存分区表记录序号，起址、长度、状态。

设计PCB块

主要是设计好成员变量

```
PCB类
成员变量：进程号，所需运行时间，所需内存空间，优先权，进程状态，进程属性，前驱进程PID，后继进程PID集合
成员函数：
获取8个变量的只读接口函数
toJson()：将属性以Json格式返回
setState()：设置进程状态
addSuccessor()：添加后继进程
process()：模拟进程运行
preFinish()：判断前驱集合是否都结束了
```

进程运行

根据题目要求每次运行后优先权-1；要求运行时间-1。优先权为1时不再变动

```
    def process(self):
        if self.__priority > 1:
            self.__priority -= 1
        if self.__time > 0:
            self.__time -= 1
        if self.__time == 0:
            self.setState(PCBState.EXIT)
            return True  # 进程结束
        else:
            self.setState(PCBState.RUNNING)
            return False
```

设计PCB队列

```
PCB队列类（保存所有PCB块）
getNextPID:自动生成PID
getPCBByPID:通过PID返回PCB
toJson:以Json格式返回
append:PCB入队列
setPCBSuccessor:用新的PCB更新队列中的PCB的后继
```

这个没有说明特殊的，只是需要用一个PCB队列来存储所有PCB块的信息，包括已完成的，方便其后继节点查询，简单封装一下就好了。

设计主存类

设计内存分区表，方便起见分区号用PID代替，其中-2代表未分配内存项，-1代表操作系统占用，显然运行过程中分区表始终以起址排序。

```
MainMemory主存类
成员变量：
__memory
成员方法：
checkAssignable：检查主存是否可分配内存给PCB，返回插入主存分区表的位置序号
insertPCB:插入新的PCB到__memory中
checkProcessable:判断是否有进程可以运行
removePCB:从主存中移除PCB
toJson:以Json格式返回内存分区表信息
__mergeMemory:合并空闲块
```

插入PCB进内存

将PCB插入到__memory的memoryBlockNum位置上，需要先取出对应的内存块并将其大小与PCB大小比较，如果相等直接改属性，否则要将内存块进行切分，分出一段空闲内存块重新插入内存中

```
    def insertPCB(self, pcb: PCB, index: int):
        partition = self.__memory[index]   # 获取被划分块
        pcb_ram = pcb.getRam()  # 获取内存大小
        pcb_id = pcb.getPID()   # 获取PID
        pcb.setState(PCBState.ACTIVE_READY)  # 更新PCB状态
        # 内存长度正好相等
        if partition[2] == pcb_ram:
            partition[0] = pcb_id
            partition[3] = MemoryBlockState.ASSIGNED
```

```
        # 否则划分空闲块
        else:
            # 插入一块被划分出来的空闲块
            self.__memory.insert(index + 1,
                                 [-2, partition[1] + pcb_ram, partition[2] -
pcb_ram, MemoryBlockState.UNASSIGNED])
            # 修改原空闲块为被分配
            partition[0] = pcb_id
            partition[2] = pcb_ram
            partition[3] = MemoryBlockState.ASSIGNED
```

移除PCB

遍历内存块找到被删除的进程后修改该内存块属性，然后调用__mergeMemory()将周围的内存块合并

```
    def removePCB(self, pcb: PCB,):
        # 从内存中移除PCB
        memoryBlockNum = 0
        pid = pcb.getPID()
        for index, value in enumerate(self.__memory):
            if value[0] == pid:
                memoryBlockNum = index
        self.__memory[memoryBlockNum][0] = -2
        self.__memory[memoryBlockNum][3] = MemoryBlockState.UNASSIGNED
        self.__mergeMemory()
```

合并内存块

合并未分配的内存空间，每次结束一个进程最多产生一段空闲内存，所以将连续的内存块存进列表。遍历内存，第一个空闲块用来改变标志位flag，紧接着的第二个空闲块才开始加入列表。最后将列表的内存块一一移出并对应增加第一个空闲块的大小属性。

```
    def __mergeMemory(self):
        flag = False
        memoryBlockToMergeList = []  # 记录要合并的第二块及其后内存块的位置
        for index, value in enumerate(self.__memory):
            pid = value[0]
            if pid == -2:  # 检测到未分配的内存空间（-2）
                if not flag:  # 第一次检测到未分配的内存空间（-2）
                    flag = True
                else:  # 不是第一次检测到未分配的内存空间（-2），即有连续的未分配空间
                    memoryBlockToMergeList.append(index)
            if pid != -2:  # 检测到已分配的内存空间（!=-2）
                flag = False
        if memoryBlockToMergeList:
            beginMemoryBlockNum = memoryBlockToMergeList[0] - 1  # 要合并的第一个内
存块的位置
            for partition in memoryBlockToMergeList[::-1]:
                memoryBlockToMerge = self.__memory.pop(partition)  # 倒序pop
                self.__memory[beginMemoryBlockNum][2] += memoryBlockToMerge[2]  #
累加大小
```

设计处理器类

```
Processor
成员变量：
__processorNum:处理机个数
成员函数：
process：模拟处理器运行一步
```

模拟进程运行

设有n个处理器，则从就绪队列（已排序好）中取出n个进程运行，注意要判断就绪队列长度是否小于处理器个数，否则会有数组越界的报错。

```
    def process(self, pcb_queue: PCBQueue, ready_queue:list):
        finish = []  # 返回结束的进程的pid
        # 从就绪队列中取出前几个分配给对应处理机进行一步运行
        for i in range(self.__processorNum):
            if i < len(ready_queue):
                # 获取pcb并使该pcb运行一步
                pcb = pcb_queue.getPCBByPID(ready_queue[i])
                if pcb.process():
                    finish.append(pcb.getPID())
        return finish
```

设计操作系统类

```
成员变量：
四种队列
pcbQueue
mainMemory
processor
成员函数：
createPCB：创建新的进程
checkSuspend：挂起或解挂进程
process：处理器运行一步
getQueue：获取四种队列
```

要点如下：

创建进程的处理：

创建PCB块并加入PCB队列，然后检查是否能放入内存，如果不能则加进后备队类；如果可以则放入内存并检查前驱，如果前驱都完成了则添加进就绪队列，否则加进阻塞队列

```
    def createPCB(self, pid: int, time: int, ram: int, priority: int, prop:
Property, precursor: set, state=PCBState.CREATE):
        pcb=PCB(pid,time,ram,priority,prop,precursor,state)
        self.pcbQueue.append(pcb)
        flag, index = self.mainMemory.checkAssignable(pcb)
        if flag:
            self.mainMemory.insertPCB(pcb, index)
            if(pcb.preFinish(self.pcbQueue)):
                self.ready_queue.append(pcb.getPID())
                self.track += 1
            else:
                self.block_queue.append(pcb.getPID())
        else:
            self.backup_queue.append(pcb.getPID())
```

进程运行一步

每次运行前就绪队列要重排序，检查是否有进程可以运行，如果有将就绪队列丢进处理器模拟一步运行并返回结果finish，finish包含这次运行后结束的进程pid列表，如果列表非空即有进程结束，阻塞队列可能有进程可以进行了,检查阻塞队列；后备队列可能有进程可以载入内存了,检查后备队列

```
    def process(self):
        self.__sortReady()
        if len(self.ready_queue) > 0:
            finish = self.processor.process(self.pcbQueue, self.ready_queue)
            for pid in finish:
                self.mainMemory.removePCB(self.pcbQueue.getPCBByPID(pid))  # 移出
主存
                self.ready_queue.remove(pid)  # 移出就绪队列
                self.track -= 1
            self.__updateBlock()
            self.__updateBackup()
            return True
        else:
            return False
```

挂起进程

在就绪队列或者阻塞队列中找到该pid并移除，根据pid找到pcb在内存中移除，添加进挂起队列。有进程挂起就有内存空闲，需要更新阻塞队列和后备队列

```
    def suspend(self, pid: int):
        if pid in self.ready_queue:
            self.ready_queue.remove(pid)
            self.track -= 1  # 道数-1
        elif pid in self.block_queue:
            self.block_queue.remove(pid)
        pcb = self.pcbQueue.getPCBByPID(pid)
        self.mainMemory.removePCB(pcb)
        self.suspend_queue.append(pid)
        pcb.setState(PCBState.SUSPENDING)  # 设置挂起状态
        self.__updateBlock()
        self.__updateBackup()
```

解挂进程

解挂进程，将该进程移除挂起队列，检查是否能放入内存，如果不能则加进后备队列；如果可以则放入内存并检查前驱，如果前驱都完成了则添加进就绪队列，否则加进阻塞队列

```
    def unsuspend(self, pid: int):
        if pid in self.suspend_queue:
            self.suspend_queue.remove(pid)
            pcb = self.pcbQueue.getPCBByPID(pid)
            flag, index = self.mainMemory.checkAssignable(pcb)
            if flag:
                self.mainMemory.insertPCB(pcb, index)
                if (pcb.preFinish(self.pcbQueue)):
                    self.ready_queue.append(pcb.getPID())
                    self.track += 1
                else:
                    self.block_queue.append(pcb.getPID())
            else:
                self.backup_queue.append(pcb.getPID())
```

接口处理

```
http://127.0.0.1:5000
/os/get_PCB_list     以'pcb_list'为键传递总的PCB信息队列
/os/create_PCB   接受表单信息以创建PCB
/os/get_main_memory 以main_memory为键传递主存分区表
/os/get_queue    传递四个队列ready_queue，backup_queue，block_queue，suspend_queue
/os/suspend      进行进程挂起或解挂的操作
/os/run          模拟进程运行一步
```

前端设计

前端采用Vue框架和element组件库辅助开发，负责接收用户输入同时做一些前端输入的简单的判断校验处理如判空，强制限选优先级，前驱进程限制为已有进程等。内存和四个队列的展示通过来展示，通过选择不同标签展示不同内容。同时和下方PCB信息的展示一样使用以表格的形式展示信息。

首先得有一个表单创建新的PCB，一个表单用来挂起/解挂队列；

然后要展示就绪队列，后备队列，阻塞队列，挂起队列，同时展示内存分区表。

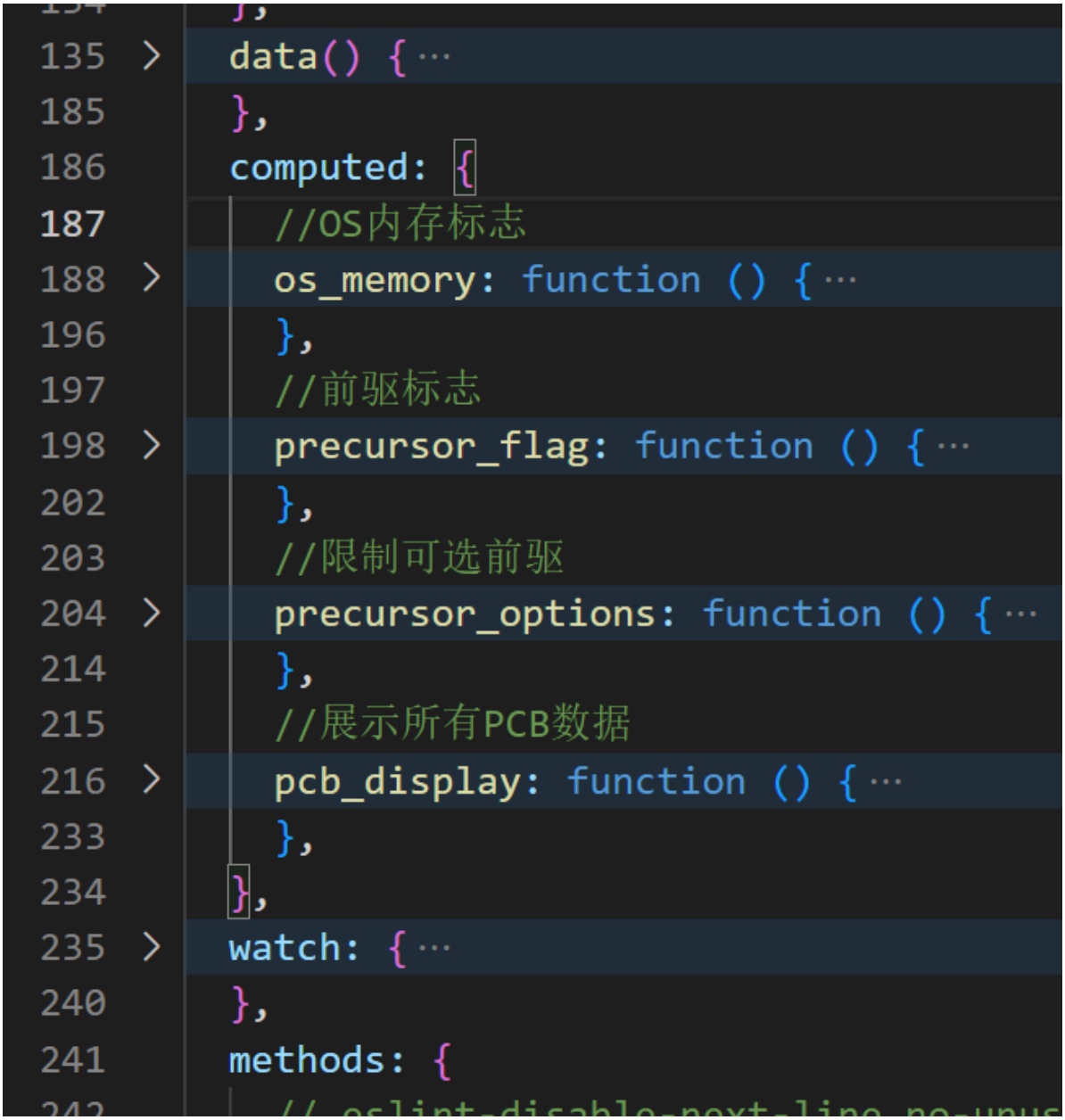
然后需要较大的区域展示PCB信息。

前端所需存储的数据：

```
operationName: 'tab_first',
displayName: 'tab_first',
//PCB内存状态表
pcbState: [
{"id": 0,"name": "进程创建",},
{"id": 1,"name": "活动就绪",},
{"id": 2,"name": "静止就绪",},
{"id": 3,"name": "阻塞状态",},
{"id": 4,"name": "进程运行",},
{"id": 5,"name": "进程挂起",},
```

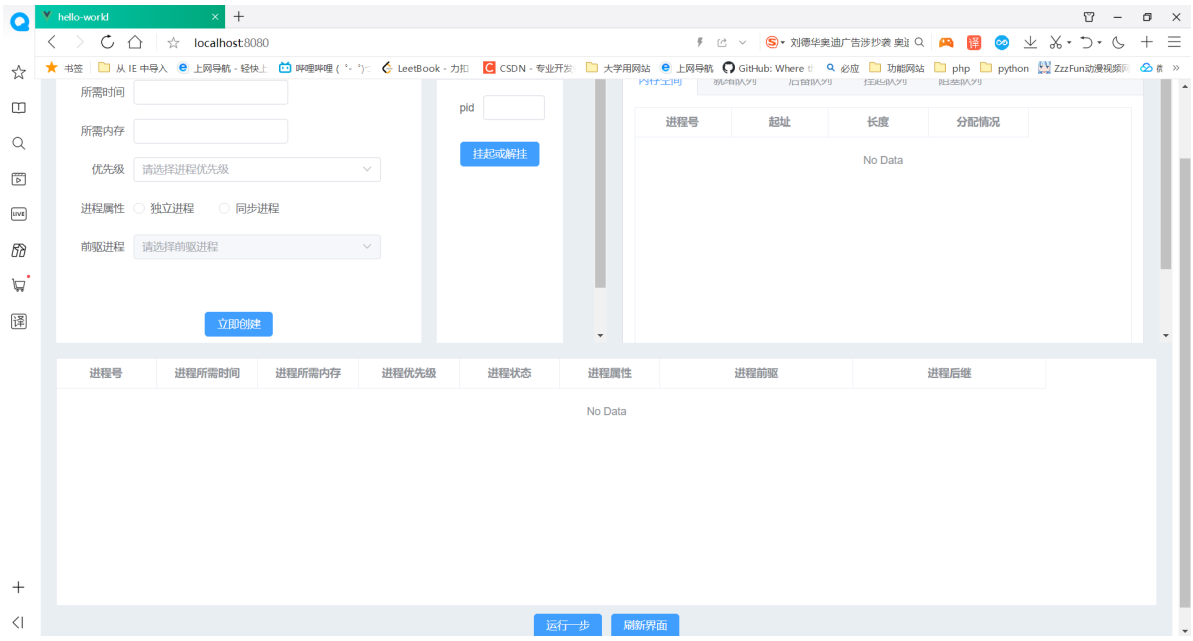
```
{"id": 6,"name": "进程结束",},
],
totalMemory: 100, //内存总容量
//创建PCB表单内容
form: {
time: '',
ram: '',
priority: '',
property: '',
precursor: [],
},
form2:{pid:''}, // 挂起解挂表单内容
pcbData: [],      //创建PCB的数据
mainMemory: [],   //主存分区表信息
processors: [],   //处理器信息
readyQueue: [],   //就绪队列
blockQueue: [],   //阻塞队列
backupQueue: [],  //后备队列
suspendQueue: [], //挂起队列
```

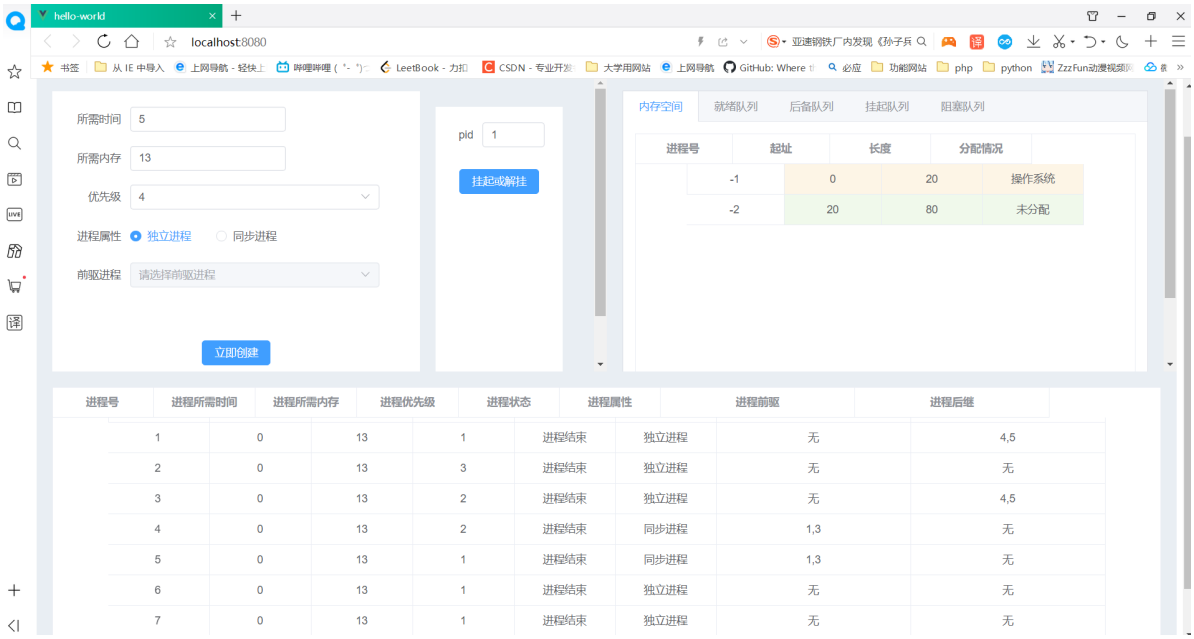
代码过多，这里只简单展示前端JS部分的结构



```
241   methods: {
242     // eslint-disable-next-line no-unused-vars
243     tableRowClassName({row, rowIndex}) {…
250     },
251     //发起请求，获取pcb_list刷新界面
252     refresh() {…
260     },
261     //发送请求，运行一步并接收新的PCB信息队列更新页面
262     run() {…
276     },
277     //发送请求，创建新的PCB块
278     createPCB() {…
314     },
315     //发送请求，获取主存分区表
316     getMainMemory() {…
324     },
325     //发送请求，获取四个队列
326     getQueue(){…
337     },
338     //挂起或解挂操作
339     suspend(){…
362     },
```

主界面设计如下：





进程号	进程所需时间	进程所需内存	进程优先级	进程状态	进程属性	进程前驱	进程后继
1	0	13	1	进程结束	独立进程	无	4,5
2	0	13	3	进程结束	独立进程	无	无
3	0	13	2	进程结束	独立进程	无	4,5
4	0	13	2	进程结束	同步进程	1,3	无
5	0	13	1	进程结束	同步进程	1,3	无
6	0	13	1	进程结束	独立进程	无	无
7	0	13	1	进程结束	独立进程	无	无