****

操作系统原理

实验报告

**姓名：\*\*\***

**班级：\*\*\*\*\*\*\***

**学号：\*\*\*\*\*\*\*\*\*\***

**计算机学院**

**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。

**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。

**二、需求分析和基本设计思想**

**1.需求分析**

本次处理机调度模拟程序主要分为以下模块：

① 输入模块：在任意时间片可以添加新进程。

② 挂起模块：在任意时间片可以对内存中进程进行挂起，转移至外存。

③ 运行模块：程序核心模块，从当前就绪队列中选取优先级最高的进程运行。当就绪队列存在空闲时从后背队列/解挂程序中调取适当程序加入就绪队列；当有进程运行完成时，需释放内存，同时调取合适进程加入就绪队列。

④ 显示模块：界面设计，显示当前各进程信息。

对于进程运行状态，分别设置四个队列进行存储：

ready\_queue：就绪队列，已分配内存进程。

back\_queue：后备队列，由于道数限制等原因未加入内存进程。

suspend\_queue：挂起队列，存储挂起进程，位于外存。

unsuspend\_queue：解挂队列，但由于道数限制等原因未加入就绪队列。

free\_mem\_link：储存空闲内存链表。

主要算法选择：

处理机调度：抢占式优先权调度算法；

主存空间的分配和回收：可变分区、首次适应算法。

**2.**基本设计思想

PCB的定义：

1. #ifndef PCB\_H
2. #define PCB\_H
3. #include<string>
4. **using** **namespace** std;
6. **typedef** **enum** {p\_ready,p\_standby,p\_running,p\_suspend} p\_state;
7. **class** pcb {
8. **public**:
9. **int** pid; //进程id
10. **int** time;//运行时间
11. **int** priority;//优先级
12. p\_state state;//状态
13. **int** base;//基址
14. **int** memory;//内存大小
15. **bool** is\_alone;//是否为独立进程
16. **int** bufempty;//p
17. **int** buffull;//v
18. **int** prev;//前驱
19. **int** succ;//后继
20. pcb(**int** p=0,**int** t=0,**int** pri=0,**int** m=0,**bool** is\_a = **true**,**int** pr = 0,**int** s = 0,**int** be = 0,**int** bf = 0):
21. pid(p),time(t),priority(pri),state(p\_state::p\_standby),base(-1),memory(m), is\_alone(is\_a),prev(pr),succ(s),bufempty(be),buffull(bf){}
22. };
24. #endif // PCB\_H

PCB内容包括：进程名/PID；要求运行时间（单位时间，time）；优先权（priority）；状态（state）；进程属性（is\_alone）：独立进程、同步进程（前趋prev、后继succ）。

空闲内存块的定义：

1. #ifndef MEM\_BLOCK\_H
2. #define MEM\_BLOCK\_H
4. **class** mem\_block{
5. **public**:
6. **int** base;//内存地址起始
7. **int** len;//长度
8. **bool** valid;//是否被分配（其实没有用）
9. mem\_block(**int** b,**int** l,**bool** v = **true**):base(b),len(l),valid(v){ };
10. };
12. #endif // MEM\_BLOCK\_H

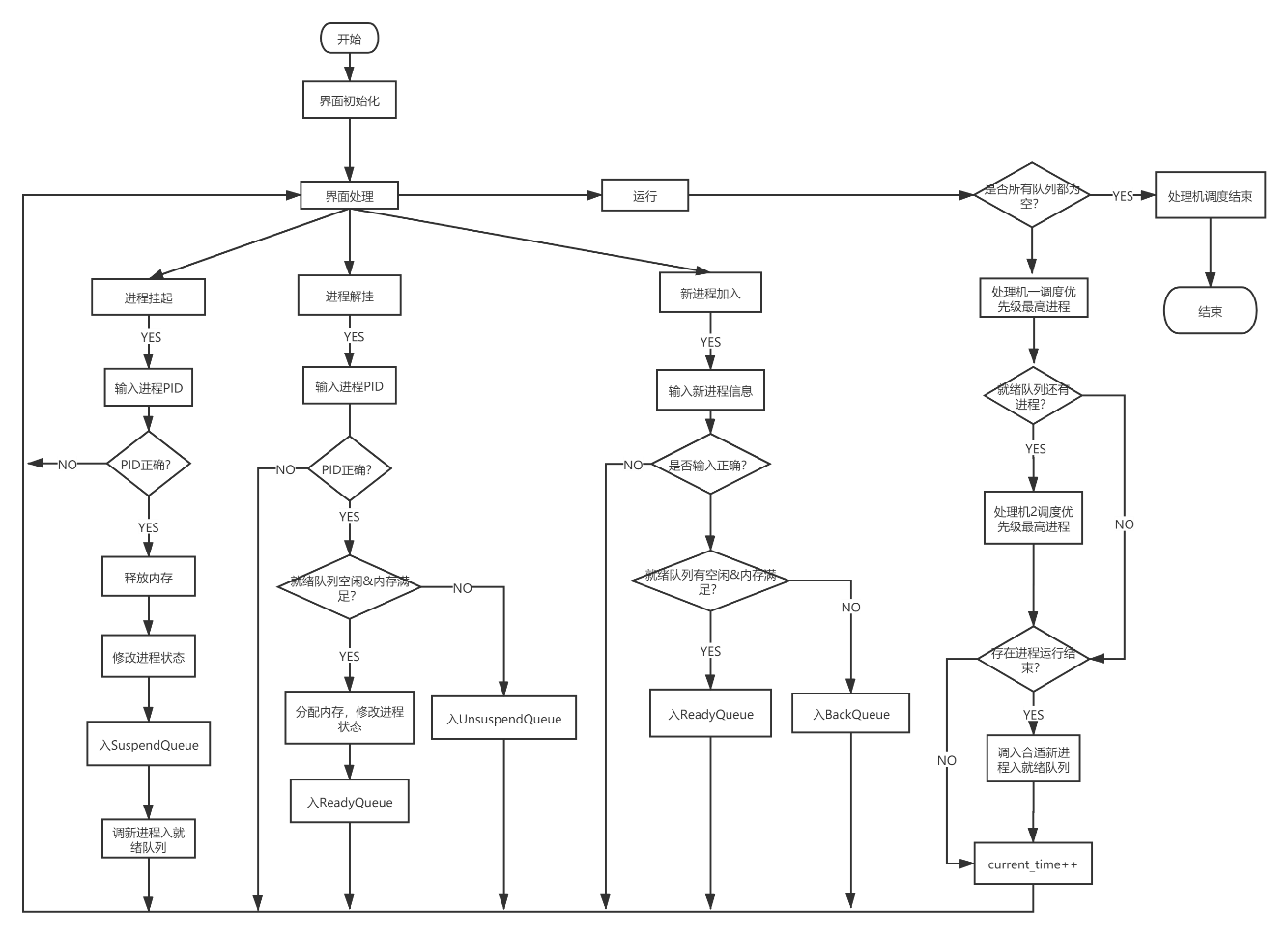
空闲内存块mem\_block内容包括空闲内存基址（base）和长度（len）。

全局变量：

1. vector<pcb> ready\_queue;  //就绪队列
2. vector<pcb> standby\_queue;//后备队列
3. vector<pcb> suspend\_queue;//挂起队列
4. vector<mem\_block> free\_mem\_link; //空闲内存链表
5. **const** **int** max\_ready = 4;  //道数
6. **const** **int** max\_mem = 100;  //内存空间100
7. **const** **int** osk\_mem = 20;   //内核总占内存20
8. **const** **int** max\_processor = 2; //两核

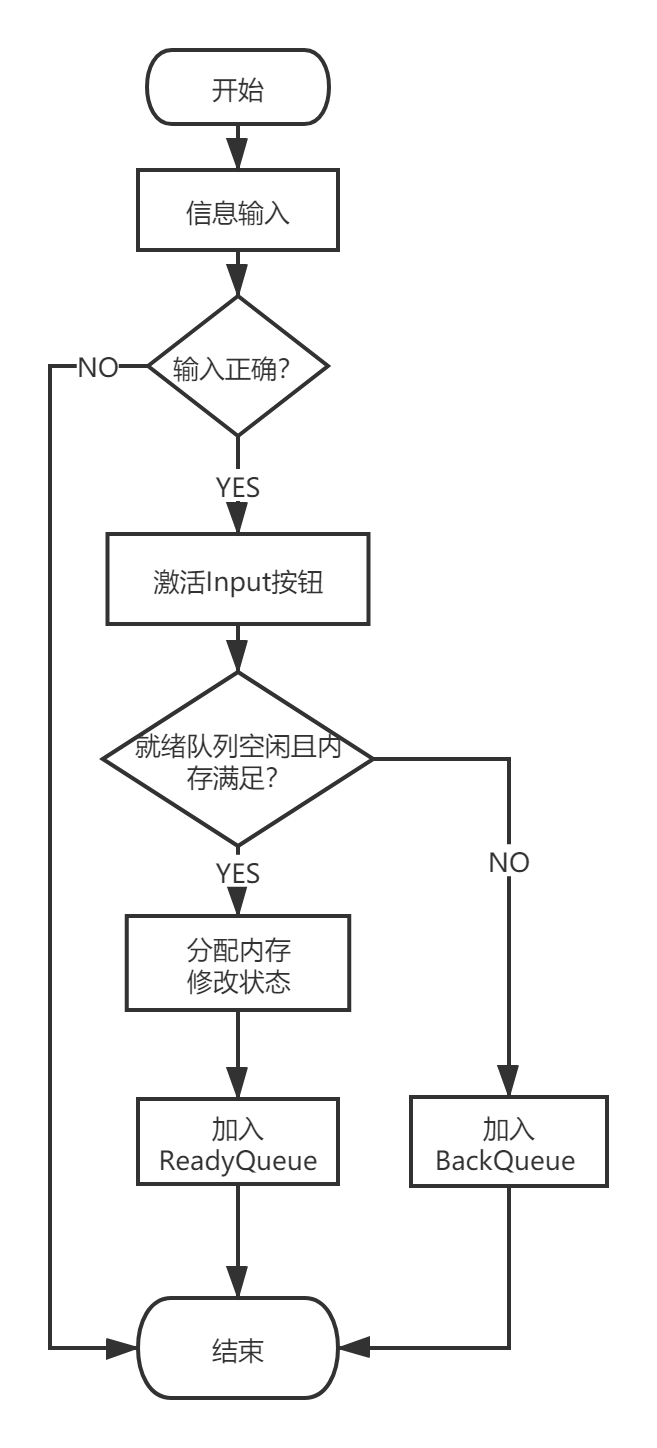
这里用vector来模拟操作系统里的链表结构。道数为驻留在内存中的最大进程数，max\_mem表示计算机内存大小。osk\_mem表示内核所占内存大小，max\_processor 代表最大处理机个数。

流程图：



**三、具体实现**

**1.输入模块**



Start按钮的槽函数，用来初始化全局变量。

1. **void** MainWindow::on\_start\_button\_clicked()
2. {
3. qDebug() << "void MainWindow::on\_start\_button\_clicked()";
4. ui->add\_job\_button->setEnabled(**true**);
5. ui->run\_button->setEnabled(**true**);
6. ui->suspend\_button->setEnabled(**true**);
7. ui->unsuspend\_button->setEnabled(**true**);
8. ui->start\_button->setEnabled(**false**);
9. ui->terminate\_button->setEnabled(**true**);
10. free\_mem\_link.push\_back(mem\_block(osk\_mem,max\_mem-osk\_mem));
11. **return** ;
12. }

Add Job 按钮是用来添加进程的，代码如下：

1. **void** MainWindow::on\_add\_job\_button\_clicked()
2. {
3. qDebug() << "void MainWindow::on\_add\_job\_button\_clicked()";
4. **int** new\_pid = ui->pid\_edit->text().toInt();
5. **int** new\_time = ui->time\_edit->text().toInt();
6. **int** new\_priority = ui->priority\_edit->text().toInt();
7. **int** new\_memory = ui->memory\_edit->text().toInt();
8. **if**(ui->syn\_edit->text().toCaseFolded() == 'n'){
9. add\_job(new\_pid,new\_time,new\_priority,new\_memory);
10. }**else**{
11. **int** new\_prev = ui->prev\_edit->text().toInt();
12. **int** new\_succ = ui->succ\_edit->text().toInt();
13. **int** new\_bufempty = ui->bufempty\_edit->text().toInt();
14. **int** new\_buffull = ui->bufful\_edit->text().toInt();
15. add\_job(new\_pid,new\_time,new\_priority,new\_memory,new\_prev,new\_succ,new\_bufempty,new\_buffull);
16. }
17. **return** ;
18. }

add\_job函数是重载的，分别对应输入的作业是独立的还是同步的。

加入独立进程的函数如下所示：

1. **void** add\_job(**int** new\_pid,**int** new\_time,**int** new\_priority,**int** new\_memory)
2. {
3. qDebug() << "void add\_job(int new\_pid,int new\_time,int new\_priority,int new\_memory)";
4. pcb new\_job\_pcb(new\_pid,new\_time,new\_priority,new\_memory);
5. **if**(check\_add\_job\_valid(new\_pid,new\_time,new\_priority,new\_memory)){
6. **int** mb\_index;
7. //尝试加入就绪队列
8. **if**((mb\_index = search\_for\_mem\_block(new\_memory)) != -1 && ready\_queue.size() < max\_ready){
9. new\_job\_pcb.base = free\_mem\_link[mb\_index].base;
10. new\_job\_pcb.state = p\_state::p\_ready;
11. allocate\_mem\_block(mb\_index,new\_memory);
12. ready\_queue.push\_back(new\_job\_pcb);
13. }**else**{
14. //道满或内存不够，加入后备队列
15. new\_job\_pcb.state = p\_state::p\_standby;
16. standby\_queue.push\_back(new\_job\_pcb);
17. }
18. }
19. **return** ;
20. }

加入同步进程的函数如下所示：

1. //同步进程
2. **void** add\_job(**int** new\_pid,**int** new\_time,**int** new\_priority,**int** new\_memory,**int** prev,**int** succ,**int** bufempty,**int** buffull)
3. {
4. qDebug() << "void add\_job(int new\_pid,int new\_time,int new\_priority,int new\_memory,int prev,int succ,int bufempty,int buffull)";
5. pcb new\_job\_pcb(new\_pid,new\_time,new\_priority,new\_memory,**false**,prev,succ,bufempty,buffull);
6. **if**(check\_add\_job\_valid(new\_pid,new\_time,new\_priority,new\_memory)){
7. **int** mb\_index;
8. //尝试加入就绪队列
9. **if**((mb\_index = search\_for\_mem\_block(new\_memory)) != -1 && ready\_queue.size() < max\_ready){
10. new\_job\_pcb.base = free\_mem\_link[mb\_index].base;
11. new\_job\_pcb.state = p\_state::p\_ready;
12. allocate\_mem\_block(mb\_index,new\_memory);
13. ready\_queue.push\_back(new\_job\_pcb);
14. }**else**{
15. //道满或内存不够，加入后备队列
16. new\_job\_pcb.state = p\_state::p\_standby;
17. standby\_queue.push\_back(new\_job\_pcb);
18. }
19. }
20. **return** ;
21. }

主要区别是初始化pcb的信息不同，同步进程的pcb需要更多的信息。

作业信息输入合法性检测函数如下，主要限制为：

① 各输入值要合法；

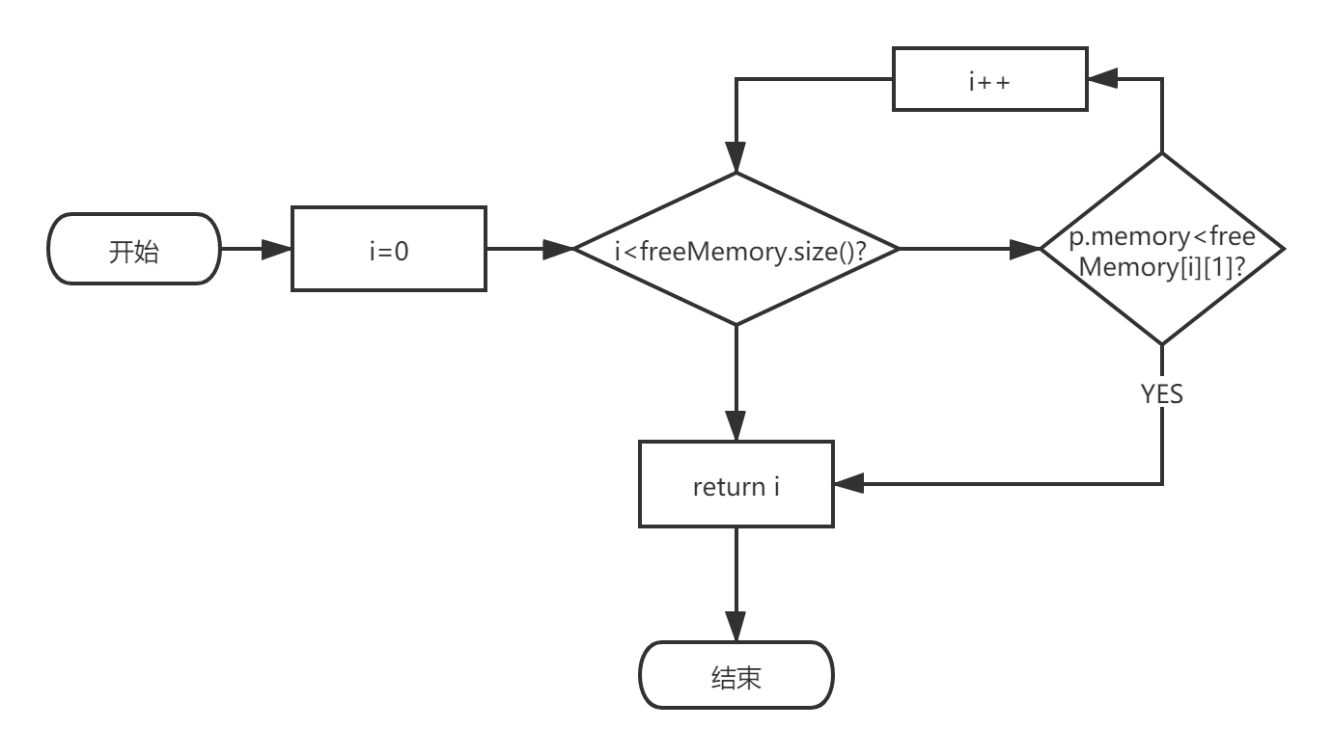
② 优先级要大于输入时间；

③ 进程pid不能已经存在。

1. **bool** check\_add\_job\_valid(**int** pid,**int** t,**int** pri,**int** mem)
2. {
3. qDebug() << "bool check\_add\_job\_valid(int pid,int t,int pri,int mem)";
4. **if**(pid <= 0 || t <= 0 || pri <= 0 || mem <= 0){
5. QMessageBox::critical(NULL,"Reject","Args should be greater than zero!",QMessageBox::Yes,QMessageBox::Yes);
6. **return** **false**;
7. }**else** **if**(mem > (max\_mem - osk\_mem)){
8. QMessageBox::critical(NULL,"Reject","Job is too large!", QMessageBox::Yes,QMessageBox::Yes);
9. **return** **false**;
10. }**else** **if**(pri < t){
11. QMessageBox::critical(NULL,"Reject","Priority should be greater than time!", QMessageBox::Yes, QMessageBox::Yes);
12. **return** **false**;
13. }
14. **for**(**int** i = 0;i < ready\_queue.size(); ++i){
15. **if**(ready\_queue[i].pid == pid){
16. QMessageBox::critical(NULL,"Reject","Pid has already existed!", QMessageBox::Yes,QMessageBox::Yes);
17. **return** **false**;
18. }
19. }
20. **for**(**int** i = 0;i < standby\_queue.size(); ++i){
21. **if**(standby\_queue[i].pid == pid){
22. QMessageBox::critical(NULL,"Reject","Pid has already existed!", QMessageBox::Yes,QMessageBox::Yes);
23. **return** **false**;
24. }
25. }
26. **for**(**int** i = 0;i < suspend\_queue.size(); ++i){
27. **if**(suspend\_queue[i].pid == pid){
28. QMessageBox::critical(NULL,"Reject","Pid has already existed!", QMessageBox::Yes,QMessageBox::Yes);
29. **return** **false**;
30. }
31. }
32. QMessageBox::about(NULL,"Accept","Add job successfully!");
33. **return** **true**;
34. }

**2.内存管理模块**

空闲内存分配流程如下：



空闲内存寻找函数如下：

1. **int** search\_for\_mem\_block(**int** p\_mem)
2. {
3. qDebug() << "int search\_for\_mem\_block(int p\_mem)";
4. sort(free\_mem\_link.begin(),free\_mem\_link.end(),cmp\_mem\_base);
5. //道满了
6. **if**(ready\_queue.size() >= max\_ready){
7. **return** -1;
8. }
9. //首次适配算法
10. **int** len = free\_mem\_link.size();
11. **for**(**int** i = 0; i < len; ++i){
12. **if**(free\_mem\_link[i].len >= p\_mem){
13. **return** i;
14. }
15. }
16. **return** -1;
17. }

先遍历空闲链表，如果找到了，则返回对应索引，否则则返回-1。

内存分配函数如下：

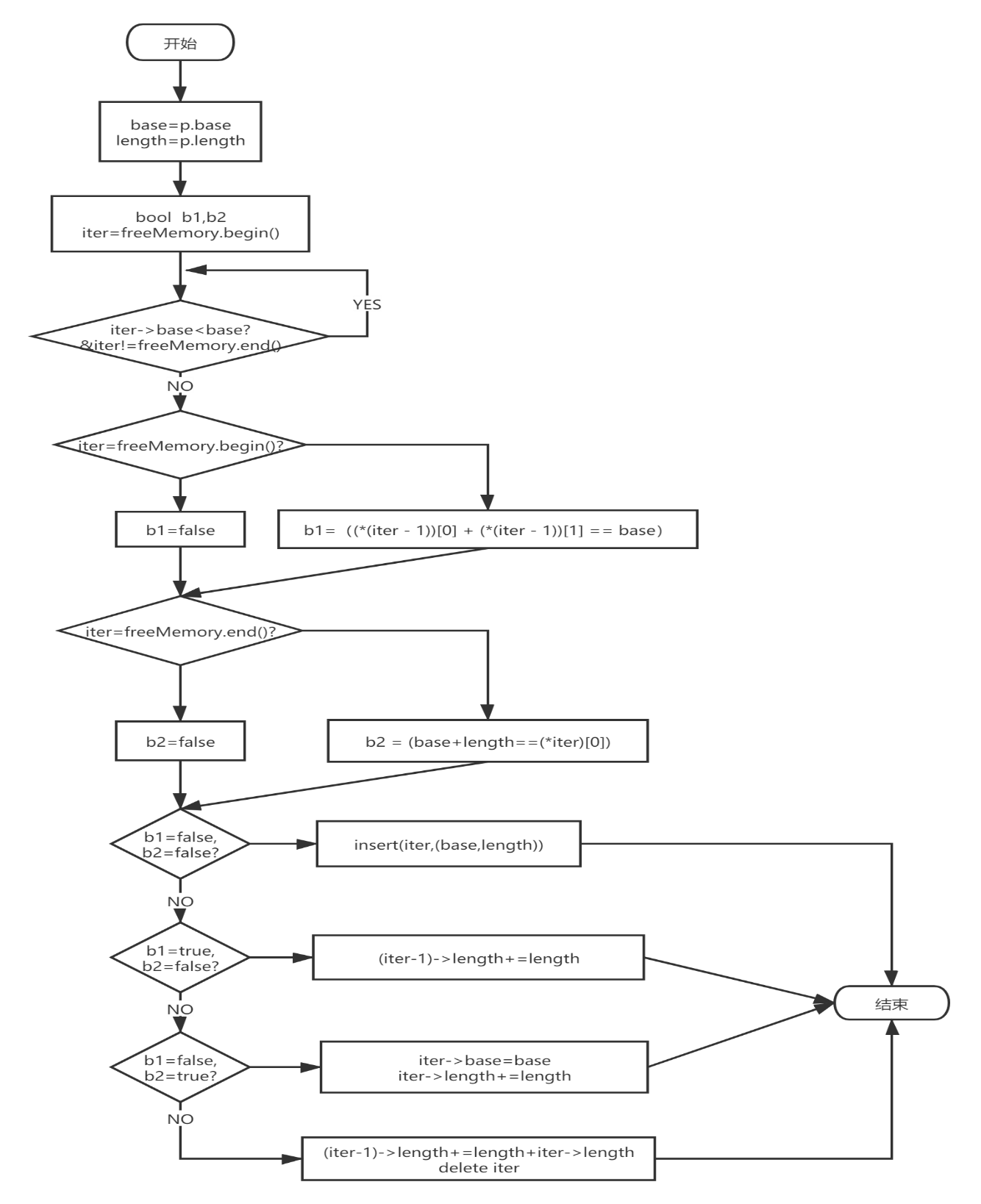
1. //分配并切割内存块
2. **void** allocate\_mem\_block(**int** index,**int** mem)
3. {
4. qDebug() << "void allocate\_mem\_block(int index,int mem)";
5. free\_mem\_link[index].base += mem;
6. free\_mem\_link[index].len -= mem;
7. **if**(free\_mem\_link[index].len == 0){
8. auto it = free\_mem\_link.begin() + index;
9. free\_mem\_link.erase(it);
10. }
11. **return** ;
12. }

对空闲内存块的大小进行修改，如果大小为0，则删除该内存块。

空闲内存块合并函数：

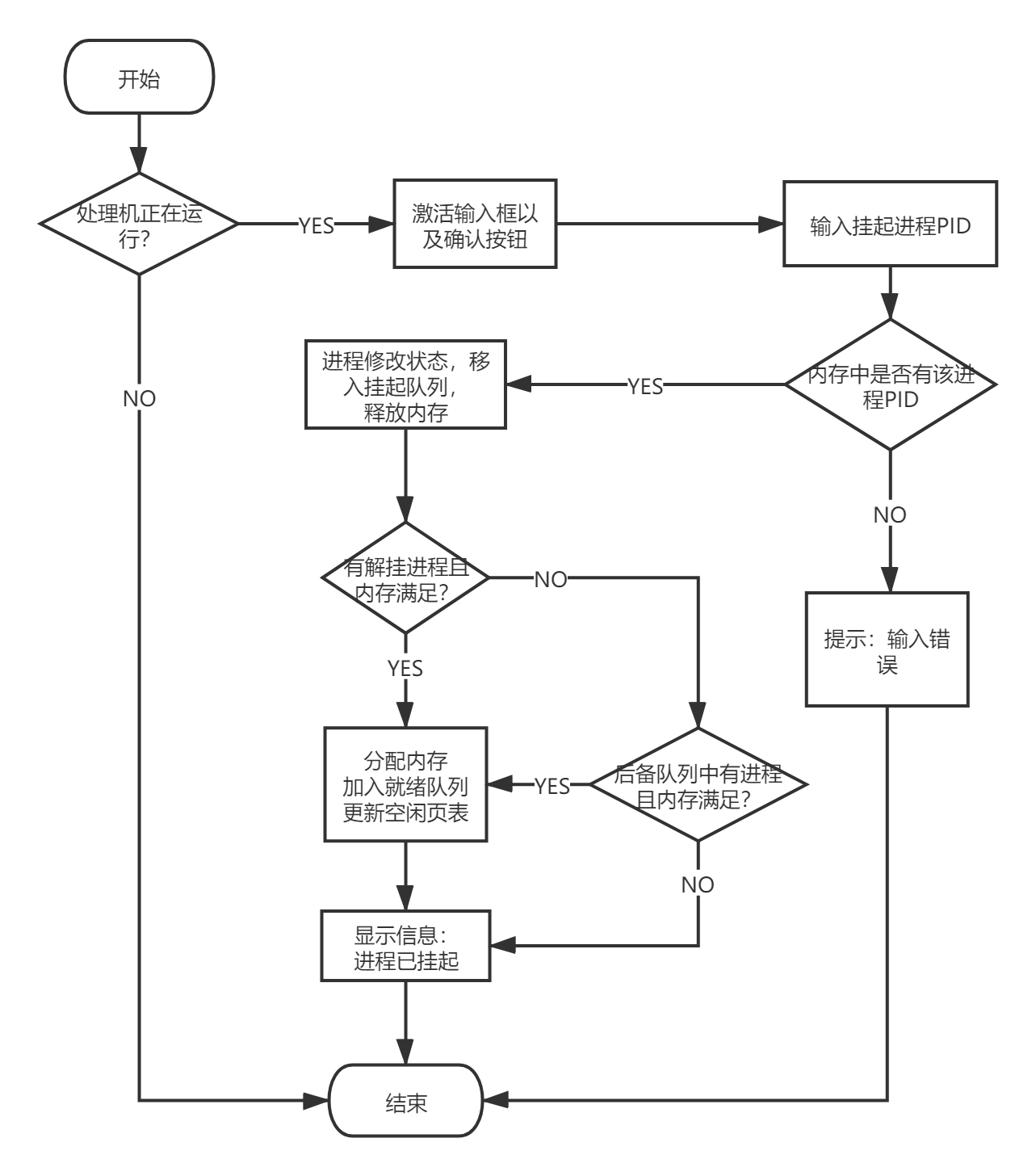
1. **void** merge\_mem\_block(**int** base)
2. {
3. qDebug() << "void merge\_mem\_block(int base)";
4. **int** i;
5. **for**(i = 0; i < free\_mem\_link.size(); ++i){
6. **if**(free\_mem\_link[i].base == base){
7. **break**;
8. }
9. }
10. **if**(i-1>=0 && free\_mem\_link[i-1].base+free\_mem\_link[i-1].len == base){
11. //前内存块要合并
12. base = free\_mem\_link[i].base = free\_mem\_link[i-1].base;
13. free\_mem\_link[i].len += free\_mem\_link[i-1].len;
14. free\_mem\_link[i-1].valid = **false**;
15. auto it = free\_mem\_link.begin() + i-1;
16. free\_mem\_link.erase(it);
17. }
18. **for**(i = 0; i < free\_mem\_link.size(); ++i){
19. **if**(free\_mem\_link[i].base == base){
20. **break**;
21. }
22. }
23. **if**(i+1<free\_mem\_link.size() && free\_mem\_link[i].base+free\_mem\_link[i].len == free\_mem\_link[i+1].base){
24. //后内存块要合并
25. free\_mem\_link[i].len += free\_mem\_link[i+1].len;
26. free\_mem\_link[i+1].valid = **false**;
27. auto it = free\_mem\_link.begin() + i+1;
28. free\_mem\_link.erase(it);
29. }
30. **return** ;
31. }

空闲内存块释放流程如下：



**3.进程挂起模块**

基本思路如流程图所示：



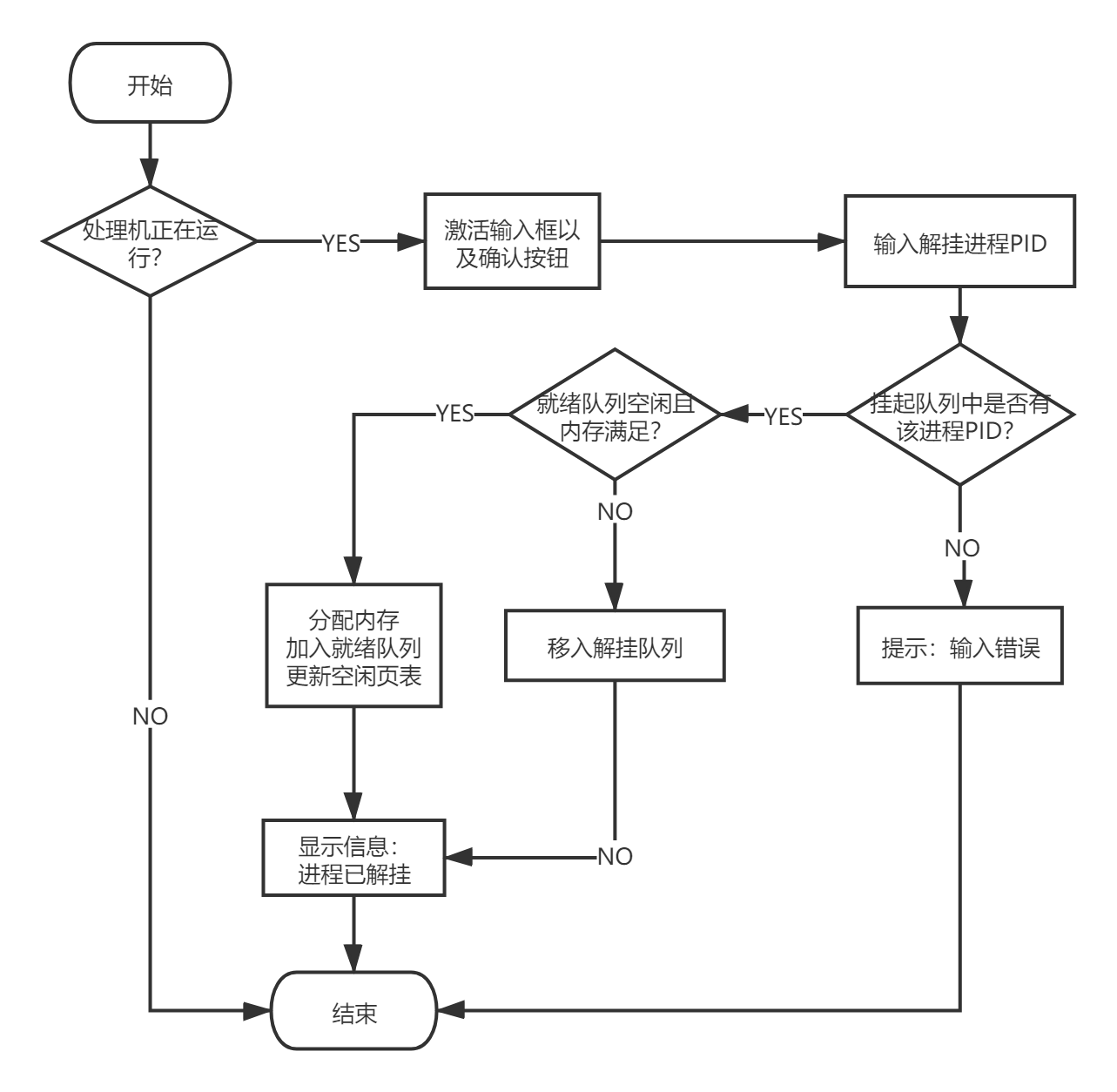
进程挂起函数：

1. **void** MainWindow::on\_suspend\_button\_clicked()
2. {
3. qDebug() << "void MainWindow::on\_suspend\_button\_clicked()";
4. **int** sus\_pid = ui->suspend\_edit->text().toInt();
5. **for**(**int** i = 0;i < ready\_queue.size(); ++i){
6. **if**(sus\_pid == ready\_queue[i].pid){
7. **int** base = ready\_queue[i].base;
8. **int** mem = ready\_queue[i].memory;
9. free\_mem\_link.push\_back(mem\_block(base,mem));
10. sort(free\_mem\_link.begin(),free\_mem\_link.end(),cmp\_mem\_base);
11. merge\_mem\_block(base);
12. auto it = ready\_queue.begin() + i;
13. ready\_queue[i].state = p\_state::p\_suspend;
14. suspend\_queue.push\_back(ready\_queue[i]);
15. ready\_queue.erase(it);
16. QMessageBox::about(NULL,"Accept","Suspend job successfully!");
17. job\_scheduling();
18. **return** ;
19. }
20. }
21. QMessageBox::critical(NULL,"Reject","No matched job!", QMessageBox::Yes, QMessageBox::Yes);
22. **return** ;
23. }

挂起进程一定来源于就绪队列，所以先遍历就绪队列，如果找到要挂起进程，就塞入挂起队列。

**4.进程解挂模块**

进程解挂流程如下：



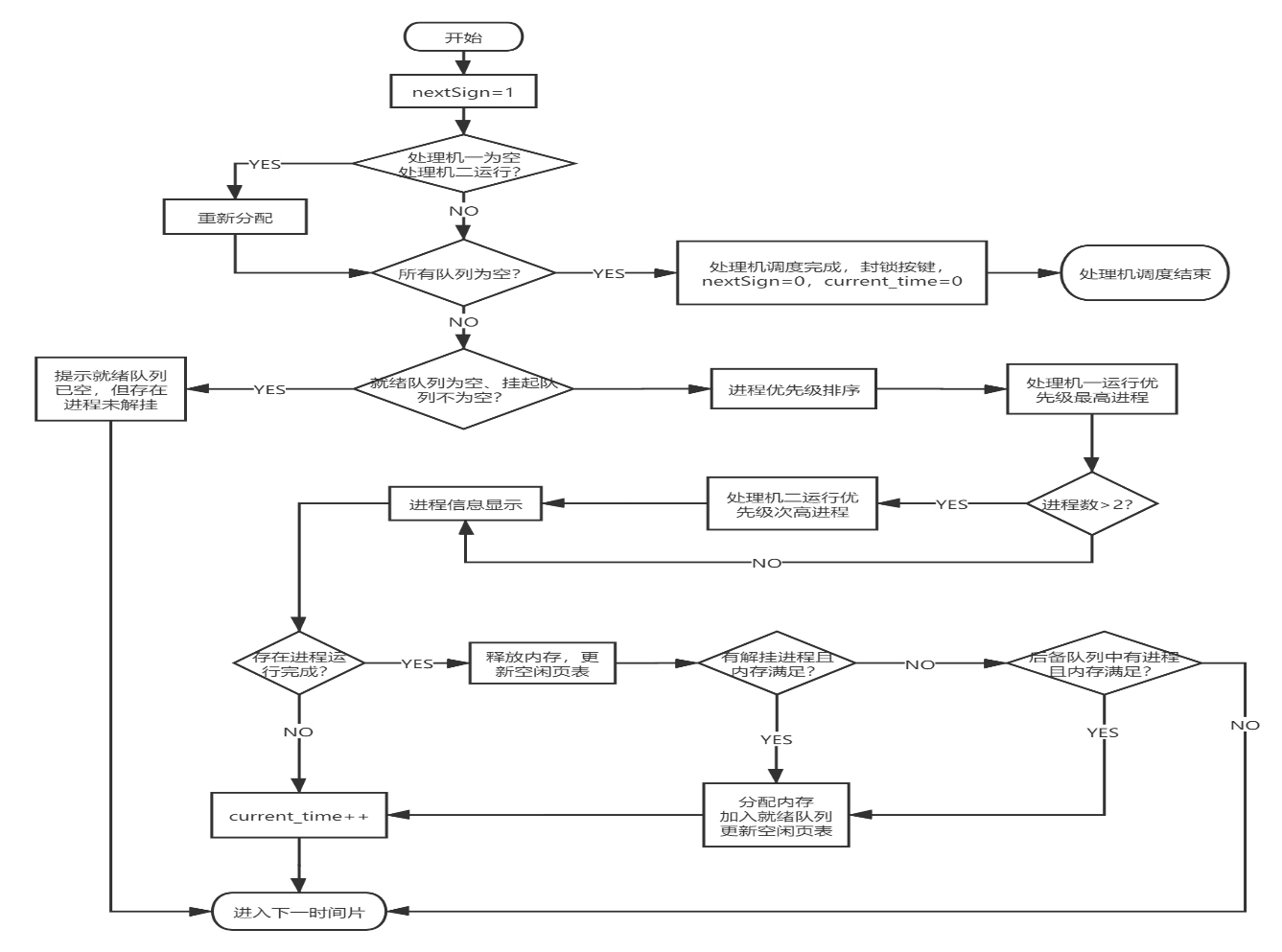
进程解挂函数如下：

1. **void** MainWindow::on\_unsuspend\_button\_clicked()
2. {
3. qDebug() << "void MainWindow::on\_unsuspend\_button\_clicked()";
4. **int** unsus\_pid = ui->unsuspend\_edit->text().toInt();
5. **for**(**int** i = 0; i < suspend\_queue.size(); ++i){
6. **if**(suspend\_queue[i].pid == unsus\_pid){
7. add\_job(suspend\_queue[i]);
8. auto it = suspend\_queue.begin() + i;
9. suspend\_queue.erase(it);
10. QMessageBox::about(NULL,"Accept","Unsuspend job successfully!");
11. **return** ;
12. }
13. }
14. QMessageBox::critical(NULL,"Reject","Can't find this job!", QMessageBox::Yes, QMessageBox::Yes);
15. }

从挂起队列找出进程删除，然后加入就绪队列/后备队列。

**5.处理机运行模块**

处理机运行流程图如下：



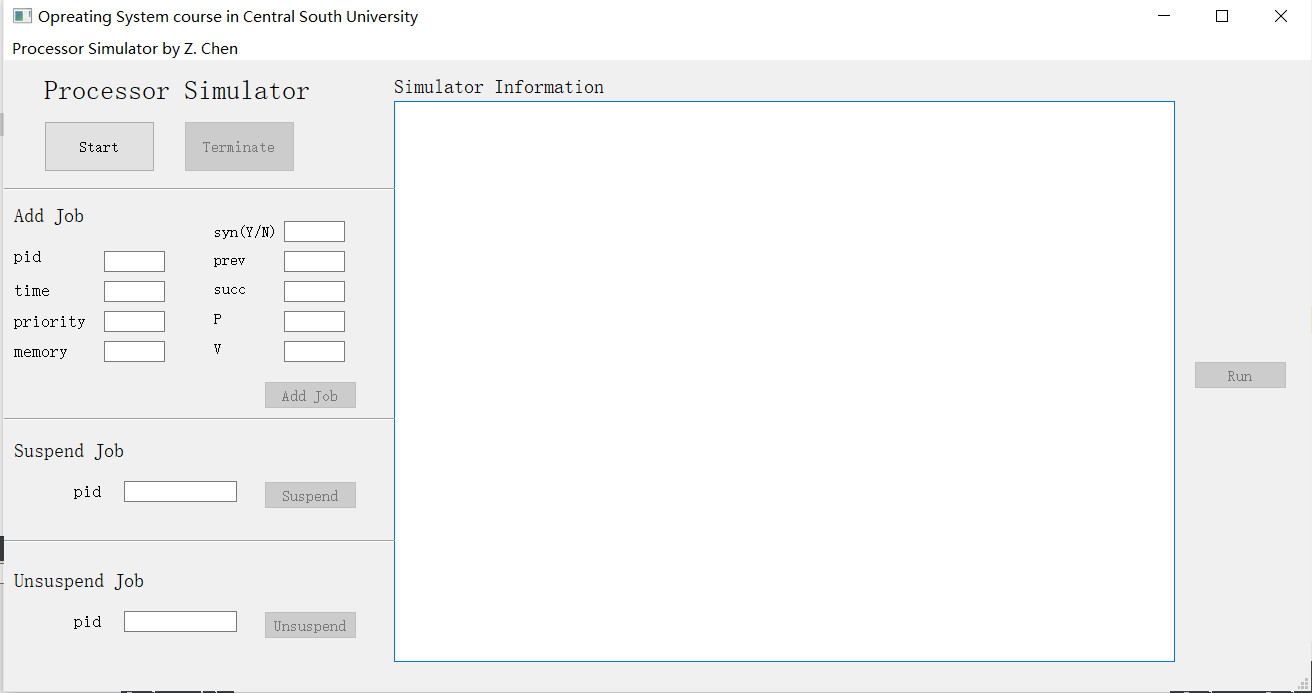
处理机运行函数如下：

1. **void** cpu\_run()
2. {
3. qDebug() << "void cpu\_run()";
4. sort(ready\_queue.begin(),ready\_queue.end(),cmp\_pcb\_pri);
5. **int** j = 0;
6. **for**(**int** i = 0; i < ready\_queue.size(); ++i){
7. **if**(j < max\_processor){
8. **if**(ready\_queue[i].is\_alone){
9. ready\_queue[i].state = p\_state::p\_running;
10. --ready\_queue[i].priority;
11. --ready\_queue[i].time;
12. ++j;
13. }**else**{
14. **if**(ready\_queue[i].bufempty - 1 >= 0){
15. ready\_queue[i].state = p\_state::p\_running;
16. --ready\_queue[i].priority;
17. --ready\_queue[i].time;
18. --ready\_queue[i].bufempty;
19. ++ready\_queue[i].buffull;
20. update\_pv(ready\_queue[i]);
21. ++j;
22. }
23. }
24. }**else**{
25. ready\_queue[i].state = p\_state::p\_ready;
26. }
27. }
28. delete\_job();
29. **return** ;
30. }

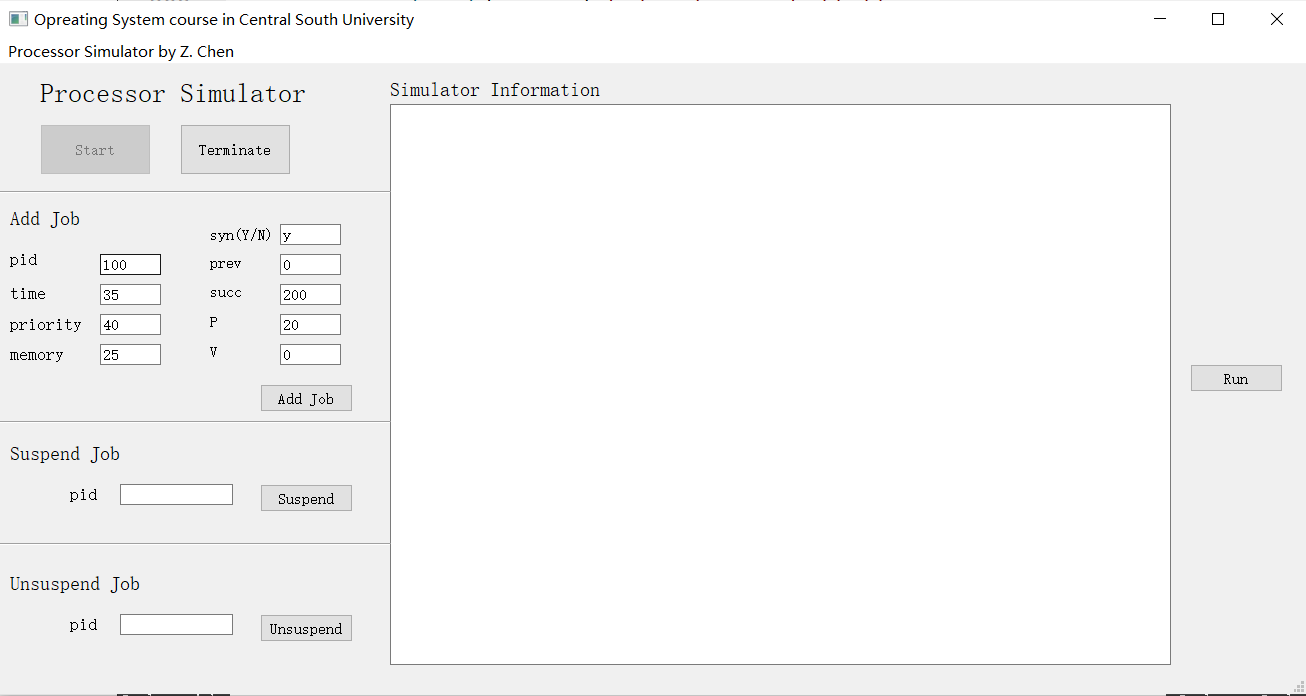
主要功能是从就绪队列里选最高优先级的两个进程，同时减少优先级和时间。

**四、运行结果**

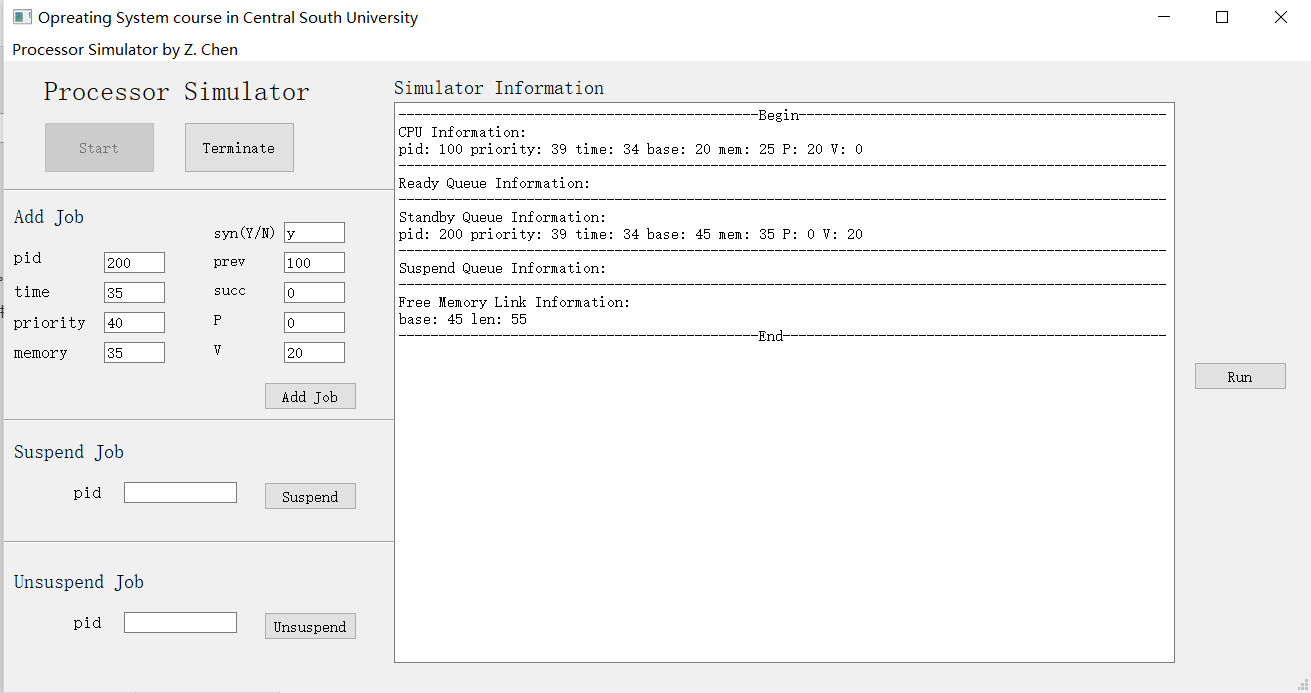
运行界面：



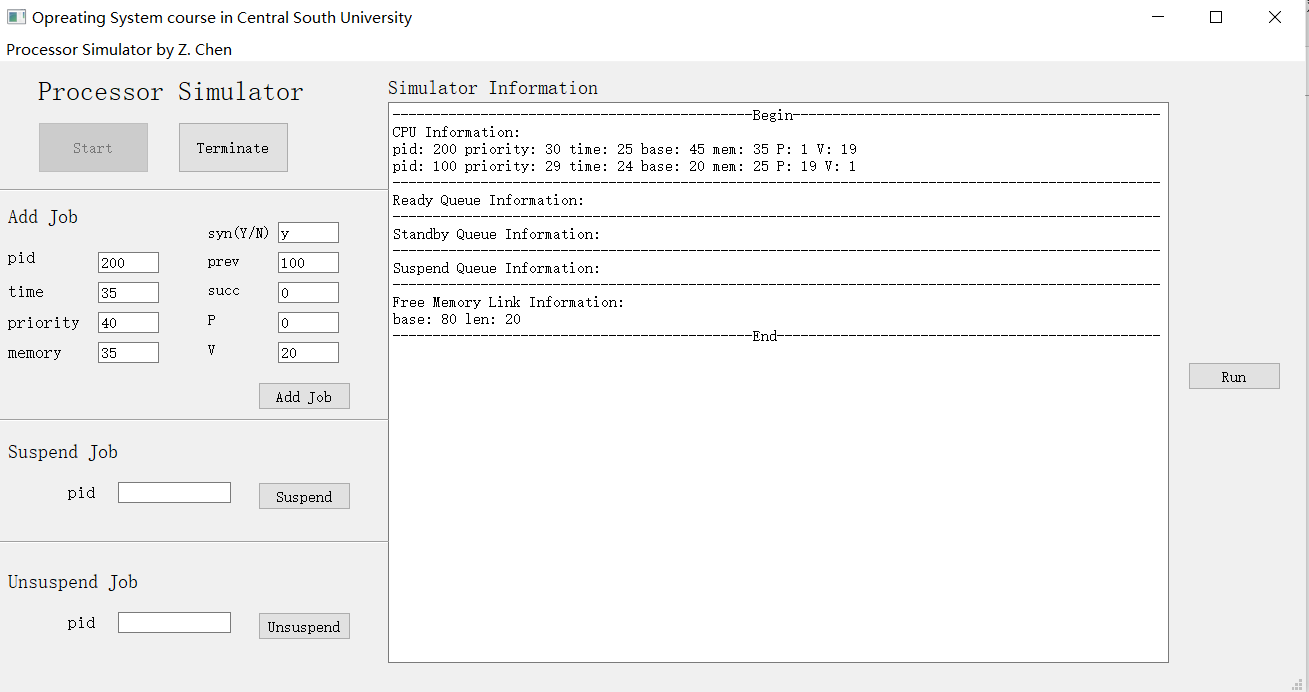
加入作业：



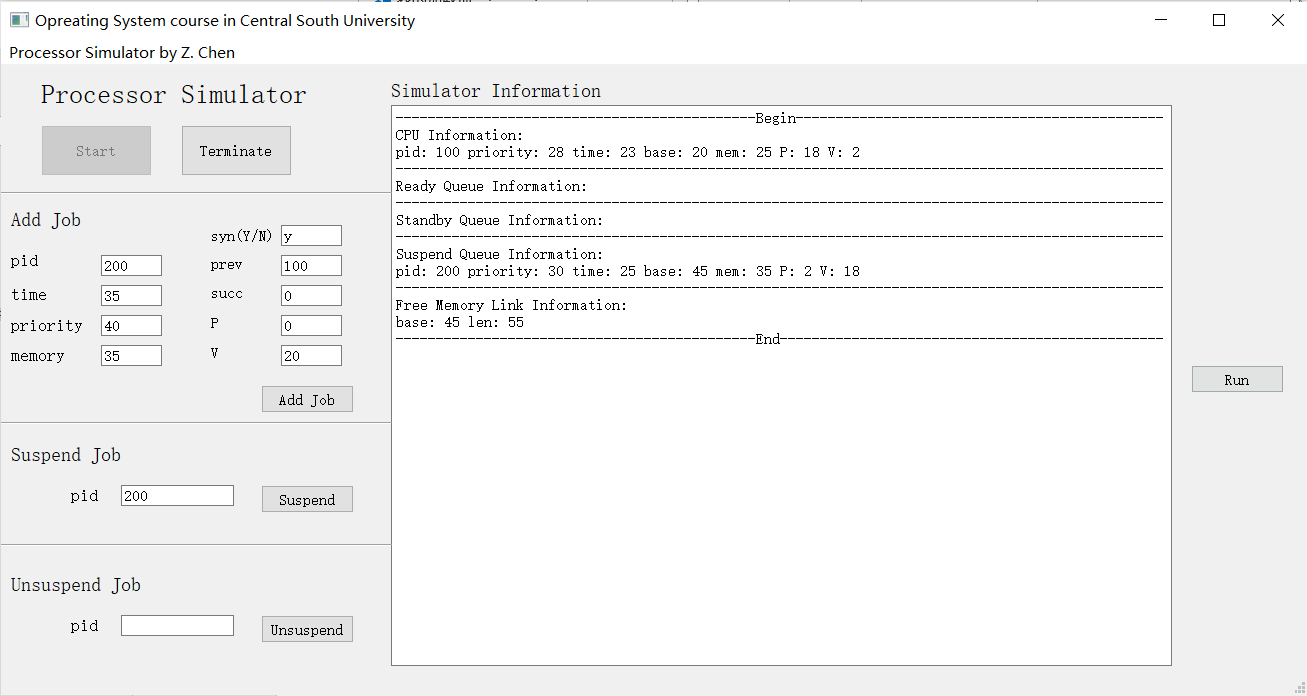
同步进程运行：



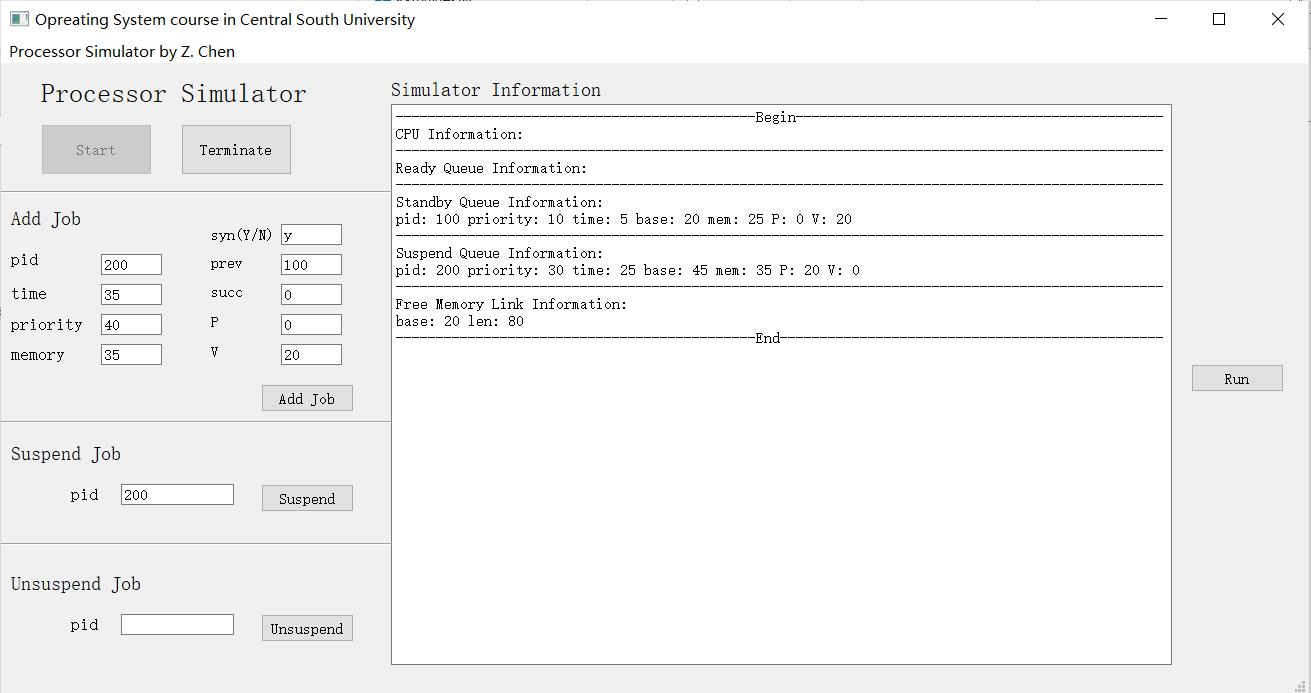
Run：



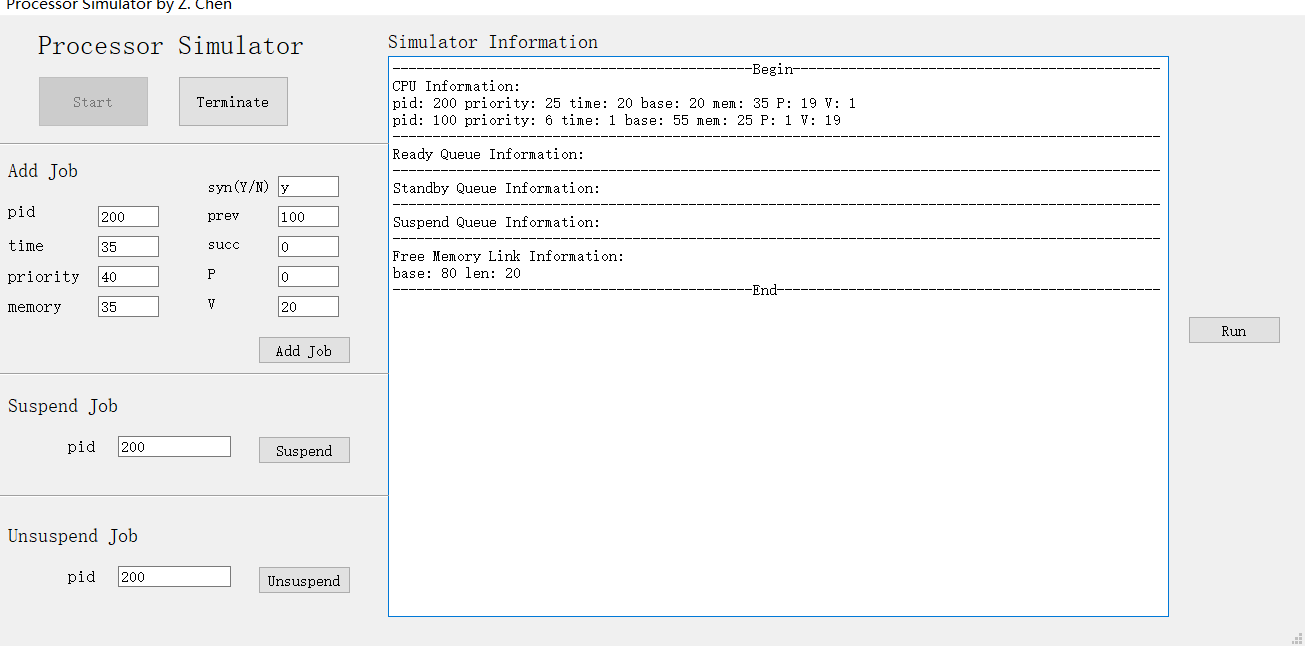
挂起pid：200 ：



pid 100 P信号量消耗净后会加入后备队列



解挂pid 200后再次运行：



运行完毕：

