

# 目录

<b>1</b>	<b>实验名称</b>	<b>3</b>
<b>2</b>	<b>实验目的</b>	<b>3</b>
<b>3</b>	<b>实验内容</b>	<b>3</b>
<b>4</b>	<b>准备知识</b>	<b>4</b>
4.1	进程控制块 (PCB)	4
4.2	进程的基本状态	4
4.3	进程控制主要原语	4
4.4	进程调度算法	5
4.5	内存分配策略及对应的数据结构	5
<b>5</b>	<b>实验设计</b>	<b>5</b>
5.1	程序设计语言	5
5.2	进程调度和内存分配策略选择	5
5.2.1	FCFS 策略	5
5.2.2	可变分区分配策略	5
5.3	数据结构设计	6
5.3.1	进程控制块 PCB	6
5.3.2	自由内存分区表	9
5.3.3	状态码接口	10
5.4	主要类设计	11
5.4.1	内存管理类	11
5.4.2	PCB 管理类	14
5.4.3	进程管理类	16
5.4.4	进程调度类	18
5.4.5	控制台类	23

<b>6</b>	<b>运行结果</b>	<b>29</b>
6.1	初始运行界面 . . . . .	29
6.2	创建进程 . . . . .	29
6.3	PCB、CPU 满负荷运行 . . . . .	30
6.4	撤销进程 . . . . .	30
6.5	阻塞进程 . . . . .	31
6.6	唤醒进程 . . . . .	31
6.7	显示内存状态 . . . . .	32
<b>7</b>	<b>总结与收获</b>	<b>32</b>
<b>8</b>	<b>附录</b>	<b>33</b>
8.1	Properties 包 . . . . .	33
8.1.1	FreeBlock.java . . . . .	33
8.1.2	PCB.java . . . . .	33
8.1.3	States.java . . . . .	33
8.2	Service 包 . . . . .	33
8.2.1	ManageMemory.java . . . . .	33
8.2.2	ManagePCB.java . . . . .	33
8.2.3	ManageProcess.java . . . . .	33
8.2.4	Process.java . . . . .	33
8.2.5	Schduler.java . . . . .	33
8.3	View 包 . . . . .	33
8.3.1	Console.java . . . . .	33

# 1 实验名称

进程管理与内存分配模拟程序

# 2 实验目的

1. 设计编写 OS 进程与内存管理模拟程序，模拟 OS 进程的创建、阻塞、唤醒、撤销等进程控制以及 OS 内存管理方法和过程，加深操作系统进程控制原语主要任务和过程的理解，加深操作系统内存分配的基本策略，加深操作系统以进程为核心的完整架构的理解
2. 提高综合性实验的分析、设计及编程实现能力

# 3 实验内容

设计一个 OS 进程与内存管理模拟程序，要求：（1）程序运行后提供一个交互界面或窗口，允许用户输入以下命令并可以对命令进行解释执行（即执行 OS 的创建进程原语）

- creatproc: 提交作业命令，要求用户提供作业估计运行时长和内存大小需求。该命令的解释执行过程为对该作业创建对应的进程，完成 PCB 建立、存储空间分配等工作
- killproc 进程号: 终止进程命令。该命令的解释执行过程为对进程进行撤销，回收存储空间和 PCB
- iostartproc 进程号: 阻塞进程命令。该命令的解释执行过程为对处于运行状态的进程进行阻塞操作，进程状态转为阻塞状态
- iofinishproc 进程号: 阻塞进程命令。该命令的解释执行过程为对处于阻塞状态的进程进行唤醒操作，进程状态转为就绪状态

- **psproc**: 显示所有进程状态命令。该命令的解释执行过程为显示出所有进程的状态信息，主要包括进程 id，进程状态，存储空间地址
- **mem**: 显示内存空间使用情况信息。该命令的解释执行过程为显示内存空间的占用和空闲情况

(2) 设计思路提示：内存空间可用数组模拟，主要原语模拟实现，包括创建进程，终止进程，阻塞进程，唤醒进程，进程调度等原语。进程调度算法可选择 FCFS、RR、SJF 中任意一种。内存分配可选择可变分区策略或页式内存分配方案中任意一种。

(3) 程序设计语言不限

## 4 准备知识

### 4.1 进程控制块 (PCB)

进程控制块 PCB 包含了有关进程的描述信息、控制信息以及资源信息，是进程动态特征的集中反映。系统根据 PCB 感知进程的存在和通过 PCB 中所包含的各项变量的变化，掌握进程所处的状态以达到控制进程活动的目的

### 4.2 进程的基本状态

包括创建、撤销、阻塞、唤醒、就绪、运行、完成等状态

### 4.3 进程控制主要原语

进程创建原语 `create`;  
进程撤销原语 `kill`;  
进程阻塞原语 `block`;  
进程唤醒原语 `wakeup`;  
进程调度原语 `schedule`

## 4.4 进程调度算法

FCFS、SJF、RR、高响应比优先、优先级调度等

## 4.5 内存分配策略及对应的数据结构

内存分配策略包括固定分区分配、可变分区分配、页式内存分配、段式内存分配、段页式内存分配等

# 5 实验设计

## 5.1 程序设计语言

编程语言：java 17.0.4.1

编译器：IDEA

## 5.2 进程调度和内存分配策略选择

本次实验中，进程调度策略采用了 FCFS（先来先服务）策略，内存分配策略采用了可变分区分配策略

### 5.2.1 FCFS 策略

FCFS（先来先服务）策略是一种非抢占式的调度策略，即一旦 CPU 分配给某进程，那么该进程将一直占用 CPU 直到完成或者发生某种事件而被迫放弃 CPU。FCFS 策略的优点是简单易于实现，但是其缺点也很明显，即平均等待时间较长，不利于提高 CPU 的利用率。

### 5.2.2 可变分区分配策略

可变分区分配策略是指内存空间被分为若干个不相等的分区，每个分区可装入一个作业，作业装入内存时，根据作业的大小，为其分配一个合适

的分区，若无合适的分区，则将其放入外存的后备队列中，等待空闲分区出现。当作业完成后，释放其所占用的分区，使之成为空闲分区，以便接纳新的作业。

## 5.3 数据结构设计

### 5.3.1 进程控制块 PCB

进程控制块 PCB 包含了有关进程的描述信息、控制信息以及资源信息，是进程动态特征的集中反映。系统根据 PCB 感知进程的存在和通过 PCB 中所包含的各项变量的变化，掌握进程所处的状态以达到控制进程活动的目的。

在本实验中，封装了 PCB 类，包含了进程的 id、状态、进程运行时间、内存大小、内存起始地址等信息，同时提供了对应的 get 和 set 方法，重写了 toString 方法，将 PCB 中存储的各类属性以一定的顺序返回，方便后续的输出；同时提供了 stateToString 方法，将状态码转换为状态，方便输出。

PCB 类的定义如下：

```
1      public class PCB { // 进程控制块
2          private int pid; // 进程标识符
3          private String state; // 进程状态
4          private long time; // 进程运行时间
5          private int size; // 进程所需内存大小
6          private int priority; // 进程优先级
7          private boolean mark; // 进程是否被标记
8          private int memoryAddress; // 进程内存地址
9          private boolean Sched; // 进程是否被调度
10
11         // 构造函数
12         public PCB() {
13             mark = false;
14         }
15
16         public int getPid() {
```

```
17         return pid;
18     }
19
20     public void setPid(int pid) {
21         this.pid = pid;
22     }
23
24     public String getState() {
25         return state;
26     }
27
28     public void setState(String state) {
29         this.state = state;
30     }
31
32     public int getPriority() {
33         return priority;
34     }
35
36     public void setPriority(int priority) {
37         this.priority = priority;
38     }
39
40     public boolean getMark() {
41         return mark;
42     }
43
44     public void setMark(boolean mark) {
45         this.mark = mark;
46     }
47
48     public int getMemoryAddress() {
49         return memoryAddress;
50     }
51
```

```

53     public void setMemoryAddress(int memoryAddress) {
        this.memoryAddress = memoryAddress;
    }

55     public boolean isSched() {
57         return Sched;
    }

59     public void setSched(boolean sched) {
61         Sched = sched;
    }

63     public long getTime() {
65         return time;
    }

67     public void setTime(long time) {
69         this.time = time;
    }

71     public int getSize() {
73         return size;
    }

75     public void setSize(int size) {
77         this.size = size;
    }

79     @Override
81     public String toString() {
        return "{" +
83             "pid=" + pid +
            ", state=" + stateToString(state) + '\ ' +
85             ", memoryAddress=" + memoryAddress +
            ", time=" + time/1000 +

```



```

87         '}' + '\n';
89     }

    private String stateToString(String state){//将状态码转换为
        状态
91        switch(state){
93            case "3" : return "Ready";
95            case "4" : return "Block";
97            case "5" : return "Run";
            default: return "Unknow";
        }
    }
}

```

PCB 类定义

### 5.3.2 自由内存分区表

自由内存分区表是指内存中空闲分区的集合，每个空闲分区由起始地址和长度组成。在本实验中，封装了 FreeBlock 类，包含了空闲分区的编号和长度，同时提供了对应的 get 和 set 方法，重写了 toString 方法，将 FreeBlock 中存储的各类属性以一定的顺序返回，方便后续的输出。

FreeBlock 类的定义如下：

```

public class FreeBlock { //空闲块
2    int index;
    int length;
4
    public FreeBlock(int index, int length) {
6        this.index = index;
        this.length = length;
8    }

10    public int getIndex() {
        return index;
    }
}

```

```

12     }

14     public void setIndex(int index) {
        this.index = index;
16     }

18     public int getLength() {
        return length;
20     }

22     public void setLength(int length) {
        this.length = length;
24     }

26     @Override
    public String toString() {
28         return "FreeBlock{" +
            "index=" + index +
30         ", length=" + length +
            '}';
32     }
}

```

FreeBlock 类定义

### 5.3.3 状态码接口

为了方便后续的状态码转换，封装了 States 接口，其中包含了进程的各种状态码，方便后续的调用。

状态码接口的定义如下：

```

1 public interface States {
    String PROCESS_CREATE = "1";// 创建
3    String PROCESS_REVOKE = "2";// 撤销
    String PROCESS_READY = "3";// 就绪

```

```

5      String PROCESS_BLOCK = "4";//阻塞
      String PROCESS_RUN = "5";//运行
7  }

```

### 状态码接口定义

其中，状态码为 1 表示创建，2 表示撤销，3 表示就绪，4 表示阻塞，5 表示运行。

## 5.4 主要类设计

### 5.4.1 内存管理类

内存管理类 ManageMemory 主要用于控制内存的分配和回收，其中包含了模拟总内存大小（本实验中设置 100，从 0 号单元开始计数）、空闲内存列表等成员变量，主要实现函数是分配内存和回收内存。

分配内存函数 allocMemory 的定义如下：

```

1      public static boolean allocateMemory(PCB pcb) { //分配内存，
      可变分区，最佳适应算法
      //对升序排序，挑选最合适空闲块
3      freeBlocksList.sort((o1, o2) -> {
          if (o1.getLength() < o2.getLength()) return -1;
5          return 1;
      });

7      boolean flag = false;
9      for (int i = 0; i < freeBlocksList.size(); i++) {
          FreeBlock a = freeBlocksList.get(i);
11         if (a.getLength() < pcb.getSize()) continue;
          pcb.setMemoryAddress(a.getIndex());
13         if (a.getLength() == pcb.getSize()) {
              freeBlocksList.remove(a);
15         flag = true;
              break;
          }
      }

```

```

17         }
18         a.setIndex(a.getIndex() + pcb.getSize());
19         a.setLength(a.getLength() - pcb.getSize());
20         flag = true;
21         break;
22     }
23     return flag;
24 }

```

### 分配内存函数定义

其中，首先对空闲内存列表进行升序排序，然后遍历空闲内存列表，找到第一个满足条件的空闲内存块，将进程的内存地址设置为该空闲内存块的起始地址：

- 如果该空闲内存块的大小正好等于进程的内存大小，则将该空闲内存块从空闲内存列表中删除
- 否则将该空闲内存块的起始地址设置为该空闲内存块的起始地址加上进程的内存大小，将该空闲内存块的大小设置为该空闲内存块的大小减去进程的内存大小

回收内存函数 `retrieveMemory` 的定义如下：

```

2 public static void retrieveMemory(PCB pcb) { // 释放内存
3     freeBlocksList.sort((o1, o2) -> {
4         if (o1.getIndex() < o2.getIndex()) return -1;
5         return 1;
6     });
7     // 回收内存
8
9     if (freeBlocksList.size() == 0) {
10        freeBlocksList.add(new FreeBlock(pcb.
11            getMemoryAddress(), pcb.getSize()));
12        return;
13    }
14 }

```

```

12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

    int i = 0;
    FreeBlock a;
    for (; i < freeBlocksList.size(); i++) {
        a = freeBlocksList.get(i);
        if (a.getIndex() > pcb.getMemoryAddress()) break;
    }

    System.out.println("第" + i + "个空闲块");
    if (i == 0) {
        FreeBlock back = freeBlocksList.get(i);
        if (back.getIndex() == pcb.getMemoryAddress() + pcb.
            getSize()) {
            back.setIndex(pcb.getMemoryAddress());
            back.setLength(pcb.getSize() + back.getLength())
                ;
        } else {
            freeBlocksList.add(new FreeBlock(pcb.
                getMemoryAddress(), pcb.getSize()));
        }
        return;
    }

    if (i >= freeBlocksList.size()) {
        FreeBlock front = freeBlocksList.get(i - 1);
        if (front.getIndex() + front.getLength() == pcb.
            getMemoryAddress()) {
            front.setLength(front.getLength() + pcb.getSize
                ());
        } else {
            freeBlocksList.add(new FreeBlock(pcb.
                getMemoryAddress(), pcb.getSize()));
        }
        return;
    }

```

```

42     FreeBlock front = freeBlocksList.get(i - 1);
    FreeBlock back = freeBlocksList.get(i);
44     if (front.getIndex() + front.getLength() == pcb.
        getMemoryAddress() && back.getIndex() == pcb.
        getMemoryAddress() + pcb.getSize()) {
        front.setLength(front.getLength() + back.getLength()
            + pcb.getSize());
46        freeBlocksList.remove(back);
    } else if (front.getIndex() + front.getLength() == pcb.
        getMemoryAddress()) {
48        front.setLength(front.getLength() + pcb.getSize());
    } else if (back.getIndex() == pcb.getMemoryAddress() +
        pcb.getSize()) {
50        back.setIndex(pcb.getMemoryAddress());
        back.setLength(pcb.getSize() + back.getLength());
52    } else {
        freeBlocksList.add(new FreeBlock(pcb.
            getMemoryAddress(), pcb.getSize()));
54    }
}

```

### 回收内存函数定义

其中，首先将空闲内存块升序排序，然后将当前释放的内存块插入到空闲内存列表中，然后遍历空闲内存列表，将相邻的空闲内存块合并。

#### 5.4.2 PCB 管理类

PCB 管理类 ManagePCB 主要用于 PCB 各属性的分配和回收，其中包含了 PCB 数组（PCB 数量为 10）、PCB 的 id 等变量，主要用于模拟 PCB 的分配和回收。

PCB 管理类 ManagePCB 的定义如下：

```

1 public class ManagePCB {
    private static int PID = 0;

```

```

3      private static PCB[] pcbs = new PCB[]{new PCB(),new PCB(),
        new PCB(),new PCB(),new PCB(),new PCB(),new PCB(),new PCB(),
        new PCB(),new PCB()};

5

6      public static PCB getFreePCB(){//获取空闲PCB
7          for(PCB a:pcbs){
8              if(a.getMark() == false)return a;
9          }
10         return null;
11     }

12
13     public static PCB getPCB(int pid){
14         for(PCB a:pcbs){
15             if(a.getPid() == pid)return a;
16         }
17         return null;
18     }

19
20     public static void retrievePCB(PCB pcb){
21         pcb.setMark(false);
22     }

23     public static int getPID(){
24         return PID++;
25     }

26
27 }

```

### PCB 管理类定义

其中，getFreePCB 函数用于获取空闲的 PCB，getPCB 函数用于根据进程的 id 获取对应的 PCB，retrievePCB 函数用于回收 PCB，getPID 函数用于获取新的进程 id。在设计中，PCB 是否被占用很巧妙地转化成了 PCB 是否被标记，这样可以很方便地进行 PCB 的分配和回收，同时避免了脏数

据的产生。

### 5.4.3 进程管理类

进程管理类 ManageProcess 包含了进程列表、就绪队列、阻塞队列、运行队列等变量，主要用于存储进程的信息，将进程调度的工作进行切割，单独成类，方便后续的调用。由于存放了三个队列一个列表，因此需要提供对应的插入列表或者队列的函数。

进程管理类 ManageProcess 的定义如下：

```
public class ManageProcess { // 进程队列
2 // 四个 static 数组 保存数据
    private static List<Process> processes = Collections.
        synchronizedList(new ArrayList<>()); // 进程列表
4    private static Queue<Process> ReadyList = new LinkedList<>()
        ; // 就绪队列
    private static Queue<Process> RunList = new LinkedList<>()
        ; // 运行队列
6    private static Queue<Process> BlockList = new LinkedList<>()
        ; // 阻塞队列

8    // 两个插入函数，往 static 数组 插入内容
    public static void insert(Queue<Process> List, Process
        process){
10        List.offer(process);
    }

12    public static void insert(List<Process> List, Process process
        ){
14        List.add(process);
    }

16

18    public static Process getProcess(PCB pcb){
        for(Process a: processes){
```



```

20         if(a.getPcb() == pcb) return a;
21     }
22     return null;
23 }
24 public static Queue<Process> getReadyList() {
25     return ReadyList;
26 }
27
28 public static Queue<Process> getRunList() {
29     return RunList;
30 }
31 public static boolean isFull() { //cpu最多同时处理三个进程
32     if(RunList.size() == 3) return true;
33     return false;
34 }
35
36 public static Queue<Process> getBlockList() {
37     return BlockList;
38 }
39
40 public static List<Process> getProcesses() {
41     return processes;
42 }
43
44 }

```

### 进程管理类定义

其中，insert 函数用于插入进程到对应的列表或者队列中，getProcess 函数用于根据进程的 PCB 获取对应的进程，getReadyList 函数用于获取就绪队列，getRunList 函数用于获取运行队列，isFull 函数用于判断运行队列是否已满（最多同时处理 3 个进程），getBlockList 函数用于获取阻塞队列，getProcesses 函数用于获取进程列表。

#### 5.4.4 进程调度类

本类是整个程序中最为核心的类，主要用于模拟进程的调度，包含了 FCFS 调度算法的实现，同时包含了进程的创建、撤销、阻塞、唤醒等函数，是整个程序的核心。

在创建进程时，首先获取一个空闲的 PCB，然后根据用户输入的进程运行时间和内存大小，设置 PCB 的各项属性，然后将 PCB 插入到进程列表中，然后调用内存分配函数，分配内存，最后将进程插入到就绪队列中。

在撤销进程时，首先根据用户输入的进程 id 获取对应的 PCB，然后根据 PCB 获取对应的进程，然后调用内存回收函数，回收内存，最后将进程从进程列表中删除。

在阻塞进程时，首先根据用户输入的进程 id 获取对应的 PCB，然后根据 PCB 获取对应的进程，然后将进程从相应队列中删除，最后将进程插入到阻塞队列中，更新进程还需要运行的时间。

在唤醒进程时，首先根据用户输入的进程 id 获取对应的 PCB，然后根据 PCB 获取对应的进程，然后将进程从阻塞队列中删除，最后将进程插入到就绪队列中。

在进程调度时，首先判断运行队列是否已满，如果已满，则直接返回，否则从就绪队列中获取一个进程，然后将进程插入到运行队列中，更新进程的状态，最后将进程的状态设置为运行。

进程调度类的定义如下：

```
public class Process { // 进程控制
2   private PCB pcb;

4   private Timer t;
   private ProcessStop task;
6   private long startTime;
   private long pauseTime;
8   private Date date = new Date();

10  public Process(PCB pcb) {
```

```

12         this.pcb = pcb;
13         creat();
14     }
15
16     public void creat() { // 创建进程
17         pcb.setMark(true);
18         pcb.setPid(ManagePCB.getPID());
19         if(ManageMemory.allocateMemory(pcb)) { // 分配内存
20             pcb.setState(States.PROCESS_READY);
21             ManageProcess.insert(ManageProcess.getProcesses(),
22                                 this);
23             ManageProcess.insert(ManageProcess.getReadyList(),
24                                 this);
25         }
26         else {
27             pcb.setState(States.PROCESS_BLOCK);
28             ManageProcess.insert(ManageProcess.getProcesses(),
29                                 this);
30             ManageProcess.insert(ManageProcess.getBlockList(),
31                                 this);
32         }
33     }
34
35     public void kill() { // 撤销进程
36         pcb.setSched(false);
37         if(pcb.getState().equals(States.PROCESS_RUN)) { // 运行状态
38             撤销
39             pcb.setSched(true);
40             ManageProcess.getRunList().remove(ManageProcess.
41                                                 getProcess(pcb));
42             ManageMemory.retrieveMemory(this.pcb);
43         }
44         if(pcb.getState().equals(States.PROCESS_READY)) { // 就绪状态
45             撤销
46             ManageProcess.getReadyList().remove(ManageProcess.

```

```

        getProcess(pcb));
        ManageMemory.retrieveMemory(this.pcb);
40    }
    if(pcb.getState().equals(States.PROCESS_BLOCK)){//阻塞状
        态撤销
42        ManageProcess.getBlockList().remove(ManageProcess.
            getProcess(pcb));
        ManageMemory.retrieveMemory(this.pcb);
44    }
    ManagePCB.retrievePCB(this.pcb);
46    ManageProcess.getProcesses().remove(this);
}

48
public void block(){//阻塞进程
50    //Timer Stop
    date.setTime(System.currentTimeMillis());
52    pauseTime = date.getTime();
    pcb.setTime(pcb.getTime() - (pauseTime - startTime));
54    t.cancel();
    task.cancel();
56    //退出RunList
    ManageProcess.getRunList().remove(this);
58    //——>BlockList
    ManageProcess.insert(getBlockList(), this);
60    //State->Block
    pcb.setState(States.PROCESS_BLOCK);
62    }

64    public void wakeup(){//唤醒进程
        ManageProcess.getBlockList().remove(this);
66        pcb.setState(States.PROCESS_READY);
        ManageProcess.insert(ManageProcess.getReadyList(), this);
68    }

70    public void schedule(){//时间分配

```

```

72         t = new Timer();
       task = new ProcessStop(this);
       date.setTime(System.currentTimeMillis());
74       startTime = date.getTime();
       t.schedule(task, pcb.getTime());
76     }

78     public PCB getPcb() {
       return pcb;
80     }

82     public Timer getT() {
       return t;
84     }

86     public ProcessStop getTask() {
       return task;
88     }

90     public void setT(Timer t) {
       this.t = t;
92     }

94     public void setTask(ProcessStop task) {
       this.task = task;
96     }

98     public long getStartTime() {
       return startTime;
100    }

102     public void setStartTime(long startTime) {
       this.startTime = startTime;
104    }

```

```

106     public long getPauseTime() {
108         return pauseTime;
109     }
110
111     public void setPauseTime(long pauseTime) {
112         this.pauseTime = pauseTime;
113     }
114
115     @Override
116     public String toString() {
117         return pcb.toString();
118     }
119 }

```

### 进程调度类定义

其中，ProcessStop 类用于模拟运行进程的停止，ProcessStop 类的定义如下：

```

2     class ProcessStop extends TimerTask {
3         private Process process;
4
5         public ProcessStop(Process process) {
6             this.process = process;
7         }
8
9         @Override
10        public void run() {
11            process.kill();
12        }
13    }

```

### ProcessStop 类定义

### 5.4.5 控制台类

控制台类 Console 主要用于模拟控制台，包含了主函数和控制台的各种命令，主要用于模拟用户的输入，方便后续的调用。

在控制台界面的设计中，使用了 Java swing 编写了一个简单的控制台界面，方便用户的输入和输出，同时使用了 JTextArea 组件，方便用户的输出。

窗体实现的代码如下：

```
1 public static void main(String[] args) {
2     JFrame frame = new JFrame("进程管理与内存分配模拟程序");
3     outputTextArea.setFont(new Font("宋体", Font.BOLD, 20));
4     outputTextArea.setEditable(false);
5     outputTextArea.setBackground(Color.cyan);
6
7     inputTextField.setBackground(Color.lightGray);
8     JScrollPane scrollPane = new JScrollPane(outputTextArea);
9     ;
10    scrollPane.setVerticalScrollBarPolicy(JScrollPane.
11        VERTICAL_SCROLLBAR_ALWAYS);
12
13    frame.getContentPane().setLayout(new BorderLayout());
14    frame.getContentPane().add(scrollPane, BorderLayout.
15        CENTER);
16    frame.getContentPane().add(inputTextField, BorderLayout.
17        SOUTH);
18
19    outputTextArea.append("—————进程管理与内存分配
20        模拟程序—————\n");
21    outputTextArea.append("—————createproc 时间 内
22        存大小:提交作业命令——\n—————killproc 进程号
23        :终止进程—————\n—————iostartproc
24        进程号:阻塞进程命令—————\n—————
25        iofinishproc 进程号:阻塞进程唤醒命令——\n
26        —————psproc:显示所有进程状态命令
```

```

17         \nmem: 显示内存空间使用情况
18         信息\n\n");
19
20         inputTextField.addActionListener(new ActionListener() {
21             @Override
22             public void actionPerformed(ActionEvent e) {
23                 String content = inputTextField.getText();
24                 inputTextField.setText("");
25                 process(content);
26             }
27         });
28
29         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30         frame.setBounds(500, 200, 800, 600);
31         frame.setVisible(true);
32         new Scheduler().start();
33     }

```

### 窗体实现代码

在控制台类中，主要实现了 process 函数，用于解析用户的输入，也就是说输入相应的原语后，对消息进行分割，对原语进行匹配解析，然后调用相应的函数，方便后续的调用。

键入了 createproc 命令，则对命令进行切割，分别解析出进程运行时间和内存大小，然后调用创建进程函数，创建进程。

```

1     if (command.startsWith("createproc ")) {
2         // 得到进程估计运行时间和所需内存大小
3         String[] contents = command.split(" ");
4         // 获取空闲PCB
5         PCB pcb = ManagePCB.getFreePCB();
6         // 失败的情况
7         if (pcb == null) {
8             outputTextArea.append("创建进程失败，没有空闲的
9             PCB" + "\n");
10            return;

```



```

    }
11    // 成功就创建新进程
    pcb.setTime(Integer.parseInt(contents[1]) * 1000);
13    pcb.setSize(Integer.parseInt(contents[2]));
    new Process(pcb);
15    outputTextArea.append(ManageProcess.getProcesses().
        toString() + "\n");
}

```

键入 createproc

键入了 killproc 命令，则对命令进行切割，分别解析出进程 id，然后调用撤销进程函数，撤销进程。

```

    else if (command.startsWith("killproc ")) {
2        String[] contents = command.split(" ");
        PCB pcb = ManagePCB.getPCB(Integer.parseInt(contents[1])
            );
4        if (pcb == null) {
            outputTextArea.append("该进程不存在" + "\n");
6            return;
        }
8        ManageProcess.getProcess(pcb).kill();
        outputTextArea.append(ManageProcess.getProcesses().
            toString() + "\n");
10    }
}

```

键入 killproc

键入了 iostartproc 命令，则对命令进行切割，分别解析出进程 id，然后调用阻塞进程函数，阻塞进程。

```

    else if (command.startsWith("iostartproc ")) {
2        String[] contents = command.split(" ");
        PCB pcb = ManagePCB.getPCB(Integer.parseInt(contents[1])
            );
4        if (pcb == null) {

```

```

        outputTextArea.append("该进程不存在" + "\n");
        return;
    }
    // 判断是否被调度
    if (ManageProcess.getProcess(pcb) == null) {
        outputTextArea.append("该进程未被调度，阻塞操作失败"
            + "\n");
        return;
    }
    ManageProcess.getProcess(pcb).block();
    outputTextArea.append(ManageProcess.getProcesses().
        toString() + "\n");
}

```

键入 iostartproc

键入了 iofinishproc 命令，则对命令进行切割，分别解析出进程 id，然后调用唤醒进程函数，唤醒进程。

```

1    else if (command.startsWith("iofinishproc ")) {
        String[] contents = command.split(" ");
3        PCB pcb = ManagePCB.getPCB(Integer.parseInt(contents[1])
            );
        if (pcb == null) {
5            outputTextArea.append("该进程不存在" + "\n");
            return;
6        }
        ManageProcess.getProcess(pcb).wakeup();
7        outputTextArea.append(ManageProcess.getProcesses().
8            toString() + "\n");
9    }
}

```

键入 iofinishproc

键入了 psproc 命令，则调用显示进程状态函数，显示进程状态。

```

else if (command.equals("psproc")) {

```

```

2      outputTextArea.append(ManageProcess.getProcesses().
        toString() + "\n");
    }

```

键入 psproc

键入了 mem 命令，则调用显示内存状态函数，显示内存状态。

```

1      else if (command.equals("mem")) {
2          ManageMemory.getFreeBlocksList().sort((o1, o2) -> {
3              if (o1.getIndex() < o2.getIndex()) return -1;
4              return 1;
5          });
6
7          ManageProcess.getProcesses().sort((o1, o2) -> {
8              if (o1.getPcb().getMemoryAddress() < o2.getPcb().
9                  getMemoryAddress()) return -1;
10             return 1;
11         });
12
13         int i = 0, j = 0, k = ManageProcess.getProcesses().size
14             (), t = ManageMemory.getFreeBlocksList().size();
15         List<Process> processes = ManageProcess.getProcesses();
16         List<FreeBlock> freeBlocksList = ManageMemory.
17             getFreeBlocksList();
18         while (i < k && j < t) {
19             if (processes.get(i).getPcb().getMemoryAddress() <
                freeBlocksList.get(j).getIndex()) {
                outputTextArea.append(processes.get(i).getPcb().
                    getMemoryAddress() + " — " + (processes.get
                        (i).getPcb().getMemoryAddress() + processes.
                            get(i).getPcb().getSize() - 1) + " 分配给" +
                                processes.get(i).getPcb().getPid() + "号进程
                                    " + "\n");
                i++;
            } else {

```

```

21         outputTextArea.append( freeBlocksList.get(j).
           getIndex() + " — " + (freeBlocksList.get(j)
           .getIndex() + freeBlocksList.get(j).getLength
           () - 1) + "空闲" + "\n");
           j++;
23     }
25     if (i < k) {
27         while (i < k) {
           outputTextArea.append( processes.get(i).getPcb().
           getMemoryAddress() + " — " + (processes.get
           (i).getPcb().getMemoryAddress() + processes.
           get(i).getPcb().getSize() - 1) + " 分配给" +
           processes.get(i).getPcb().getPid() + "号进程
           " + "\n");
           i++;
29     }
31     if (j < t) {
           while (j < t) {
           outputTextArea.append( freeBlocksList.get(j).
           getIndex() + " — " + (freeBlocksList.get(j)
           .getIndex() + freeBlocksList.get(j).getLength
           () - 1) + " 空闲" + "\n");
           j++;
33     }
35 }
}

```

键入 mem

## 6 运行结果

### 6.1 初始运行界面

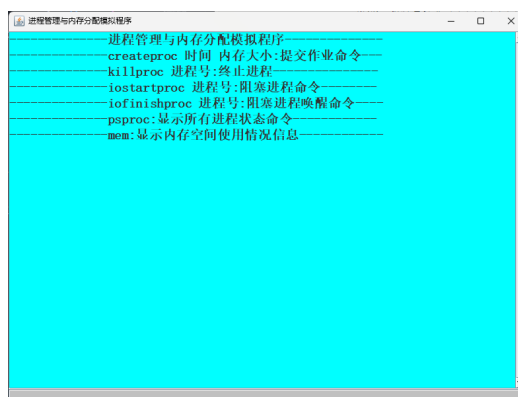


图 1: 初始运行界面

### 6.2 创建进程

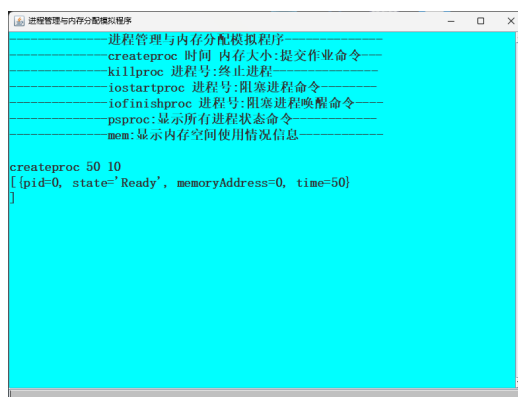


图 2: 创建进程

## 6.3 PCB、CPU 满负荷运行

```
进程管理与内存分配模拟程序
(pid=4, state='Ready', memoryAddress=30, time=100)
(pid=5, state='Ready', memoryAddress=35, time=100)
(pid=6, state='Ready', memoryAddress=0, time=100)
(pid=7, state='Ready', memoryAddress=5, time=100)
(pid=8, state='Ready', memoryAddress=40, time=100)
(pid=9, state='Ready', memoryAddress=45, time=100)
]
createproc 100 5
[(pid=1, state='Run', memoryAddress=10, time=100)
(pid=2, state='Run', memoryAddress=20, time=100)
(pid=3, state='Run', memoryAddress=25, time=100)
(pid=4, state='Ready', memoryAddress=30, time=100)
(pid=5, state='Ready', memoryAddress=35, time=100)
(pid=6, state='Ready', memoryAddress=0, time=100)
(pid=7, state='Ready', memoryAddress=5, time=100)
(pid=8, state='Ready', memoryAddress=40, time=100)
(pid=9, state='Ready', memoryAddress=45, time=100)
(pid=10, state='Ready', memoryAddress=50, time=100)
]
createproc 100 5
创建进程失败，没有空闲的PCB
```

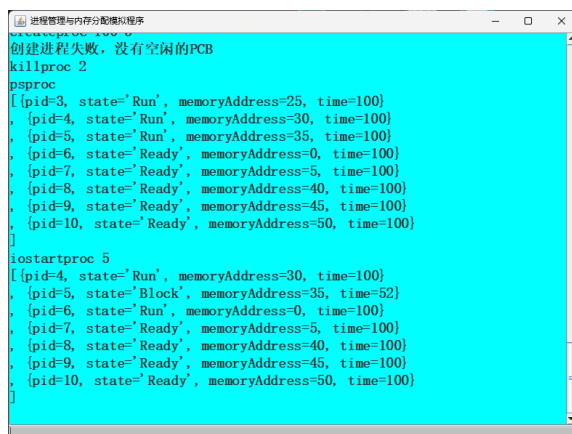
图 3: PCB、CPU 满负荷运行

## 6.4 撤销进程

```
进程管理与内存分配模拟程序
(pid=4, state='Ready', memoryAddress=30, time=100)
(pid=5, state='Ready', memoryAddress=35, time=100)
(pid=6, state='Ready', memoryAddress=0, time=100)
(pid=7, state='Ready', memoryAddress=5, time=100)
(pid=8, state='Ready', memoryAddress=40, time=100)
(pid=9, state='Ready', memoryAddress=45, time=100)
(pid=10, state='Ready', memoryAddress=50, time=100)
]
createproc 100 5
创建进程失败，没有空闲的PCB
killproc 2
psproc
[(pid=3, state='Run', memoryAddress=25, time=100)
(pid=4, state='Run', memoryAddress=30, time=100)
(pid=5, state='Run', memoryAddress=35, time=100)
(pid=6, state='Ready', memoryAddress=0, time=100)
(pid=7, state='Ready', memoryAddress=5, time=100)
(pid=8, state='Ready', memoryAddress=40, time=100)
(pid=9, state='Ready', memoryAddress=45, time=100)
(pid=10, state='Ready', memoryAddress=50, time=100)
]
```

图 4: 撤销进程

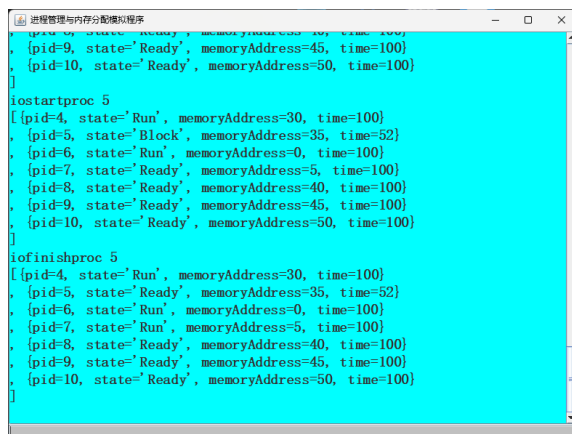
## 6.5 阻塞进程



```
进程管理与内存分配模拟程序
创建进程失败，没有空闲的PCB
killproc 2
psproc
[[pid=3, state='Run', memoryAddress=25, time=100]
, [pid=4, state='Run', memoryAddress=30, time=100]
, [pid=5, state='Run', memoryAddress=35, time=100]
, [pid=6, state='Ready', memoryAddress=0, time=100]
, [pid=7, state='Ready', memoryAddress=5, time=100]
, [pid=8, state='Ready', memoryAddress=40, time=100]
, [pid=9, state='Ready', memoryAddress=45, time=100]
, [pid=10, state='Ready', memoryAddress=50, time=100]
]
iostartproc 5
[[pid=4, state='Run', memoryAddress=30, time=100]
, [pid=5, state='Block', memoryAddress=35, time=52]
, [pid=6, state='Run', memoryAddress=0, time=100]
, [pid=7, state='Ready', memoryAddress=5, time=100]
, [pid=8, state='Ready', memoryAddress=40, time=100]
, [pid=9, state='Ready', memoryAddress=45, time=100]
, [pid=10, state='Ready', memoryAddress=50, time=100]
]
```

图 5: 阻塞进程

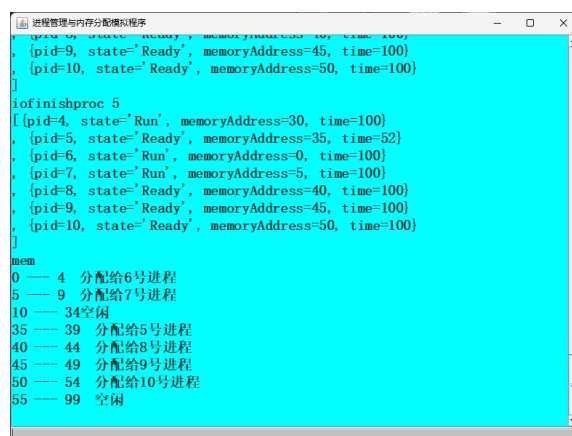
## 6.6 唤醒进程



```
进程管理与内存分配模拟程序
[[pid=9, state='Ready', memoryAddress=45, time=100]
, [pid=10, state='Ready', memoryAddress=50, time=100]
]
iostartproc 5
[[pid=4, state='Run', memoryAddress=30, time=100]
, [pid=5, state='Block', memoryAddress=35, time=52]
, [pid=6, state='Run', memoryAddress=0, time=100]
, [pid=7, state='Ready', memoryAddress=5, time=100]
, [pid=8, state='Ready', memoryAddress=40, time=100]
, [pid=9, state='Ready', memoryAddress=45, time=100]
, [pid=10, state='Ready', memoryAddress=50, time=100]
]
iofinishproc 5
[[pid=4, state='Run', memoryAddress=30, time=100]
, [pid=5, state='Ready', memoryAddress=35, time=52]
, [pid=6, state='Run', memoryAddress=0, time=100]
, [pid=7, state='Run', memoryAddress=5, time=100]
, [pid=8, state='Ready', memoryAddress=40, time=100]
, [pid=9, state='Ready', memoryAddress=45, time=100]
, [pid=10, state='Ready', memoryAddress=50, time=100]
]
```

图 6: 唤醒进程

## 6.7 显示内存状态



```
进程管理与内存分配模拟程序
(pid=9, state='Ready', memoryAddress=45, time=100)
(pid=10, state='Ready', memoryAddress=50, time=100)
]
iofinishproc 5
[[pid=4, state='Run', memoryAddress=30, time=100]
(pid=5, state='Ready', memoryAddress=35, time=52]
(pid=6, state='Run', memoryAddress=0, time=100]
(pid=7, state='Run', memoryAddress=5, time=100]
(pid=8, state='Ready', memoryAddress=40, time=100]
(pid=9, state='Ready', memoryAddress=45, time=100]
(pid=10, state='Ready', memoryAddress=50, time=100]
]
mem
0 --- 4  分配给6号进程
5 --- 9  分配给7号进程
10 --- 34空闲
35 --- 39 分配给5号进程
40 --- 44 分配给8号进程
45 --- 49 分配给9号进程
50 --- 54 分配给10号进程
55 --- 99 空闲
```

图 7: 显示内存状态

## 7 总结与收获

通过本次实验，我对进程管理和内存管理有了更深刻的理解。

在进程调度方面，本次实验使用了 FCFS 调度算法，即先来先服务调度算法，这种调度算法的优点是简单，容易实现，但是缺点是平均等待时间较长，不利于提高 CPU 的利用率，同时也不利于提高系统的吞吐量。实现起来比较简单，只需要将就绪队列中的第一个进程调度到运行队列中即可。逻辑清晰，代码简单。在实验的时候能够很好地复现出来，但是在实际的操作系统中，这种调度算法的使用场景并不多，由于本人水平有限，暂时还没有了解其他的调度算法的代码实现，希望在以后的学习中能够学习到其他的调度算法，提高自己的水平。

在内存管理方面，本次实验使用了可变分区的内存管理方式，即每个进程的内存大小不固定，根据进程的内存大小分配对应的内存块，这种内存管理方式的优点是能够充分利用内存空间，缺点是容易产生内存碎片，同时也不利于提高内存的利用率。实现起来比较简单，只需要将空闲内存块



按照大小进行排序，然后遍历空闲内存块，找到第一个满足条件的空闲内存块即可。实际内存管理中，多采用段页式内存管理方式，这种内存管理方式能够很好地解决内存碎片的问题，同时也能够提高内存的利用率，但是实现起来比较复杂，需要考虑的条件很多，还需要继续学习，提高编程技巧

总的来说，本次实验收获很大，对进程管理和内存管理有了更深刻的理解，同时也提高了编程能力，希望在以后的学习中能够继续提高自己的水平，为以后的学习打下坚实的基础。

## 8 附录

### 8.1 Properties 包

#### 8.1.1 FreeBlock.java

#### 8.1.2 PCB.java

#### 8.1.3 States.java

### 8.2 Service 包

#### 8.2.1 ManageMemory.java

#### 8.2.2 ManagePCB.java

#### 8.2.3 ManageProcess.java

#### 8.2.4 Process.java

#### 8.2.5 Schduler.java

### 8.3 View 包

#### 8.3.1 Console.java