中南大學
CENTRAL SOUTH UNIVERSITY

进程控制和内存管理模拟程序

学生姓名 孔德彬

学生学号 8208181404

指导教师 沈海澜

专业班级 计科 1801

完成日期 2020.6.10

计算机学院

目录

1. 实验目的

（1）掌握进程的创建、阻塞、唤醒、撤销等进程控制原语。

（2）掌握进程的处理机调度的过程和算法。

（3）掌握内存分配策略。

2. 实验内容

进程控制和内存管理模拟实现。

3. 实验要求

（1）设计一个允许 n 个进程并发运行的 OS 进程管理模拟程序，模拟实现创建新进程原语、阻塞进程原语、唤醒进程原语，终止进程原语、调度进程原语等功能；每个进程用一个 PCB 表示，其内容可根据具体情况设置。进程调度算法可选择 FCFS、时间片轮转或其他任意一种。内存分配可采用可变分区策略+最佳适应算法（或页式等其他内存分配方案，自选一种），进程创建时需为进程分配内存空间，进程终止时需要回收进程的内存空间。

（2）程序在运行过程中能显示或打印各进程的状态及有关参数的变化情况，以便观察诸进程的运行过程及系统的管理过程。

4. 实验分析与设计

4.1. 设计方案

此项目采用的进程调度算法是 FCFS，内存分配方式为可变分区策略和最佳适应算法，主要包括了四个类，分别是进程 PCB 类，地址 Address 类，界面 Process 类，功能实现 ChooseFun 类。接下来将会对每一个类进行介绍。

- PCB

该类中主要包括了五个进程的相关信息，分别是进程编号 id、进程状态 status、所需最大内存资源 maxRam、运行时间 time、具体存储位置 addressPCB，主要函数主要是 get 和 set 类函数，用来获取或修改进程信息。

- Address

该类主要模拟内存的分配，其中的变量有代表内存起始地址的 start、代表进程终止的 end、代表进程长度的 length、指向下一个内存单元的指针 next。在其构造函数中，传入进程的 start 和 end，通过 length＝end－start＋1 更新 length 值，

讲 next 默认设为空值 null。

- Process

该类主要用与显示界面，继承了 JFrame，实现了 Runable 接口。为了更好的观察，此项目加入了 GUI 开发，在 Process 类中，建立自己的进程，用来监控进程的变换，将所有的进程信息，显示在界面上，并每隔一秒钟对正在运行的进程时间减一，若减为 0，则将该进程移除，并选取下一个进程进行运行。同时，建立 ChooseFun 进程，用于接收用户的操作指令。

- ChooseFun

该类重要用户实现用户功能，其实现了 Runnable 接口，其中的主要变量有 allProcess，processNum、addressHead。其中通过 HashMap 来存储每个进程的 PCB，并将其赋值给 allPrecess，计算所有的进程值，将其赋值给 processNum、在该进程起始时，创建以 0 为起始，以 199 为终止的内存空间，赋值给 addressHead，用于存储所有进程的同地址。

该类涉及到的函数主要包括 run、create、block、wake、stop、getMaxKey、getNextKey、getAddress、delAddress、updateAddress 和 show 几个函数，接下来将会对这几个函数进行介绍。

run 函数是 Runnable 中要实现的函数，其主要功能就是接收用户输入的语句命令，通过 BufferedReader.readline 的方法来读入，并通过 switch 语句来选择相对应的函数功能，若操作失败或输入语句无效，将会给予相对应的提示。

create 函数主要是用来建立新的进程。在运行时，将会提示用户输入要建立进程的 id、maxRam 和 time。为即将创建的进程分配 address，若为事先设置好的出错值，则返回 false，并提示用户进程分配错误。判断是否有正在运行的进程，若没有，则将创建的进程 status 设为 running 状态，若存在有正在运行的进程，则将进程状态设置为 ready 状态。最后将其加入到 allProcess 中，同时提示用户创建进程成功，显示创建进程信息。

block 函数用来模拟正在模拟正在运行的进程遇到 I/O 操作时进入状态。主要是遍历 allProcess 中所有的进程，找到状态为 running 状态的进程，并将其状态改为 waiting，若该进程不为最后一个进程，则将其下一个状态为 ready 的进程改为 running 状态，否则，只是将其改称为 waiting 状态。

wake 函数，主要是唤醒一个正在 waiting 状态的进程，提示用户输入要唤醒进程的 id，判断其是否为 waiting 状态。如果该进程是最后一个进程，则直接进入 running 状态，若不是，则将其状态改称为-1，并复制该进程，插入到 allProcess 的最后，状态改为 ready。

stop 函数为终止进程函数，提示用户输入想要终止的进程 id，将其状态设为-1，并将拍 processNum 减一，并归还该进程的所占用的内存值。

getMaxKey 函数返回 allProcess 的最大 key 值，其主要用在 wake 唤醒原语中，要复制进程信息到最末端。

getNextKey 函数用来返回传入参数的下一个 key 值，主要用于该进程结束，要选取下一个有效进程运行时。

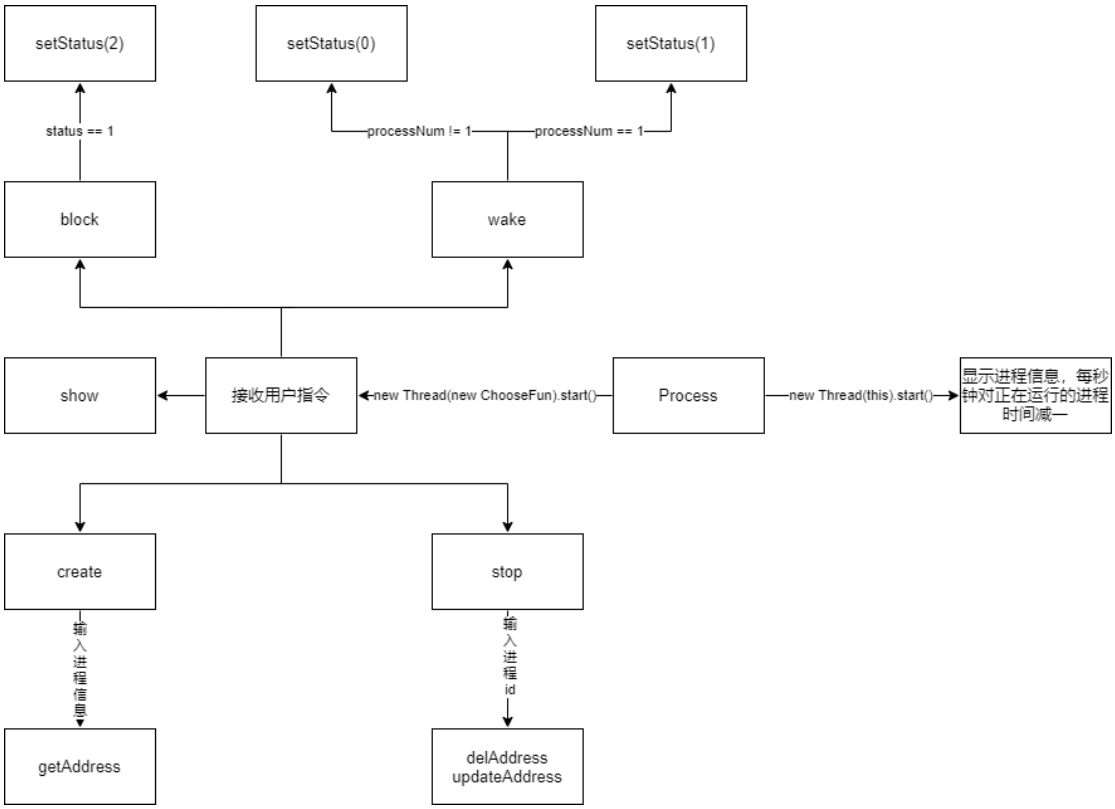
getAddress 函数主要用来为进程分配地址空间，先建立一个 start 和 end 均为-1 的地址空间作为错误符号，接着从 addressHead 开始，遍历所有的内存块，如果内存块的 length 大于 maxRam 的话，就分配对应的内存给目标进程，若遍历所有内存块均为找到合适的地址，则显示内存不足，创建进程失败，并返回错误符号。

delAddess 函数主要用来归还内存地质单元，一共有三种情况，待归还的地址单元在最前面，待归还的地址段元在中间和待归还的地址单元在最后面，将要删除的地址单元放到对应的位置，并调用 updateAddress 函数，将相邻的地址单元进行合并。

updateAddress 函数用来合并相邻的地址单元，从 addressHead 开始遍历，如果当前地址单元的 end 和他 next 的地址单元的 start＋1 相等的时候，则将两个地址单元进行合并处理。

show 函数主要用来显示当前状态下，所有可用的内存块。

4.2. 项目流程图



5. 实验运行结果与相关代码

5.1. 相关代码

5.1.1. PCB

```
public class PCB{
    public int id;                  // 进程 id
    public int status;              // 进程状态
    public int maxRam;              // 所需资源
    public int time;                // 运行时间
    public Address addressPCB;      // 具体存储位置

    /*0-ready
    * 1-running
    * 2-waiting*/
```

```
PCB(int _id, int _status, int _maxRam, int _time, Address _addressPCB){
        this.id = _id;
        this.status = _status;
        this.maxRam = _maxRam;
        this.time = _time;
        this.addressPCB = _addressPCB;
}

public int getId(){
        return this.id;
}

public int getStatus(){
        return this.status;
}

public int getMaxRam(){
        return this.maxRam;
}

public int getTime(){
        return this.time;
}

public Address getAddressPCB() { return this.addressPCB; }

public void setStatus(int _status){
        this.status = _status;
```

```
    }

    public void setTime(int _time){
        this.time = _time;
    }
}
```

5.1.2. Address

```
public class Address {
    public int start;
    public int end;
    public int length;
    public Address next = null;

    Address(int _start, int _end){
        this.start = _start;
        this.end = _end;
        this.length = end - start + 1;
    }
}
```

5.1.3. Process

```
public class Process extends JFrame implements Runnable{
    private JTextArea jTextArea = new JTextArea();
    private ChooseFun chooseFun = new ChooseFun();

    Process(){
        this.setTitle("进程管理器");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
        this.setSize(800, 600);
```

```
        this.add(jTextArea);
        this.setLocationRelativeTo(null);
        jTextArea.setFont(new Font("Serif", 0, 15));

        new Thread(chooseFun).start();
        new Thread(this).start();
    }

    @Override
    public void run() {
        while(true){
            try {
                Thread.sleep(1000);
                jTextArea.setText("");
                for(Integer key : chooseFun.allProcess.keySet()){
                    // 若为-1，则表示进程已经完成，或进程已经被 stop 掉了
                    if(chooseFun.allProcess.get(key).getStatus() != -1){
                        jTextArea.append(" id: "
                                + chooseFun.allProcess.get(key).getId()
                                + " \t| status: ");
                        /*输出各个状态下数字对应的进程状态
                        * 0 - ready
                        * 1- running
                        * 2 - waiting*/
                        if(chooseFun.allProcess.get(key).getStatus() == 0){
                            jTextArea.append("Ready    ");
                        }else if(chooseFun.allProcess.get(key).getStatus()== 1){
                            jTextArea.append("Running");
                        } else {
```

```
                    jTextArea.append("Waiting");
                }
                jTextArea.append("\t| maxRam: "
                        + chooseFun.allProcess.get(key).getMaxRam()
                        + "\t| time: "
                        + chooseFun.allProcess.get(key).getTime()
                        + "\t| address: "
                        +
chooseFun.allProcess.get(key).addressPCB.start
                        + "--"
                        +
chooseFun.allProcess.get(key).addressPCB.end
                        + "\n");

                // 对正在运行的进程时间减一
                if(chooseFun.allProcess.get(key).getStatus() == 1){
                    chooseFun.allProcess.get(key).setTime(
                            chooseFun.allProcess.get(key).getTime() - 1
                    );

                    // 如果时间小于 0，则表示进程已经结束
                    if (chooseFun.allProcess.get(key).getTime() <= 0){
                        if(chooseFun.processNum != 1){
                            chooseFun.allProcess.get(
                                    chooseFun.getNextKey(key)
                            ).setStatus(1);
                        }

                        chooseFun.allProcess.get(key).setStatus(-1);
```

```
                                chooseFun.processNum -= 1;
                                chooseFun.delAddress(
                                        chooseFun.allProcess.get(key)
                                                .getAddressPCB()
                                );
                            }
                        }
                    }
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        }
}
```

5.1.4. ChooseFun

```
public class ChooseFun implements Runnable {
    public Map<Integer, PCB> allProcess = new HashMap<Integer, PCB>();
    public int processNum = 0;        // 表示现有进程数
    public Address addressHead = new Address(0, 199);

    @Override
    public void run() {
        try {
            while (true) {
                BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
                String string = bufferedReader.readLine();

                switch (string) {
                    case "create":
```

```
                    if (!create()) {
                        System.out.println("进程创建失败");
                    }
                    break;
                case "block":
                    if (!block()) {
                        System.out.println("阻塞进程失败失败");
                    }
                    break;
                case "wake":
                    if (!wake()) {
                        System.out.println("唤醒进程失败失败");
                    }
                    break;
                case "stop":
                    if (!stop()) {
                        System.out.println("终止进程失败失败");
                    }
                    break;
                case "show":
                    show();
                    break;
                default:
                    System.out.println("输入错误，请从新输入!!!  ");
                    break;
            }
        }
    } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    private boolean create(){
        int[] pcbNum = new int[3];
        BufferedReader bufferedReader;
        try{
            System.out.print("id: ");
          bufferedReader=new BufferedReader(new InputStreamReader(System.in));
            pcbNum[0] = Integer.parseInt(bufferedReader.readLine());
            System.out.print("maxRam: ");
          bufferedReader=new BufferedReader(new InputStreamReader(System.in));
            pcbNum[1] = Integer.parseInt(bufferedReader.readLine());
            System.out.print("time: ");
          bufferedReader=new BufferedReader(new InputStreamReader(System.in));
            pcbNum[2] = Integer.parseInt(bufferedReader.readLine());
        } catch (IOException e){
            e.printStackTrace();
            return false;
        }

        Address address = getAddress(pcbNum[1]);

        if(address.start == -1 && address.end == -1) return false;

        if (processNum == 0){
            // 若现在没有进程，则刚创建的进程开始运行
            System.out.println("id:  " + pcbNum[0] + "\tstatus:  Running" +
```

```
"\tmaxRam: " + pcbNum[1] + "\taddress: " + address.start + "--" + address.end + "\ttime:
" + pcbNum[2]);
                allProcess.put(0, new PCB(pcbNum[0], 1, pcbNum[1], pcbNum[2],
address));
            } else {
                // 若存在进程，则创建进程变为 ready 状态
                System.out.println("id: " + pcbNum[0] + "\tstatus: Ready" + "\tmaxRam:
" + pcbNum[1] + "\taddress: " + address.start + "--" + address.end + "\ttime: " +
pcbNum[2]);
                allProcess.put(getMaxKey() + 1, new PCB(pcbNum[0], 0, pcbNum[1],
pcbNum[2], address));
            }
            processNum += 1;
            return true;
        }

        private boolean block() {
            int flag = 0;
            for(Integer key: allProcess.keySet()){
                if(flag == 1){
                    allProcess.get(key).setStatus(1);
                    break;
                }
                if (allProcess.get(key).getStatus() == 1){
                    allProcess.get(key).setStatus(2);
                    flag = 1;
                }
            }
            return true;
```

```
    }

    private boolean wake() {
        System.out.print("id: ");
        try{
            BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
            String string = bufferedReader.readLine();
            for(Integer key: allProcess.keySet()){
                if(allProcess.get(key).getStatus() == 2 &&
allProcess.get(key).getId() == Integer.parseInt(string)){
                    // 找到输入 id 对应的 key 值
                    if(allProcess.size() != 1){
                        // 如果不是最后一个，则将其变为 ready 状态
                        PCB pcb = new PCB(allProcess.get(key).getId(), 0,
allProcess.get(key).getMaxRam(), allProcess.get(key).getTime(),
allProcess.get(key).getAddressPCB());

                        allProcess.put(getMaxKey() + 1, pcb);
                        allProcess.get(key).setStatus(-1);

                        break;
                    } else {
                        // 若是最后一个，则进程开始运行
                        allProcess.get(key).setStatus(1);
                        break;
                    }
```

```
                }
            }
        } catch (IOException e){
            e.printStackTrace();
        }

        return true;
    }

    private boolean stop() {
        System.out.print("id: ");
        try{
            BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
            String string = bufferedReader.readLine();
            for(Integer key: allProcess.keySet()){
                if(allProcess.get(key).getStatus() == 1 &&
allProcess.get(key).getId() == Integer.parseInt(string)){
                    // 找到输入 id 对应的 key 值
                    if(processNum !=
1){ allProcess.get(getNextKey(key)).setStatus(1); } // 若还存在进程，则将其变成
running 状态
                    allProcess.get(key).setStatus(-1);
                    processNum -= 1;
                    delAddress(allProcess.get(key).getAddressPCB());
                    break;
                }
            }
```

```
        } catch (IOException e){ e.printStackTrace(); }
        return true;
    }

    public int getMaxKey(){
        int maxKey = 0;
        for(Integer key: allProcess.keySet()){ maxKey = key; }
        return maxKey;
    }

    public int getNextKey(int tmp){
        int nextKey = 0, flag = 0;
        for(Integer key: allProcess.keySet()){
            if(flag == 1){
                nextKey = key;
                break;
            }
            if (key == tmp){ flag = 1; }
        }
        return nextKey;
    }

    public Address getAddress(int maxLength){
        Address addressPCB = new Address(-1, -1);
        Address addressTmp = addressHead;
        while(true){
            if (addressTmp.length >= maxLength){
                addressPCB = new Address(addressTmp.start, addressTmp.start +
maxLength - 1);
```

```
                addressTmp.start += maxLength;
                addressTmp.length -= maxLength;
                break;
            }
            if(addressTmp.next != null){
                addressTmp = addressTmp.next;
            } else {
                System.out.println("内存不足，创建进程失败!!! ");
                break;
            }
        }
        return addressPCB;
    }

    public void delAddress(Address address){
        if (addressHead.start > address.end){ //  归还内存在最前面
            address.next = addressHead;
            addressHead = address;
            updateAddress();
            return;
        } else {
            Address addressTmp = addressHead;
            while(addressTmp != null){
                if (addressTmp.end < address.start && addressTmp.next != null
&& addressTmp.next.start > address.end){ //  归还内存在中间
                    address.next = addressTmp.next;
                    addressTmp.next = address;
                    updateAddress();
```

```
                    return;
                } else if (addressTmp.next == null && addressTmp.end < address.start){ // 归还内存在最后
                    addressTmp.next = address;
                    updateAddress();
                    return;
                }
                addressTmp = addressTmp.next;
            }
        }
    }

    public void updateAddress(){
        Address addressTmp = addressHead;
        while(addressTmp.next != null){
            if (addressTmp.end == addressTmp.next.start - 1){
                addressTmp.end = addressTmp.next.end;
                addressTmp.length += addressTmp.next.length;
                addressTmp.next = addressTmp.next.next;
            } else {
                addressTmp = addressTmp.next;
            }
        }
    }

    public void show(){
        Address addressTmp = addressHead;
        while(addressTmp != null){
            System.out.println(addressTmp.start + "    " + addressTmp.end + "    " +
```

```
addressTmp.length);
                addressTmp = addressTmp.next;
            }
        }
    }
```

5.2. 运行结果

5.2.1. 创建进程

```
create
id: 0
maxRam: 20
time: 20
id: 0   status: Running maxRam: 20  address: 0--19  time: 20
```

```
id: 0          | status: Running          | maxRam: 20  | time: 11      | address: 0--19
```

5.2.2. 阻塞进程与唤醒进程

```
create
id: 0
maxRam: 20
time: 900
id: 0   status: Running maxRam: 20  address: 0--19  time: 900
```

```
id: 1
maxRam: 20
time: 900
id: 1   status: Ready   maxRam: 20  address: 20--39 time: 900
block
wake
id: 0
```

```
id: 0          | status: Running          | maxRam: 20  | time: 886     | address: 0--19
id: 1          | status: Ready            | maxRam: 20  | time: 900     | address: 20--39

id: 0          | status: Waiting          | maxRam: 20  | time: 875     | address: 0--19
id: 1          | status: Running          | maxRam: 20  | time: 898     | address: 20--39

id: 1          | status: Running          | maxRam: 20  | time: 857     | address: 20--39
id: 0          | status: Ready            | maxRam: 20  | time: 875     | address: 0--19
```

5.2.3. 终止进程

```
create
id: 0
maxRam: 20
time: 900
id: 0   status: Running maxRam: 20  address: 0--19  time: 900
```

```
create
id: 0
maxRam: 20
time: 900
id: 0   status: Running maxRam: 20  address: 0--19  time: 900
```

```
id: 0        | status: Running          | maxRam: 20  | time: 871     | address: 0--19
id: 1        | status: Ready            | maxRam: 20  | time: 900     | address: 20--39
```

```
stop
id: 0
```

```
id: 1        | status: Running          | maxRam: 20  | time: 891     | address: 20--39
```

5.2.4. 内存分配

当我建立两个内存都为 20 的进程时，可以观察他们的内存地址相连。

```
id: 0        | status: Running          | maxRam: 20  | time: 885     | address: 0--19
id: 1        | status: Ready            | maxRam: 20  | time: 900     | address: 20--39
```

当我们终止进程 0，并新建一个内存为 10 的进程 2 时，可以发现进程 2 的内存从 0 开始，表示进程 0 的地址已经归还

```
id: 1        | status: Running          | maxRam: 20  | time: 883     | address: 20--39
id: 2        | status: Ready            | maxRam: 10  | time: 900     | address: 0--9
```

当我们终止进程 1 以后，新建一个进程 3，内存为 40，可以发现，内存 0 剩余的 10 个内存块，进程 1 归还的内存块和剩余的内存块进行了连接。

```
id: 2        | status: Running          | maxRam: 10  | time: 887     | address: 0--9
id: 3        | status: Ready            | maxRam: 40  | time: 900     | address: 10--49
```

6. 遇到的问题及解决方法

在完成实验的时候，我遇到了一些困难，在这里进行举例，并写出我自己的解决方法。

在实现终止进程时，我首选的方法时使用 HashMap 的 remove 的函数，但是会报错，原因是我在界面显示的进程里使用了 for(Integer key: allProcess.keySet()) 方法对所有的进程进行遍历，但是要是其中一个 remove 掉，在这里会出现 key 已经撤销掉的情况，所有会报错，解决的方法主要有两种，一是改用遍历方法，另一种是我应用到的新增一个状态-1 为终止状态。由于我在多出使用了这种遍历方式，所以我采用了第二种解决方法，这种解决方案的好处是我可以修改 JTextArea 的显示数据条件，来看每一个进程的状况，更好的对进程进行控制。

在我选用了这种方法解决问题的时候，新的问题也就来了。在 wake 函数里，由于我直接复制的待改变进程，而导致在我改变其中一个进程的时候，另一个进

程也在跟着改变，最后，我选择了根据原有的进程，新建一个进程，从而进行修改。

另外的一个问题就是，由于我采取了这样的方式来存储进程，造成了进程的id和HashMap中的key没有直接的关系，所以在每次用户输入id时，我需要遍历所有的进程来找到这个id对应的key，从而对目标进程进行操作。接着面临的问题就是在查找下一个进程和将进程放到最后这两部操作时，都没有办法直接进行操作，因为直接key加一的话可能存在其进程status的值为-1的情况，所以新建了两个函数，分别是getMaxKey和getNextKey。

这就是在整个完成项目的过程中，遇到的比较典型的困难及其解决方案，在完成的过程中，既增加了我的编码能力，又增加了我对操作系统进程控制和内存管相关知识的熟悉程度，让我得到了提升。