实验四 --段页式虚拟存储管理

- 姓名：庞晓宇
- 学号：2020118100

实验目的

1. 加深理解段页式虚拟存储管理的概念和原理。
2. 掌握段页式存储管理中存储分配（和回收）方法；
3. 深入了解段页式虚拟存储管理中地址重定位（即地址映射）方法。
4. 深入理解段页式虚拟存储管理中缺段、缺页中断处理方法。

目录结构

```
SPVSM\SRC\MAIN\JAVA
│  config.ini
│
└─com
    └─xftxyz
        └─spvsm
            ├─controller
            │  ├─activity
            │  │      FileUtil.java
            │  │
            │  └─service
            │          OS.java
            │          XFSetting.java
            │
            ├─model
            │  └─domain
            │          Frame.java
            │          Memory.java
            │          PageEntry.java
            │          PCB.java
            │          SegmentEntry.java
            │
            └─view
                ├─ui
                │      Shell.java
                │
                └─utils
                        Input.java
```

实验内容

编写程序完成段页式虚拟存储管理存储分配、地址重定位和缺页中断处理。

1. 为一个进程的内存申请（多少个段，每个段多大）分配内存，当一个进程（完成）结束时回收内存；
2. 对一个给定逻辑地址，判断其是否缺段、缺页，若不缺段、不缺页，则映射出其物理地址；
3. 若缺段则进行缺段中断处理，若缺页则进行缺页中断处理。

假定内存64K，内存块（页框）大小为1K，进程逻辑地址空间最多4个段，每个段最大16K，进程驻留集大小为8页。假设进程运行前未预先装入任何地址空间，页面淘汰策略采用局部（驻留集内）置换策略。

```
memorySize=65536
pageSize=1024
maxSegmentNum=4
maxSegmentSize=16384
maxResidentSetNum=8
```

输出每次存储分配/回收时，内存自由块分布情况、相关进程的段表和页表信息。

```
>>> create process 1 13 15 12 7
IO: 将进程 1 段(0) 页(0) 读入页框 0 中
IO: 将进程 1 段(1) 页(0) 读入页框 1 中
IO: 将进程 1 段(2) 页(0) 读入页框 2 中
IO: 将进程 1 段(3) 页(0) 读入页框 3 中
创建进程 1 成功
>>> show memory
内存使用情况:
0-7:    | 1     | 1     | 1     | 1     |       |       |       |       |
8-15:   |       |       |       |       |       |       |       |       |
16-23:  |       |       |       |       |       |       |       |       |
24-31:  |       |       |       |       |       |       |       |       |
32-39:  |       |       |       |       |       |       |       |       |
40-47:  |       |       |       |       |       |       |       |       |
48-55:  |       |       |       |       |       |       |       |       |
56-63:  |       |       |       |       |       |       |       |       |
>>> create process 3 564 123 789
IO: 将进程 3 段(0) 页(0) 读入页框 4 中
IO: 将进程 3 段(1) 页(0) 读入页框 5 中
IO: 将进程 3 段(2) 页(0) 读入页框 6 中
创建进程 3 成功
>>> create process 4 1164 1223 1789
IO: 将进程 4 段(0) 页(0) 读入页框 7 中
IO: 将进程 4 段(0) 页(1) 读入页框 8 中
IO: 将进程 4 段(1) 页(0) 读入页框 9 中
IO: 将进程 4 段(1) 页(1) 读入页框 10 中
IO: 将进程 4 段(2) 页(0) 读入页框 11 中
IO: 将进程 4 段(2) 页(1) 读入页框 12 中
创建进程 4 成功
>>> show memory
内存使用情况:
0-7:    | 1     | 1     | 1     | 1     | 3     | 3     | 3     | 4     |
8-15:   | 4     | 4     | 4     | 4     | 4     |       |       |       |
16-23:  |       |       |       |       |       |       |       |       |
24-31:  |       |       |       |       |       |       |       |       |
```

```
32-39:  |       |       |       |       |       |       |       |       |
40-47:  |       |       |       |       |       |       |       |       |
48-55:  |       |       |       |       |       |       |       |       |
56-63:  |       |       |       |       |       |       |       |       |
>>> destroy process 1
销毁进程1成功
>>> show memory
内存使用情况:
0-7:    |       |       |       |       | 3     | 3     | 3     | 4     |
8-15:   | 4     | 4     | 4     | 4     | 4     |       |       |       |
16-23:  |       |       |       |       |       |       |       |       |
24-31:  |       |       |       |       |       |       |       |       |
32-39:  |       |       |       |       |       |       |       |       |
40-47:  |       |       |       |       |       |       |       |       |
48-55:  |       |       |       |       |       |       |       |       |
56-63:  |       |       |       |       |       |       |       |       |
>>> create process 5 1164 4223 3789
IO: 将进程 5 段(0) 页(0) 读入页框 0 中
IO: 将进程 5 段(0) 页(1) 读入页框 1 中
IO: 将进程 5 段(1) 页(0) 读入页框 2 中
IO: 将进程 5 段(1) 页(1) 读入页框 3 中
IO: 将进程 5 段(1) 页(2) 读入页框 13 中
IO: 将进程 5 段(1) 页(3) 读入页框 14 中
IO: 将进程 5 段(1) 页(4) 读入页框 15 中
IO: 将进程 5 段(2) 页(0) 读入页框 16 中
创建进程 5 成功
>>> show memory
内存使用情况:
0-7:    | 5     | 5     | 5     | 5     | 3     | 3     | 3     | 4     |
8-15:   | 4     | 4     | 4     | 4     | 4     | 5     | 5     | 5     |
16-23:  | 5     |       |       |       |       |       |       |       |
24-31:  |       |       |       |       |       |       |       |       |
32-39:  |       |       |       |       |       |       |       |       |
40-47:  |       |       |       |       |       |       |       |       |
48-55:  |       |       |       |       |       |       |       |       |
56-63:  |       |       |       |       |       |       |       |       |
```

拓展

- 采用LRU页面置换算法实现页面淘汰。

提示

1. 内存状态描述
    1. 分块（页框）说明表内容：编号、状态
    2. 组织方式：线性表，位图?
    3. 设置初始内存分配状态：随机设定若干块为已分配。
2. 段表、页表设计及其关系
3. 逻辑地址的表示
4. 缺段、缺页中断处理中的页面淘汰
    1. 使用最简单的FIFO策略，选择要淘汰的页

测试输出

- 输出当前内存分配情况（有多少可用块、哪些块可用？）；

```
>>> show memory
内存使用情况:
0-7:    | 5     | 5     | 5     | 5     | 3     | 3     | 3     | 4     |
8-15:   | 4     | 4     | 4     | 4     | 4     | 5     | 5     | 5     |
16-23:  | 5     | 6     | 6     | 6     | 6     | 6     | 6     | 6     |
24-31:  | 6     |       |       |       |       |       |       |       |
32-39:  |       |       |       |       |       |       |       |       |
40-47:  |       |       |       |       |       |       |       |       |
48-55:  |       |       |       |       |       |       |       |       |
56-63:  |       |       |       |       |       |       |       |       |
```

- 手工输入进程的内存总需求（多少段，每个段多大）；
- 手工输入某进程的内存申请（哪几个段，各自需要多少块？），输出系统为其分配内存后的段表和页表内容。

```
>>> create process 6 6048 7192 5681
IO: 将进程 6 段(0) 页(0) 读入页框 17 中
IO: 将进程 6 段(0) 页(1) 读入页框 18 中
IO: 将进程 6 段(0) 页(2) 读入页框 19 中
IO: 将进程 6 段(0) 页(3) 读入页框 20 中
IO: 将进程 6 段(0) 页(4) 读入页框 21 中
IO: 将进程 6 段(0) 页(5) 读入页框 22 中
IO: 将进程 6 段(1) 页(0) 读入页框 23 中
IO: 将进程 6 段(1) 页(1) 读入页框 24 中
创建进程 6 成功
```

- 模拟内存访问指令的地址映射：比如手工输入一个逻辑地址，系统提示是否缺段、缺页，若不缺段、不缺页，则输出其物理地址；

```
>>> address 5 5 5
操作失败，进程5 段(5)不存在
>>> address 5 2 3
进程5段(2) 段偏移(3) 物理地址为: 16387
```

- 若缺段或缺页，则输出装入（被淘汰）的块号，输出缺段或缺页中断处理后的段表和页表信息。

```
>>> h
create process 进程id 各个段大小          创建一个进程
destroy process 进程id           销毁一个进程
show memory                       显示内存使用情况
show process 进程id               显示该进程驻留集、置换策略、段表、页表
```

```
address 进程名 段号 段偏移                将逻辑地址映射为物理地址
help or h                          获取帮助
quit or q                          退出
```

代码

FileUtil.java

```
package com.xftxyz.spvsm.controller.activity;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

import com.xftxyz.spvsm.controller.service.XFSetting;

/**
 * 文件工具类
 */
public class FileUtil {

    /**
     * 读取配置文件
     *
     * @return 配置文件的内容
     * @throws IOException 加载配置文件时可能出现的异常
     */
    public static Map<String, Integer> readIniFile() {
        InputStream in =
FileUtil.class.getClassLoader().getResourceAsStream(XFSetting.configFilePath);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));
        Properties props = new Properties();
        try {
            props.load(br);
        } catch (IOException e) {
            try (PrintWriter pw = new PrintWriter("config.ini")) {
                pw.println("memorySize=" + XFSetting.DEFAULT_MEMORY_SIZE);
                pw.println("pageSize=" + XFSetting.DEFAULT_PAGE_SIZE);
                pw.println("maxSegmentNum=" + XFSetting.DEFAULT_MAX_SEGMENT_NUM);
                pw.println("maxSegmentSize=" +
XFSetting.DEFAULT_MAX_SEGMENT_SIZE);
                pw.println("maxResidentSetNumber=" +
XFSetting.DEFAULT_MAX_RESIDENT_SET_NUMBER);

            } catch (IOException e1) {
                throw new RuntimeException("初始化配置文件失败");
```

```
            }
            try {
                // 再次加载
                props.load(br);
            } catch (IOException e1) {
                throw new RuntimeException("初始化配置文件失败");
            }
        }
        Map<String, Integer> map = new HashMap<String, Integer>();
        for (Object s : props.keySet()) {
            String key = s.toString();
            Integer value = Integer.parseInt(props.getProperty(key));
            map.put(key, value);
            // map.put(s.toString(), props.getProperty(s.toString()));
        }
        return map;
    }
}
```

OS.java

```
package com.xftxyz.spvsm.controller.service;

import java.util.HashMap;
import java.util.Map;

import com.xftxyz.spvsm.model.domain.Memory;
import com.xftxyz.spvsm.model.domain.PCB;
import com.xftxyz.spvsm.model.domain.PageEntry;
import com.xftxyz.spvsm.model.domain.SegmentEntry;

public class OS {
    public static int memorySize = 64 * 1024; // 内存大小
    public static int pageSize = 1 * 1024; // 页大小，为了简化地址转换，为2的幂
    public static int maxSegmentNum = 4; // 一个程序最多有多少个段
    public static int maxSegmentSize = 16 * 1024; // 一个段最大大小
    public static int maxResidentSetNum = 8; // 进程驻留集最多多少个页

    public static enum REPLACE_POLICY {
        FIFO, LRU
    }; // 置换策略 FIFO or LRU

    public static REPLACE_POLICY ReplacePolicy = REPLACE_POLICY.LRU; // 置换策略，
默认为LRU

    private Map<String, PCB> processes = new HashMap<>();
    private Memory memory = null;
    // private Memory memory = new Memory(memorySize / pageSize);

    // public OS() {
    // PCB.setMemory(memory);
    // }
```

```
    public void init(Map<String, Integer> config) {
        if (config.containsKey("memorySize")) {
            memorySize = config.get("memorySize");
        }
        if (config.containsKey("pageSize")) {
            pageSize = config.get("pageSize");
        }
        if (config.containsKey("maxSegmentNum")) {
            maxSegmentNum = config.get("maxSegmentNum");
        }
        if (config.containsKey("maxSegmentSize")) {
            maxSegmentSize = config.get("maxSegmentSize");
        }
        if (config.containsKey("maxResidentSetNum")) {
            maxResidentSetNum = config.get("maxResidentSetNum");
        }
        // if (config.containsKey("ReplacePolicy")) {
        // ReplacePolicy = REPLACE_POLICY.valueOf(config.get("ReplacePolicy"));
        // }
        memory = new Memory(memorySize / pageSize);
        PCB.setMemory(memory);
    }

    /**
     * 创建一个进程，返回创建是否成功
     */
    public boolean createProcess(String id, int[] segments) {
        String mess = validate(id, segments);
        if (mess != null) {
            System.out.println("创建进程失败(" + mess + ")");
            return false;
        }
        // 验证是否有足够的内存
        PCB process = new PCB(id, segments, ReplacePolicy);
        if (process.residentSetCount > memory.unusedFrameCount()) {
            System.out.println("创建进程失败(内存不足)");
            return false;
        }
        // 申请内存，设置驻留集
        processes.put(id, process);
        int[] frame = memory.mallocFrame(id, process.residentSetCount);
        process.residentSet = frame;
        // 随机载入一些页
        process.initLoad();

        System.out.println("创建进程 " + id + " 成功");
        return true;
    }

    /**
     * 验证创建的进程合法性，若合法，返回null；否则返回错误信息
     */
    private String validate(String id, int[] segments) {
```

```
        if (processes.containsKey(id)) {
            return "进程名重复";
        }
        if (segments.length == 0 || segments.length > 4) {
            return "一个进程只能有1 到 " + OS.maxSegmentNum + " 个段";
        }
        for (int i = 0; i < segments.length; i++) {
            if (segments[i] <= 0 || segments[i] > OS.maxSegmentSize) {
                return "一个段必须小于 " + OS.maxSegmentSize + "KB";
            }
        }

        return null; // 合法，返回null
    }

    /**
     * 销毁一个进程
     */
    public void destroyProcess(String id) {
        PCB process = processes.get(id);
        if (process == null) {
            System.out.println("操作失败，进程" + id + "不存在");
            return;
        }

        int[] frames = process.residentSet;
        memory.freeFrame(frames);
        processes.remove(id);
        System.out.println("销毁进程" + id + "成功");
    }

    /**
     * 将逻辑地址（段号+段偏移）转化成物理地址。若出错，返回-1
     * 如果发生缺页中断，根据置换策略选择一个页换出，并将请求的页载入到内存中
     */
    public int toPhysicalAddress(String id, int segmentNum, int segmentOffset) {
        PCB process = processes.get(id);
        if (process == null) {
            System.out.println("操作失败，进程" + id + "不存在");
            return -1;
        }
        // 判断请求的段是否存在
        if (segmentNum < 0 || segmentNum >= process.STable.length) {
            System.out.println("操作失败，进程" + id + " 段(" + segmentNum + ")不存
在");
            return -1;
        }

        SegmentEntry segment = process.STable[segmentNum];
        // 若段偏移大于段大小，则请求失败
        if (segmentOffset > segment.segmentSize) {
            System.out.println("操作失败，进程 " + id + " 段偏移(" + segmentOffset
+ ") 越界");
            return -1;
```

```
            }

            // 根据segmentOffset计算页号和页偏移
            int pageNum = segmentOffset / OS.pageSize;
            int pageOffset = segmentOffset % OS.pageSize;

            PageEntry page = segment.pageTable[pageNum];
            if (page.load == false) {
                // 如果该页不在内存中，根据淘汰策略淘汰一个页面，并将该页载入
                System.out.println("请求的页不再内存中，发生缺页中断");
                process.replacePage(segmentNum, pageNum);
            }

            // 计算物理地址、设置该页使用时间
            page.usedTime = System.currentTimeMillis();
            int frameNum = page.frameNum;
            int beginAddress = memory.getFrame(frameNum).beginAddress;
            System.out.println(
                    "进程" + id + "段(" + segmentNum + ") 段偏移(" + segmentOffset + ")
物理地址为: " + (beginAddress + pageOffset));
            return beginAddress + pageOffset;
        }

        /**
         * 设置默认置换策略
         */
        public static void setReplacePolicy(OS.REPLACE_POLICY policy) {
            OS.ReplacePolicy = policy;
        }

        /**
         * 返回内存使用情况
         */
        public void showMemory() {
            System.out.println(memory.toString());
            System.out.println();
        }

        /**
         * 返回进程的段表和页表
         */
        public void showProcess(String id) {
            PCB process = processes.get(id);
            if (process == null) {
                System.out.println("要查看的进程不存在");
                return;
            }

            StringBuilder sb = new StringBuilder();

            int[] frames = process.residentSet;
            sb.append("驻留集: [ ");
            for (int elem : frames) {
                sb.append(elem + " ");
```

```
        }
        sb.append("]\n");
        sb.append("置换策略: ");
        if (process.policy == REPLACE_POLICY.FIFO) {
            sb.append("FIFO ");
            sb.append("[ ");
            for (Integer[] something : process.loadQueue) {
                sb.append("(" + something[0] + ", " + something[1] + ") ");
            }
            sb.append("]\n\n");
        } else {
            sb.append("LRU\n\n");
        }

        for (SegmentEntry segment : process.STable) {
            sb.append("进程" + id + " 段号:" + segment.segmentNum + " 段大小:" +
segment.segmentSize + "\n");
            sb.append("------------------------------------------------------------
------\n");
            sb.append("| 页号\t| 是否载入\t| 页框号\t| 页框起始地址\t| 上一次访问时间
\t|\n");
            sb.append("------------------------------------------------------------
------\n");
            for (PageEntry page : segment.pageTable) {
                sb.append("| " + page.pageNum + "\t");
                if (page.load) {
                    sb.append("| load\t\t| " + page.frameNum + "\t| " +
memory.getFrame(page.frameNum).beginAddress
                            + "\t\t| " + page.usedTime + " |\n");
                } else {
                    sb.append("| unload\t| \t| \t\t| \t\t|\n");
                }
            }
            sb.append("------------------------------------------------------------
------\n\n");
        }

        System.out.print(sb.toString());
    }

}
```

XFSetting.java

```
package com.xftxyz.spvsm.controller.service;

/**
 * 配置
 */
public interface XFSetting {
    String version = "1.0.0";
    String name = "段页式虚拟存储管理";
```

```
    String configFilePath = "config.ini";

    String helpMess = "create process 进程id 各个段大小\t创建一个进程\n" +
            "destroy process 进程id\t\t销毁一个进程\n" +
            "show memory\t\t\t显示内存使用情况\n" +
            "show process 进程id\t\t显示该进程驻留集、置换策略、段表、页表\n" +
            "address 进程名 段号 段偏移\t\t将逻辑地址映射为物理地址\n" +
            "help or h\t\t\t获取帮助\n" +
            "quit or q\t\t\t退出\n";
    String description = "内存大小64K，页框大小为1K，一个进程最多有4个段，且每个段最大为16K。一个进程驻留集最多为8页。\n"
            + "驻留集置换策略：局部策略（仅在进程的驻留集中选择一页）\n"
            + "页面淘汰策略：FIFO、LRU\n"
            + "进程初始载入策略：从第0个、第1个段...依次载入页，直到驻留集已全部载入\n"
            + "放置策略：决定一个进程驻留集存放在内存什么地方。优先放在低页框\n";

    // 配置默认值
    int DEFAULT_MEMORY_SIZE = 65536; // 内存大小（默认值）
    int DEFAULT_PAGE_SIZE = 1024; // 页大小，为了简化地址转换，为2的幂（默认值）
    int DEFAULT_MAX_SEGMENT_NUM = 4; // 一个程序最多有多少个段（默认值）
    int DEFAULT_MAX_SEGMENT_SIZE = 16384; // 一个段最大大小（默认值）
    int DEFAULT_MAX_RESIDENT_SET_NUMBER = 8; // 进程驻留集最多多少个页（默认值）
}
```

Frame.java

```
package com.xftxyz.spvsm.model.domain;

/**
 * 页框类
 */
public class Frame {

    // 以下的属性都是public的，方便在Memory类中被访问
    public int frameNum; // 页框号
    public int beginAddress; // 该页框起始地址
    public boolean used; // 标志该页框是否使用

    // 该页框现在被那个进程使用，不是页框必须的信息，只是为了展示内存时更直观
    public String id;

    /**
     * 构造函数
     *
     * @param frameNum     页框号
     * @param beginAddress 该页框起始地址
     */
    public Frame(int frameNum, int beginAddress) {
        super();
        this.frameNum = frameNum;
        this.beginAddress = beginAddress;
```

```
        setUnused();
    }

    /**
     * 设置该页框被使用
     *
     * @param id 使用该页框的进程id
     */
    public void setUsed(String id) {
        this.used = true;
        this.id = id;
    }

    /**
     * 设置该页框未使用
     */
    public void setUnused() {
        this.used = false;
        this.id = null;
    }
}
```

Memory.java

```
package com.xftxyz.spvsm.model.domain;

import com.xftxyz.spvsm.controller.service.OS;

public class Memory {

    private Frame[] memory; // 内存
    private int unusedFrameCount = 0; // 未使用的页框数

    /**
     * 创建有frameNum个未使用页框的内存
     *
     * @param frameNum 内存中页框的数量
     */
    public Memory(int frameNum) {
        memory = new Frame[frameNum]; // 创建内存，含有frameNum个页框
        for (int i = 0; i < frameNum; i++) {
            memory[i] = new Frame(i, i * OS.pageSize); // 创建页框，页框号为i，起始
地址为i*pageSize
        }
        unusedFrameCount = frameNum;// 开始时，全部页框都未使用
    }

    /**
     * 申请页框
     *
     * 放置策略：优先放在低页框号的页框中
     * 申请(设置used为true)前n个未使用的页框，返回包含页框号的数组；若剩余内存不够，返回
```

```
null
     *
     * @param id 进程id
     * @param n  需要申请的页框数
     * @return 包含页框号的数组，若剩余内存不够，返回null
     */
    public int[] mallocFrame(String id, int n) {
        // 剩余页框数量不足
        if (unusedFrameCount < n) {
            return null;
        }
        // 结果数组，保存申请到的页框号
        int[] result = new int[n];
        // 遍历内存，找到n个未使用的页框
        for (int i = 0, index = 0; index < n && i < memory.length; i++) {
            if (memory[i].used == false) {
                result[index] = memory[i].frameNum;
                memory[i].setUsed(id);
                index++;
            }
        }
        // 申请成功，更新未使用页框数
        unusedFrameCount -= n;
        // 返回页框号数组
        return result;
    }

    /**
     * 释放页框
     *
     * 将frames中的页框号对应的页框设置为未使用
     *
     * @param frames 页框号数组
     */
    public void freeFrame(int[] frames) {
        for (int i = 0; i < frames.length; i++) {
            memory[frames[i]].setUnused();
        }
        // 更新未使用页框数
        unusedFrameCount += frames.length;
    }

    /**
     * 获取未使用页框数
     *
     * @return 未使用页框数
     */
    public int unusedFrameCount() {
        return unusedFrameCount;
    }

    /**
     * 模拟从外存读入一页
     *
```

```
     * 从外存读入id进程segmentNum段pageNum页frameNum的页框中
     *
     * @param id         进程id
     * @param segmentNum 段号
     * @param pageNum    页号
     * @param frameNum   页框号
     */
    public void readPage(String id, int segmentNum, int pageNum, int frameNum) {
        System.out.println("IO: 将进程 " + id + " 段(" + segmentNum + ") 页(" +
pageNum + ") 读入页框 " + frameNum + " 中");
    }

    /**
     * 模拟将程序的一页写入外存。
     *
     * 将frameNum的内容写入外存，写入内容为id进程segmentNum段pageNum页
     *
     * @param id         进程id
     * @param segmentNum 段号
     * @param pageNum    页号
     * @param frameNum   页框号
     */
    public void writePage(String id, int segmentNum, int pageNum, int frameNum) {
        System.out.println("IO: 将页框" + frameNum + "内容写入外存。进程 " + id + "
段(" + segmentNum + ") 页(" + pageNum + ")");
    }

    /**
     * 获取页框
     *
     * 返回页框号为frame的页框，若请求页框不存在，返回null
     *
     * @param frameNum 页框号
     * @return 页框，若请求页框不存在，返回null
     */
    public Frame getFrame(int frameNum) {
        if (frameNum >= 0 && frameNum < memory.length) {
            return memory[frameNum];
        }
        return null;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("内存使用情况: ");
        for (int i = 0; i < memory.length; i++) {
            if (i % 8 == 0) { // 一行显示8个页框
                sb.append("\n" + i + "-" + (i + 7) + ":\t| ");
            }
            if (memory[i].used) {
                // 截取进程id前5个字符输出
                String id = memory[i].id;
                if (id.length() > 5) {
```

```
                    id = id.substring(0, 4);
                }
                sb.append(id + "\t| ");
            } else {
                sb.append("     \t| ");
            }
        }

        return sb.toString();
    }
}
```

PageEntry.java

```
package com.xftxyz.spvsm.model.domain;

/**
 * 页表项
 */
public class PageEntry {
    public int pageNum; // 页号
    public boolean load; // 该页是否载入

    // load为false时，以下字段无意义
    public int frameNum; // 该页载入的页框号。
    // 该页最近一次被访问的时间：用于实现页面置换策略LRU，当该页被载入内存或被访问时，重置该时间
    public long usedTime;
    public String info = ""; // 其他信息，如设置保护、共享等

    /**
     * 构造函数
     *
     * 创建一个指定页号、未载入的页
     *
     * @param pageNum 页号
     */
    public PageEntry(int pageNum) {
        this.pageNum = pageNum;
        setUnload();
    }

    /**
     * 载入内存
     *
     * 设置该页载入到页框号为frameNum的页框中
     *
     * @param frameNum 页框号
     */
    public void setLoad(int frameNum) {
        this.load = true;
        this.frameNum = frameNum;
```

```
            usedTime = System.currentTimeMillis();
        }

        /**
         * 将该页载出内存
         */
        public void setUnload() {
            this.load = false;
            this.frameNum = -1;
            usedTime = -1;
        }
    }
```

PCB.java

```
package com.xftxyz.spvsm.model.domain;

import java.util.LinkedList;
import java.util.Queue;

import com.xftxyz.spvsm.controller.service.OS;

public class PCB {

    private static Memory memory = null;

    public String id;
    public SegmentEntry[] STable;
    public int residentSetCount; // 驻留集页个数
    public int[] residentSet; // 驻留集页框号
    public OS.REPLACE_POLICY policy; // 替换策略

    // 页载入内存的顺序。其中Integer数组元素分别为段号、页号
    // 用于实现替换策略的 FIFO
    // 重要：每当页载入内存时更新队列
    public Queue<Integer[]> loadQueue = new LinkedList<>();

    public PCB(String id, int[] segments, OS.REPLACE_POLICY policy) {
        this.id = id;
        this.policy = policy;
        STable = new SegmentEntry[segments.length];
        for (int i = 0; i < STable.length; i++) {
            STable[i] = new SegmentEntry(i, segments[i]);
        }

        // 计算驻留集大小
        residentSetCount = 0;
        for (int i = 0; i < STable.length; i++) {
            residentSetCount += STable[i].pageTable.length;
        }
        if (residentSetCount > OS.maxResidentSetNum) {
            residentSetCount = OS.maxResidentSetNum;
```

```
        }
    }

    /**
     * 只需在调用PCB类方法之前调用一次，用于设置Memory对象
     */
    public static void setMemory(Memory m) {
        memory = m;
    }

    /**
     * 创建进程完成后，载入一些页。若该程序可以全部放入驻留集中，则将全部程序载入
     * 初始载入策略：从第0个、第1个段...依次载入页，直到驻留集已全部载入
     */
    public void initLoad() {
        int index = 0;
        for (SegmentEntry segment : STable) {
            for (PageEntry page : segment.pageTable) {
                if (index >= residentSetCount) {
                    break;
                }
                page.setLoad(residentSet[index]);
                loadQueue.add(new Integer[] { segment.segmentNum, page.pageNum });
                memory.readPage(id, segment.segmentNum, page.pageNum,
residentSet[index]);
                index++;
            }
        }
    }

    /* 置换策略：FIFO or LRU */
    /**
     * 根据FIFO策略选择一个页，依次返回该页的段号、页号
     */
    private Integer[] selectReplacePage_FIFO() {
        return loadQueue.poll();
    }

    /**
     * 根据LRU策略选择一个页，依次返回该页的段号、页号
     */
    private Integer[] selectReplacePage_LRU() {
        long leastTime = System.currentTimeMillis() + 1000000; // 设置为现在以后的
时间
        int segmentNum = -1;
        int pageNum = -1;

        // 遍历所有页，找到usedTime最小的
        for (SegmentEntry segment : STable) {
            for (PageEntry page : segment.pageTable) {
                if (page.load && page.usedTime < leastTime) {
                    leastTime = page.usedTime;
                    segmentNum = segment.segmentNum;
                    pageNum = page.pageNum;
```

```
                }
            }
        }

        return new Integer[] { segmentNum, pageNum };
    }

    /**
     * 依据policy策略选择一页换出驻留集，并将segmentNum段pagNum页换入
     * SN SegmentNum
     * PN PageNum
     */
    public void replacePage(int inSN, int inPN) {
        Integer[] something;
        if (policy == OS.REPLACE_POLICY.FIFO) {
            something = selectReplacePage_FIFO();
        } else {
            something = selectReplacePage_LRU();
        }

        int outSN = something[0];
        int outPN = something[1];

        PageEntry inPage = STable[inSN].pageTable[inPN];
        PageEntry outPage = STable[outSN].pageTable[outPN];
        int frameNum = outPage.frameNum;
        memory.writePage(id, outSN, outPN, frameNum);
        outPage.setUnload();
        memory.readPage(id, inSN, inPN, frameNum);
        inPage.setLoad(frameNum);
        loadQueue.add(new Integer[] { inSN, inPN });
    }
}```

SegmentEntry.java
```java
package com.xftxyz.spvsm.model.domain;

import com.xftxyz.spvsm.controller.service.OS;

/**
 * 段表项
 */
public class SegmentEntry {
    public int segmentNum = -1; // 段号
    public int segmentSize = -1; // 段长
    public PageEntry[] pageTable = null; // 对应的页表

    /**
     * 段表项构造函数
     *
     * @param segmentNum  段号
     * @param segmentSize 段长
     */
```
```

```
    public SegmentEntry(int segmentNum, int segmentSize) {
        this.segmentNum = segmentNum;
        this.segmentSize = segmentSize;

        // 计算需要的页表项数量
        int count = segmentSize / OS.pageSize; // 页表的大小
        // 如果页表大小不能整除段长，则需要多一个页表项
        if (segmentSize % OS.pageSize != 0) {
            count++;
        }
        // 创建页表
        pageTable = new PageEntry[count];
        for (int i = 0; i < count; i++) {
            pageTable[i] = new PageEntry(i);
        }
    }
}
```

Shell.java

```java
package com.xftxyz.spvsm.view.ui;

import java.util.Map;

import com.xftxyz.spvsm.controller.activity.FileUtil;
import com.xftxyz.spvsm.controller.service.OS;
import com.xftxyz.spvsm.controller.service.XFSetting;
import com.xftxyz.spvsm.view.utils.Input;

/**
 * 主程序
 */
public class Shell {
    private OS os = new OS();

    public static void main(String[] args) {
        new Shell().run();
    }

    private void run() {
        printMessage();
        initialize();
        setReplacePolicy();
        System.out.println("输入h[elp]获取更多帮助信息");
        shell();
        Input.close();
    }

    /**
     * 打印一些说明信息
     */
    private void printMessage() {
        System.out.println("============================" + XFSetting.name + " v
" + XFSetting.version
```

```
                + "==============================\n");

        System.out.println(XFSetting.description + "\n");
    }

    private void initialize() {
        Map<String, Integer> config = FileUtil.readIniFile();
        os.init(config);
    }

    /**
     * 设置默认置换策略
     */
    public void setReplacePolicy() {
        System.out.print(">>> 请设置置换策略(0[FIFO] 1[LRU]): ");
        while (true) {
            String mess = Input.nextLine().trim();
            if ("0".equals(mess) || "FIFO".equals(mess)) {
                OS.setReplacePolicy(OS.REPLACE_POLICY.FIFO);
                System.out.println("设置置换策略为FIFO");
                break;
            } else if ("1".equals(mess) || "LRU".equals(mess)) {
                OS.setReplacePolicy(OS.REPLACE_POLICY.LRU);
                System.out.println("设置置换策略为LRU");
                break;
            } else {
                System.out.print(">>> 输入有误，请设置置换策略(0[FIFO] 1[LRU]): ");
            }
        }
    }

    /**
     * 1. create process pname segments
     * 2. destroy process pname
     * 3. show memory
     * 4. show process pname
     * 5. help or h
     * 6. quit or q
     * 8. address pname sgementNum segmentOffset
     */
    public void shell() {
        System.out.print(">>> ");
        while (true) {
            String command = Input.nextLine();
            if (command == null || command.trim().equals("")) {
                System.out.print(">>> ");
                continue;
            }

            String[] words = command.split(" ");
            if (words.length >= 4 && "create".equals(words[0].trim()) &&
"process".equals(words[1].trim())) {
                String processId = words[2].trim();
                int[] segments = new int[words.length - 3];
```

```
                try {
                    for (int i = 3, index = 0; i < words.length; i++, index++) {
                        segments[index] = Integer.parseInt(words[i]);
                        if (segments[index] <= 0) {
                            throw new Exception();
                        }
                    }
                } catch (Exception ex) {
                    System.out.println("命令有误 段大小必须为正整数(help获取帮助)");
                    System.out.print(">>> ");
                    continue;
                }
                os.createProcess(processId, segments);

            } else if (words.length == 3 && "destroy".equals(words[0].trim()) &&
"process".equals(words[1].trim())) {
                String processId = words[2].trim();
                os.destroyProcess(processId);

            } else if (words.length == 2 && "show".equals(words[0].trim()) &&
"memory".equals(words[1].trim())) {
                os.showMemory();

            } else if (words.length == 3 && "show".equals(words[0].trim()) &&
"process".equals(words[1].trim())) {
                String processId = words[2].trim();
                os.showProcess(processId);

            } else if (words.length == 1 && "help".equals(words[0].trim()) ||
"h".equals(words[0].trim())) {
                System.out.println(XFSetting.helpMess);

            } else if (words.length == 1 && "quit".equals(words[0].trim()) ||
"q".equals(words[0].trim())) {
                System.out.println("quit");
                break;
            } else if (words.length == 4 && "address".equals(words[0].trim())) {
                String porcessId = words[1].trim();
                int segmentNum, segmentOffset;
                try {
                    segmentNum = Integer.parseInt(words[2].trim());
                    segmentOffset = Integer.parseInt(words[3].trim());
                    if (segmentNum < 0 || segmentOffset < 0) {
                        throw new Exception();
                    }
                } catch (Exception ex) {
                    System.out.println("命令有误 段号和段偏移必须为正整数(help获取帮
助)");
                    System.out.print(">>> ");
                    continue;
                }
                os.toPhysicalAddress(porcessId, segmentNum, segmentOffset);

            } else {
```

```
                System.out.println("命令有误(help获取帮助)");
            }

            System.out.print(">>> ");
        }
    }

}
```

Input.java

```
package com.xftxyz.spvsm.view.utils;

import java.util.Scanner;

/**
 * 当有多个Scanner对象时，调用close()方法，会顺便关闭System.in对象，
 * 如果调用了其他Scanner对象的方法，会出现java.util.NoSuchElementException异常
 *
 * 此类是对Scanner的简单封装，用于统一获取输入和关闭Scanner对象
 */
public class Input {
    private static Scanner input = new Scanner(System.in);

    public static void close() {
        input.close();
    }

    public static String nextLine() {
        return input.nextLine();
    }
}
```

AppTest.java

```
package com.xftxyz.spvsm;

import static org.junit.Assert.assertTrue;

import org.junit.Test;

/**
 * Unit test for simple App.
 */
public class AppTest {
    /**
     * Rigorous Test :-)
     */
    @Test
```

```
    public void shouldAnswerWithTrue() {
        assertTrue(true);
    }

    @Test
    public void configReadTest() {
        // System.out.println("asdfasdfa");
        try {
            // Map<String, String> map = FileUtil.readIniFile();
            // // 遍历map
            // for (Map.Entry<String, String> entry : map.entrySet()) {
            // System.out.println(entry.getKey() + ":" + entry.getValue());
            // }
        } catch (Exception e) {
            e.printStackTrace();

        }

    }

}
```