توضیحات پروژه سیستم عامل

دکتر خانمیرزا

سپهر شاهسوار 9727543

علی الهی 9724743

تقسیم کارها بدین صورت بوده است :

سپهر شاهسوار : بخش های entities, utils و همینطور متد allocate

علی الهی : بخش های start و متد deallocate

اکثر قسمت ها بصورت مشترک پیاده سازی شد.

فولدر Entities

Process.java

```java
    private int PID;
    private boolean allocate;
    private int size;
    private int address;
    private int duration;
    private int requests;
    private Address usedAddress;

    public Process(int PID) {
        RandomValue random = new RandomValue();
        this.PID = PID;
        allocate = random.bool();
        duration = random.time();
        requests = random.request();
        address = random.address();
        size = random.size();
    }

```

لیست attribute ها و constructor کلاس Process

متد request :

```java
public void request(MemoryManager memoryManager) throws InterruptedException {
    for (int i = 0; i < requests; i++) {
        if (allocate) {
            TimeUnit.SECONDS.sleep(duration);
            usedAddress = new Address();
            usedAddress = memoryManager.allocate(size);
        } else {
            TimeUnit.SECONDS.sleep(duration);
            memoryManager.deallocate(size);
        }
    }
}
```

کارکرد این متد به طور ساده به این شکل است که وقتی فراخوانی میشود چک میکند که فرآیند مورد نظر Allocate شده است یا خیر.

اگر مقدار allocate برابر با false باشد آنوقت یک آدرس جدید ساخته میشود و تابع memoryManager.allocate() فراخوانی میشود.

(تابع بالا به طور کامل در ادامه توضیح داده خواهد شد...)

اما اگر مقدار allocate برابر با true باشد ، در آنصورت نیاز است تا تابع memoryManager.deallocate() فراخوانی شود.

(این تابع نیز به طور کامل در ادامه توضیح داده خواهد شد...)

متد getAddress :

```java
public void getAddress() {
    if (usedAddress == null)
        System.out.println("\uD83D\uDCA4 Process with Number : " + PID + " don't use any space");
    else {
        System.out.println("\uD83D\uDCBB Process with Number : " + PID + " used address: " + usedAddress.getBase()
                + "  with size: " + usedAddress.getSize());
    }
}
```

این متد نشان میدهد فرآیند مورد نظر آدرسی را اشغال کرده است یا خیر.

متد getInfo :

```java
    public void getInfo() {
        System.out.println("\uD83D\uDCCB Process Number " + PID + ":");

        if (allocate)
            System.out.println("\uD83D\uDCBE Allocate memory size: " + size + " Kilobytes");
        else
            System.out.println("\uD83D\uDCBE Deallocate memory address: " + address);
        System.out.println("⚙ Number of Requests: " + requests);
        System.out.println("⏱ Time: " + duration);

    }
}
```

این متد تمام وضعیت های فرآیند را نشان میدهد. شامل :

allocated memory, requests, duration …

Block.js

این کلاس دارای دو attribute ، start و finish است.

دو متد getStart() و getFinish() هم برای استفاده بیرون از کلاس Block

---

Address.java

این کلاس نیز دارای دو attribute ، base و size است.

دو متد getBase() و getSize() هم برای استفاده بیرون از کلاس Address

فولدر OS

MemoryManager.java

این کلاس در اصل وظیفه ی مدیریت حافظه را دارد .

```java
public class MemoryManager {

    private List<List<Block>> blocks;
    private HashMap<Integer, Integer> addressList;


    public MemoryManager(int memorySize) {

        addressList = new HashMap<>();
        blocks = new ArrayList<>();
        int numberOfBlocks = (int) Math.ceil(Math.log(memorySize) / Math.log(2));

        for (int i = 0; i <= numberOfBlocks; i++) {
            blocks.add(i, new ArrayList<>());
        }
        blocks.get(numberOfBlocks).add(new Block( start: 0,  finish: memorySize - 1));
    }

    public Address allocate(int size) {...}

    public void deallocate(int address) {...}
}
```

addressList : ابتدا و اندازه تمام آدرس هایی که توسط فرآیند ها اشغال شده اند در این مپ نگهداری میشود که امکان آزاد کردن فضای آدرس اشغال شده را برای ما راحتتر میکند .

blocks: یک لیست است که خود شامل زیرلیست های دیگریست که همان لیست بلاک ها با فضای متفاوت است . یعنی هر زیر لیست خود نماینده ی بلاک هایی با یک سایز مشخص است .

Constructor : در کانستراکتور این برنامه کار اساسی انجام شده درست کردن لیست هایست که بلاک های با سایز یکسان را نگه میدارند . به اینصورت که بسته به اینکه اندازه مموری ما چقدر باشد این بلاک میتوانند از سایز

n مگابایتی باشند تا 32 کیلوبایتی . البته واضح است که بزرگترین بلاکی که به هر فرآیند اختصاص داده می شود 1024کیلوبایت است .

تابع allocate :

```
28    public Address allocate(int size) {
29
30        int exponent = (int) Math.ceil(Math.log(size) / Math.log(2));
31
32        int index;
33
34        Block temp_block;
35
36        if (blocks.get(exponent).size() > 0) {...}
46
47
48        for (index = exponent + 1; index < blocks.size(); index++) {...}
54
55        if (index == blocks.size()) {...}
59
60        temp_block = blocks.get(index).remove( index: 0);
61
62        index--;
63
64        while (index >= exponent) {...}
73
74        System.out.println("\uD83D\uDCE5 Allocate Memory From " + temp_block.getStart() + " -> " + temp_block.getFinish());
75
76        addressList.put(temp_block.getStart(), temp_block.getFinish() - temp_block.getStart() + 1);
77        return new Address(temp_block.getStart(),  size: temp_block.getFinish() - temp_block.getStart() + 1);
78    }
79
```

نمای کلی از تابع allocate

این تابع برای پیدا کردن فضا و اختصاص دادن آن به فرآیندی که آنرا درخواست کرده است .

توضیح نحوه کارکرد تابع :

ابتدا توان 2 ای که سایز مورد درخواست فرآیند است را بدست می آوریم . درواقع این توان 2 برای این است که ما اندیس لیستی که باید درآن دنبال فضا بگردیم را پیدا کنیم . بنابراین این طور میتوان گفت که 2 بتوان اندیس برابر است با سایز بلاک های لیست . حال اگر لیست مورد بلاکی موجود داشت آنرا برای فرآیند رزرو میکنیم و آن را از لیست بلاک های آزاد پاک میکنیم . سپس فضای آدرس مورد نیاز را از بلاک برداشته و در لیست آدرس ها وارد میکنیم . حال اگر بلاکی در لیست مربوط به سایز خودش نبود باید چک کنیم ببینیم که آیا بلاک های بزرگتری وجود دارند که بتوان آنها را شکست ؟ اگه چنین نباشد چون ما لیست بلاک های خالی

را نگه میداریم یعنی کل فضای حافظه ی ما پر شده است و باید خطا چاپ کنیم . اگر بلاک بزرگتر موجود باشد باید آنرا به میزانی که نیاز داریم بشکنیم .پس آن بلاک بزرگتر باید به دو بلاک کوچکتر تقسیم شود و بلاک بزرگتر از لیستش حذف میشود و بلاک های کوچکتر بوجود آمده به لیست مربوط به خود اضافه میشوند . این فرآیند تا جایی تکرار میشود که به سایز مورد نیاز فرآیند برسیم سپس فضای آدرس بلاک را به فرآیند اختصاص میدهیم و آدرس آن را به لیست آدرس های اشغال شده انتقال میدهیم . در نهایت آدرس ابتدای فرآیند را به فرآیند برمیگردانیم .

تابع deallocate :

```
80      public void deallocate(int address) {
81
82          if (!addressList.containsKey(address)) {...}
86
87          int size = (int) Math.ceil(Math.log(addressList.get(address))
88                  / Math.log(2));
89
90          int index, buddy_level, buddy_address;
91
92          blocks.get(size).add(new Block(address,  finish: address + (int) Math.pow(2, size) - 1));
93
94          System.out.println("\uD83D\uDCE4 Memory is released from " + address + " to "
95                  + (address + (int) Math.pow(2, size) - 1));
96
97          buddy_level = address / addressList.get(address);
98
99          if (buddy_level % 2 != 0) {...} else {...}
104
105         for (index = 0; index < blocks.get(size).size(); index++) {...}
128         addressList.remove(address);
129     }
130 }
```

نمای کلی تابع deallocate

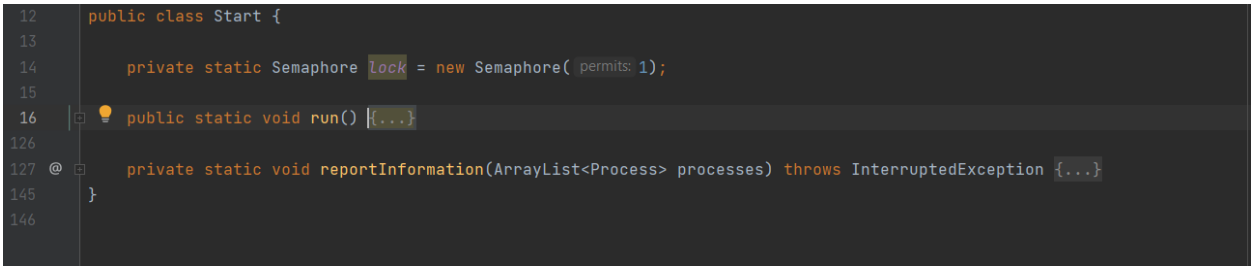
این تابع برای آزاد سازی فضای اشغال شده توسط فرآیند ها استفاده می شود . هر فرآیند آدرس بلاک خود را به این تابع می دهد و این تابع آن آدرس را از لیست آدرس های اشغال شده حذف میکنیم و بلاک آنرا به لیست بلاک های آزاد برمیگردانیم و همچینین اگر بلاک هم سایز مجاور این بلاک هم خالی بود این دو رو با هم ترکیب کرده و بلاک بزرگتری را میسازیم .

توضیح نحوه ی کارکرد تابع :

ابتدا آدرس داده شده توسط فرآیند چک میشود که آیا در لیست اشغال شده ها موجود است یا خیر اگر نبود خطا چاپ میکنیم . در غیر اینصورت آن بلاک را ازاد میکنیم . حال باید چک کنیم که آیا بلاک خالی با همان سایز و در کنار آن وجود دارد که آن دو را با هم ترکیب کنیم یا خیر . برای این کار ابتدا باید ببینیم که این بلاکی که اشغال شده بود در کدام قسمت قرار گرفته است .منظور از قسمت یعنی اگر کل بلاک حافظه را ریشه یک درخت باینری فرض کنیم هر بار که یک بلاک را به دو قسمت تقسیم میکنیم دو فرزند برای والد بوجود میاید در اینجا میخواهیم آدرس ابتدای فرزند دیگر را بیابیم . برای اینکار آدرس ابتدای بلاک را بر سایز آن تقسیم میکنیم اگر عدد حاصل زوج باشد فرزند سمت راست است و اگر فرد باشد فرزند سمت چپ است . با مشخص شدن اینکه کدام فرزند است با اضافه یا کم کردن سایز بلاک از آدرس ابتدای بلاکی آزاد شده است میتوان آدرس ابتدای فرزند دیگر را بدست آورد . حال باید لیست مربوط به آن سایز را بگردیم . اگر بلاکی با آدرس ابتدایی محاسبه موجود باشد یعنی بلاک فرزند هم ارتفاع با بلاک آزاد شده ، آزاد است . پس این دو را به یک بلاک برزگتر تبدیل کرده و دو بلاک دیگر را حذف میکنیم .

در آخر آدرسی که فرآیند آنرا اشغال کرده بود را از مپ حذف میکنیم .

Start.java



در این کلاس هر فرآیند ساخته شده ، زمان بندی شده و اجرا میشود . همچنین گزارشی از نحوه عملکرد سیستم مدیریت حافظه و وضعیت هر فرآیند برای کاربر فراهم آمده و چاپ می شود

نحوه کارکرد تابع Run :

یک نمونه از ماژول مدیریت حافظه ساخته میشود . سپس هر فرآیند ساخته میشود . سپس یک ترد ساخته میشود تا هر کدام از این فرآیند ها را اجرا کند . چون بخش درخاست به سیستم مدیریت حافظه یک بخش بحرانی است به طور یکجا برای آن یک قفل از نوع سمافور گرفته شده است تا دچار ددلاک نشود .

سپس ترد اجرا میشود .

حال برای اینکه این فرآیند ها فقط یکبار اجرا نشوند یک schduledExecuterService درست میکنیم .برای اینکه هر 5 ثانیه ترد گزارش ها اجرا شود . هم این توهم بوجود آید که هر فرآیند چندبار اجرا میشود .

پس از هربار تکرار فرآیند ها را دوباره از اول میسازیم و آنها را اجرا میکنیم . و در آخر تابع reportInformation وظیفه این را دارد که اطلاعات خواسته شده را جمع آوری و گزارش کند .

Config.java

```java
package com.operatingsystem.utils;

public class Config {
    public static final int MAX = 1024;
    public static final int MIN = 32;
    public static final int INITIAL_SIZE = 2048;
}
```

دراین کلاس ثابت های برنامه ما که شامل مقدار مینیمم و ماکزیمم اندازه بلاک ها و اندازه کل قطعه ی حافظه است قرار داده شده است .

RandomValue.java

```java
package com.operatingsystem.utils;

import java.util.Random;

public class RandomValue {

    private int maximum = Config.MAX;
    private int minimum = Config.MIN;

    public int address() { return (int) ((Math.random() * (maximum - 2)) + 2); }

    public int size() { return (int) ((Math.random() * (maximum - minimum)) + minimum); }

    public Boolean bool() {
        Random random = new Random();
        return random.nextBoolean();
    }

    public int time() { return (int) ((Math.random() * (3)) + 0); }

    public int request() { return 1; }
}
```

این کلاس برای تولید اعداد رندم مورد نیاز برای مشخص کردن به تربیت : میزان خوابیدن هر فرآیند ، تولید آدرس تصادفی ، سایز درخواستی رندوم ، درخواست رندم آزاد کردن یا گرفتن حافظه از سیستم عامل ، میزان بیدار بودن هر فرآیند ایجاد شده است .