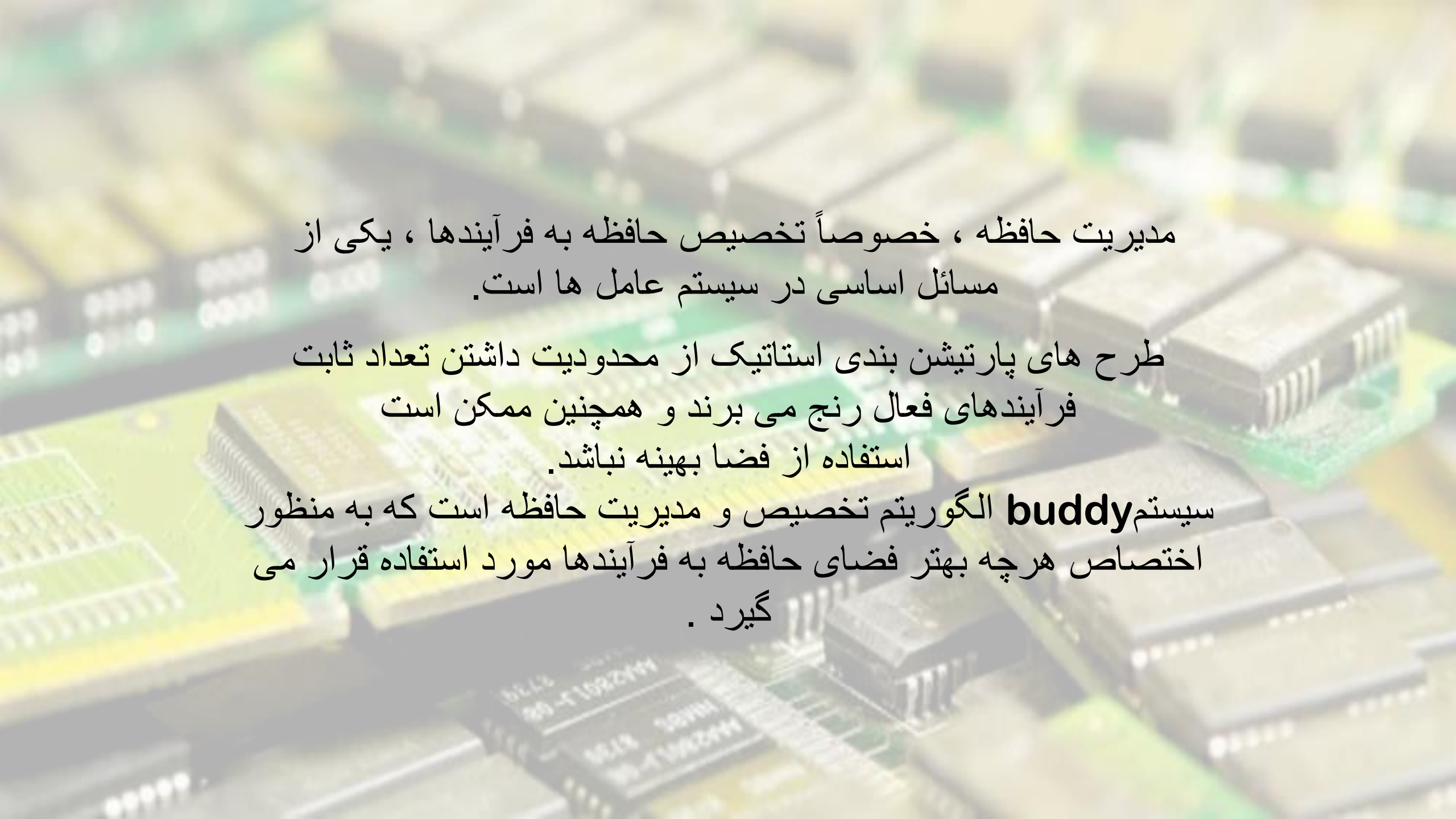


به نام خدا

# پروژه Operating System

استاد درس : دکتر خانمیرزا

ثنا نوری 9731003  
سینا خلیج 9731843



مدیریت حافظه ، خصوصاً تخصیص حافظه به فرآیندها ، یکی از مسائل اساسی در سیستم عامل ها است.

طرح های پارتیشن بندی استاتیک از محدودیت داشتن تعداد ثابت فرآیندهای فعال رنج می برند و همچنین ممکن است استفاده از فضا بهینه نباشد.

سیستم **buddy** الگوریتم تخصیص و مدیریت حافظه است که به منظور اختصاص هرچه بهتر فضای حافظه به فرآیندها مورد استفاده قرار می گیرد .



روش **buddy** اقسام گوناگونی دارد که معمولترین آن ها استفاده از توان های 2 است : هر بلوک به دو بلوک کوچکتر تقسیم می شود تا زمانی که به کوچکترین بلوکی برسیم که فرآیند مورد نظر بتواند در آن قرار گیرد .

هر بلوک حافظه در این سیستم یک نظم دارد ، جایی که ترتیب یک عدد صحیح است که از 0 تا یک حد فوقانی مشخص را در بر می گیرد . اندازه یک بلوک از نظم  $n$  متناسب با  $2^n$  است ، به طوری که بلوک ها دقیقاً دو برابر بلوک هایی هستند که یک مرتبه کمتر هستند . این شیوه مقیاس پذیری بلوک ها را ساده تر می کند .

# مزایا :

- پیاده سازی آسان
- اختصاص صحیح حافظه بلوک ها
- آسان در بهم وصل کردن بلوک های خالی (حفره ها)
- تسریع Allocate و Deallocate

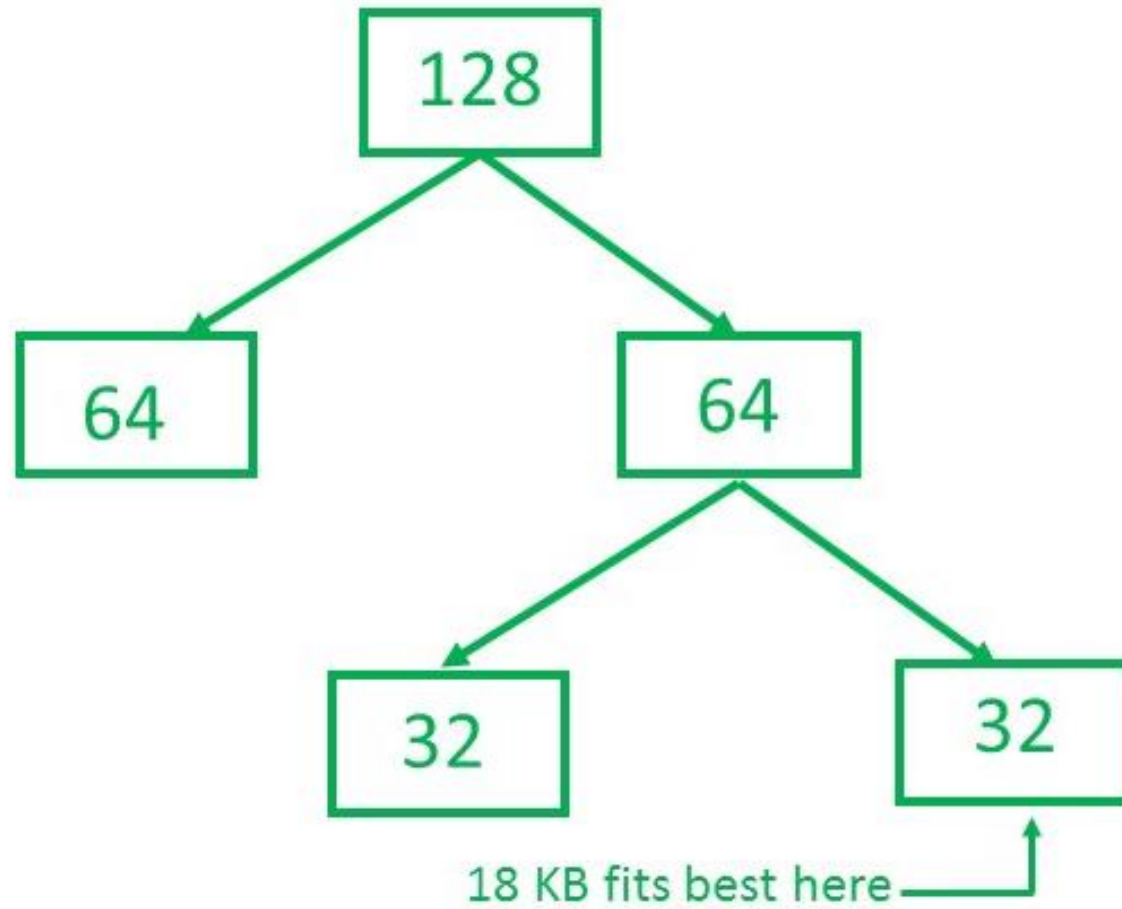
# معایب :

- نیاز به اختصاص صرف بلوک هایی با سایز توان 2

- افزایش تکه تکه شدن داخلی



**Size = 18KB**



# توضیحات کد :



در کلاس **MemoryManagementUnit** روند کلی روش **Buddy** انجام می شود .

دو **Map** برای نگهداری بلاک های استفاده شده **usedBlock** و بلاک های آزاد **freeBlock** در نظر می گیریم که کلید آنها آدرس شروع و مقدار آن ها سایز بلوک می باشد که باید یکی از مقادیر **32,64,128,512,1024** را اختیار کند .

چون این **Map** ها بین تمام **thread** ها مشترک است برای دسترسی به آن ها دو قفل با نام های **usedlistLock** و **freelistLock** تعریف می کنیم .

همچنین قفلی برای چاپ کردن اطلاعات فقط یک **thread** در یک زمان مشخص تعریف می کنیم .

توابع **getter** و **constructor** متناظر کلاس را تعریف کرده و در **constructor** در یک حلقه ی **while** در آغاز کار ، بلاک های **freeBlock** که **1024KB** هستند را اختصاص می دهیم .

```
1 package KNTU;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import java.util.concurrent.locks.ReentrantLock;
6
7 import static KNTU.Process.getProcess;
8
9 public class MemoryManagementUnit {
10     private final ReentrantLock usedListLock;
11     private final ReentrantLock freeListLock;
12     public static final ReentrantLock print = new ReentrantLock();
13     // <beginning address, size>: used blocks -> size can be 1024, 512, 256, 128, 64, 32
14     private final Map<Integer, Integer> usedBlocks = new HashMap<>();
15     // <beginning address, size>: free blocks -> size can be 1024, 512, 256, 128, 64, 32
16     private final Map<Integer, Integer> freeBlocks = new HashMap<>();
17     private final int memorySize;
18
19     public int getMemorySize() { return memorySize; }
20
21     public Map<Integer, Integer> getUsedBlocks() { return usedBlocks; }
22
23     public Map<Integer, Integer> getFreeBlocks() { return freeBlocks; }
24
25     public MemoryManagementUnit(int memorySize) {
26         usedListLock = new ReentrantLock();
27         freeListLock = new ReentrantLock();
28         this.memorySize = memorySize;
29         int remaining = memorySize, address = 0;
30         while (remaining >= 1024) {
31             freeBlocks.put(address, 1024);
32             remaining -= 1024;
33             address += 1024;
34         }
35     }
36 }
37
```



در تابع **allocate** ابتدا **pid** و سائیز مد نظر **thread** را به عنوان ورودی دریافت می کنیم .

دو متغیر برای نگهداری فضای بلاک **spaceOfBlock** (که همان توان های 2 هستند ) و فضای استفاده شده توسط فرآیند **usedSpaceOfBlock** (که هر عددی میتواند باشد )

همچنین یک **Map** برای نگهداری فضای اشغال شده توسط فرآیندها تعریف می کنیم .

در حلقه ی **for** اول چک می کنیم آیا بلوک های از قبل اشغال شده توسط فرآیند ، فضای خالی برای ذخیره اطلاعات جدید دارند یا خیر ؛ اگر فضا داشتند در همان بلاک ها ذخیره می شود و مقدار 1- برگردانده می شود .

در حلقه ی **for** دوم با استفاده از دو متغیر **currentSize** (سائیز بلوکی که میخواهیم به فرآیند اختصاص دهیم) و **sizeFound** (سائیز بلوکی که هم اکنون پیدا کردیم ) کوچکترین سائیز بلوک خالی که **size** را می تواند در خود جای دهد پیدا می کنیم و **address** آن را ذخیره می کنیم .

```
37 public int allocate(int pid, int size) throws Exception {
38     int spaceOfBlock, usedSpaceOfBlock;
39     Map<Integer, Integer> processUsedSpace = getProcess(pid).getUsedSpace();
40     for (Integer usedAddress : processUsedSpace.keySet()) {
41         usedListLock.lock();
42         spaceOfBlock = usedBlocks.get(usedAddress);
43         usedListLock.unlock();
44         usedSpaceOfBlock = processUsedSpace.get(usedAddress);
45         if (spaceOfBlock - usedSpaceOfBlock >= size) {
46             processUsedSpace.remove(usedAddress);
47             processUsedSpace.put(usedAddress, size + usedSpaceOfBlock);
48             return -1;
49         }
50     }
51     int address = 0;
52     int sizeFound, currentSize = 0;
53     freeListLock.lock();
54     for (Integer freeAddress : freeBlocks.keySet()) {
55         sizeFound = freeBlocks.get(freeAddress);
56         if (currentSize < size || (sizeFound >= size && sizeFound < currentSize)) {
57             address = freeAddress;
58             currentSize = sizeFound;
59         }
60     }
61 }
```

```

62 freeListLock.unlock();
63 if (currentSize < size) { // cannot allocate
64     StringBuilder sb = new StringBuilder("cannot allocate memory: process " + pid +
65         " requested " + size + "KB, free spaces: ");
66     freeListLock.lock();
67     for (Integer size1 : freeBlocks.values())
68         sb.append(size1).append("KB, ");
69     if (freeBlocks.size() == 0) {
70         sb.append("none");
71     } else {
72         sb.delete(sb.length() - 2, sb.length());
73     }
74     freeListLock.unlock();
75     throw new Exception(sb.toString());
76 }
77 freeListLock.lock();
78 freeBlocks.remove(address);
79 while (size <= currentSize / 2) {
80     currentSize /= 2;
81     freeBlocks.put(address + currentSize, currentSize);
82 }
83 usedListLock.lock();
84 freeListLock.unlock();
85 usedBlocks.put(address, currentSize);
86 getProcess(pid).usedSpaceLock.lock();
87 getProcess(pid).getUsedSpace().put(address, size);
88 getProcess(pid).usedSpaceLock.unlock();
89 usedListLock.unlock();
90 System.out.println("allocated " + currentSize + "KB to process " + pid + " starting from address " + address);
91 return address;
92 }

```

در شرط **if** اگر سائز بلوکی که می خواهیم به فرآیند اختصاص دهیم از **size** کمتر باشد یعنی در حلقه **for** قبلی نتوانسته ایم بلوک خالی با سائز مناسب پیدا کنیم در نتیجه درخواست رد می شود و حافظه نمی تواند **allocate** شود .

پس از اختصاص فضای مناسب لازم است تا بلوک های آزاد بروزرسانی شوند یعنی بلوک اختصاص یافته به دو زیر بلوک تقسیم می شود که یکی از آن ها خالی است پس به **freeBlocks** اضافه می شود . بلوک بعدی به همین نحو تا انتها ...

همچنین بلوک استفاده شده را به لیست بلوک های اشغال شده **usedBlocks** اضافه کرده ، همچنین به لیست بلوک های اشغال شده خود فرآیند یعنی **usedSpace** اضافه کرده ،

پیام **allocate** را چاپ کرده و در نهایت آدرس محل ذخیره سازی را برمیگردانیم .

```
93  
94 public void deallocate(int pid, int address) {  
95     usedListLock.lock();  
96     int size = usedBlocks.get(address);  
97     usedBlocks.remove(address);  
98     getProcess(pid).usedSpaceLock.lock();  
99     getProcess(pid).getUsedSpace().remove(address);  
100    getProcess(pid).usedSpaceLock.unlock();  
101    usedListLock.unlock();  
102    freeListLock.lock();  
103    combineFreeSpaces(address, size);  
104    freeListLock.unlock();  
105    System.out.println("process " + pid + " deallocated " + size + "KB starting from address " + address);  
106 }  
107
```

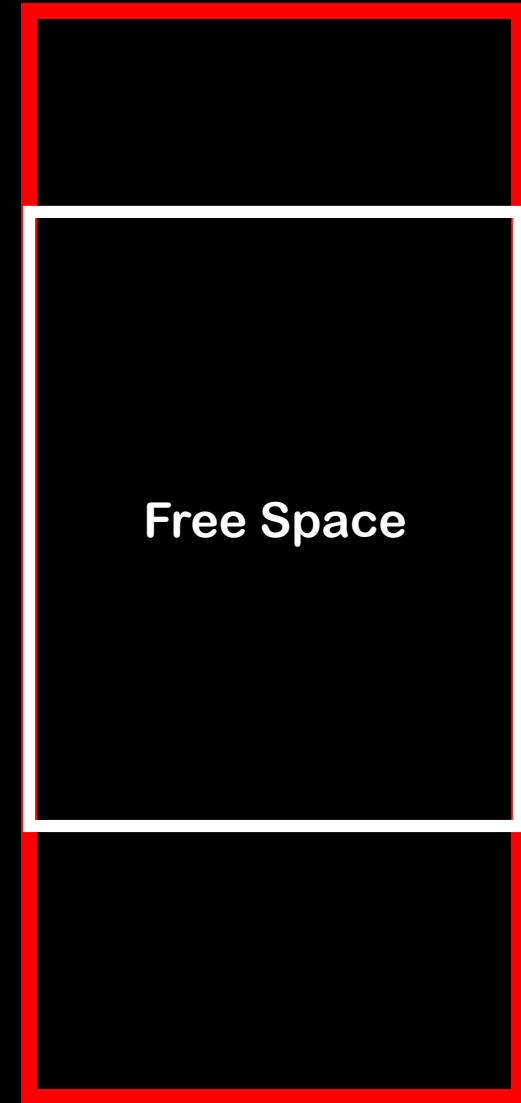
در تابع **deallocate** ، **PID** فرآیند و آدرس جایی که میخواهد **deallocate** شود را به عنوان ورودی به تابع می دهیم .

سایز بلوک انتخاب شده را در متغیر **size** ذخیره می کنیم .

سپس بلوک را از لیست بلوک های اشغال شده **usedBlock** همچنین از لیست فضاهای اشغال شده خود فرآیند یعنی **Map usedSpace** خارج می کنیم .

برای افزودن بلوک آزاد شده به لیست بلوک های آزاد **freeBlocks** ، باید بررسی کنیم که اگر بلوک بالا یا پایین بلوک آزاد شده نیز خالی باشند ، این فضاهای خالی را به هم متصل کنیم . این کار را با استفاده از تابع **combineFreeSpaces** انجام می دهیم .





## Memory ManagementUnit

```
public void combineFreeSpaces(int gAddress, int gSize) {  
    int address_before, size_before, address_after, size_after;  
    while (true) {  
        address_before = 0;  
        size_before = 0;  
        address_after = memorySize;  
        size_after = 0;  
        for (Integer freeAddress : freeBlocks.keySet()) {  
            if (freeAddress < gAddress) {  
                if (address_before <= freeAddress) {  
                    address_before = freeAddress;  
                    size_before = freeBlocks.get(freeAddress);  
                }  
            } else if (freeAddress > gAddress) {  
                if (address_after >= freeAddress) {  
                    address_after = freeAddress;  
                    size_after = freeBlocks.get(freeAddress);  
                }  
            }  
        }  
    }  
}
```

Free Space  
before

deallocate

Free Space  
after

در این این تابع ابتدا آدرس فضای **deallocate** شده و سائز آن را به عنوان ورودی دریافت می کنیم .

**Address\_before** : آدرس بلوک خالی قبلی

**Size\_before** : سائز بلوک خالی قبلی

**Address\_after** : آدرس بلوک خالی بعدی

**Size\_after** : سائز بلوک خالی بعدی

در حلقه های **for** نزدیک ترین بلوک های خالی قبل و بعد از بلوک انتخابی را پیدا می کنیم و متغیرهای بالا را مقداردهی می کنیم . (بلوک های پیدا شده ممکن است متصل به بلوک فعلی نباشند !!!)

```
if (address_before + size_before == gAddress && gSize == size_before && gSize != 1024) {
    freeBlocks.remove(gAddress);
    freeBlocks.remove(address_before);
    if (gAddress + gSize == address_after && 2 * gSize == size_after && size_after != 1024) {
        freeBlocks.remove(address_after);
        freeBlocks.put(address_before, gSize + size_before + size_after);
        gAddress = address_before;
        gSize = gSize + size_before + size_after;
    } else {
        freeBlocks.put(address_before, gSize + size_before);
        gAddress = address_before;
        gSize = gSize + size_before;
    }
}
```

در این قسمت بررسی می کنیم اگر : 1-بلوک قبلی به بلوک **deallocate** متصل باشد و 2-سایز آن ها نیز با هم برابر باشد و 3- سایز آن ها 1024 نباشد(بلوک های بزرگتر از 1024k نداریم) :

ابتدا دو بلوک را از لیست فضاهاى خالى حذف می کنیم

سپس چک می کنیم اگر : 1-بلوک بعدی نیز به بلوک **deallocate** متصل بود و 2-سایز آن نیز به اندازه مجموع بلوک های قبلی و **deallocate** بود یعنی می توان آن ها را بهم متصل کرد پس بلوک پایین را نیز از لیست بلوک های آزاد حذف کرده و بلوک آزاد جدید را (مجموع سه بلوک) به لیست بلوک های آزاد اضافه می کنیم و آدرس و سایز را بروزرسانی می کنیم

در غیر این صورت (نتوانستیم بلوک بعدی را نیز **combine** کنیم) فقط همان بلوک جدید حاصل از بلوک قبلی و **deallocate** را به لیست اضافه می کنیم

Free Space  
before

deallocate

Free Space  
after



```
} else if (gAddress + gSize == address_after && gSize == size_after && gSize != 1024) {  
    freeBlocks.remove(address_after);  
    if (address_before + size_before == gAddress && 2 * gSize == size_before && size_before != 1024) {  
        freeBlocks.remove(address_before);  
        freeBlocks.remove(gAddress);  
        freeBlocks.put(address_before, gSize + size_before + size_after);  
        gAddress = address_before;  
        gSize = gSize + size_before + size_after;  
    } else {  
        freeBlocks.remove(gAddress);  
        freeBlocks.put(gAddress, gSize + size_after);  
        gSize = gSize + size_after;  
    }  
}
```

در این قسمت بررسی می کنیم اگر : 1-بلوک بعدی به بلوک **deallocate** متصل باشد و 2-سایز آن ها نیز با هم برابر باشد و 3-سایز آن ها 1024 نباشد(بلوک های بزرگتر از 1024k نداریم) :

ابتدا دو بلوک را از لیست فضاهاى خالى حذف می کنیم

سپس چک می کنیم اگر : 1-بلوک قبلى نیز به بلوک **deallocate** متصل بود و 2-سایز آن نیز به اندازه مجموع بلوک های بعدی و **deallocate** بود یعنی می توان آن ها را بهم متصل کرد پس بلوک بالا را نیز از لیست بلوک های آزاد حذف کرده و بلوک آزاد جدید را (مجموع سه بلوک) به لیست بلوک های آزاد اضافه می کنیم و آدرس و سایز را بروزرسانی می کنیم

در غیر این صورت (نتوانستیم بلوک قبلى را نیز **combine** کنیم) فقط همان بلوک جدید حاصل از بلوک بعدی و **deallocate** را به لیست اضافه می کنیم

Free Space  
before

deallocate

Free Space  
after

```
} else if (address_before + size_before == gAddress && gAddress + gSize == address_after && size_before == size_after && size_after != 1024) {  
    freeBlocks.remove(address_before);  
    freeBlocks.remove(address_after);  
    freeBlocks.remove(gAddress);  
    if ((2 * size_after == gSize && gSize != 1024) || size_after == 0) {  
        freeBlocks.put(address_before, gSize + size_before + size_after);  
        gAddress = address_before;  
        gSize = gSize + size_before + size_after;  
    } else {  
        freeBlocks.put(address_before, 2 * size_before);  
        freeBlocks.put(gAddress + 2 * size_before, gSize);  
        gAddress = gAddress + 2 * size_before;  
    }  
}
```

Free Space  
before

deallocate

Free Space  
after

حال اگر سایز بلوک قبلی و بعدی با هم برابر باشد ولی سایز آن ها با سایز بلوک **deallocate** برابر نباشد ، میتوان بلوک های قبلی و بعدی را با هم **combine** کرد

حال اگر سایز بلوک **deallocate** به اندازه مجموع سایز بلوک های قبلی و بعدی باشد ، می توان سه بلوک را **combine** کرد

در غیر این صورت فقط بلوک قبلی و بعدی را بهم متصل کرده و فضای آزاد حاصل از **deallocate** را در ادامه بلوک جدید حاصل از بلوک قبلی و بعدی قرار می دهیم .

## Memory ManagementUnit

```
} else {  
    freeBlocks.put(gAddress, gSize);  
    break;  
}
```

در یک حلقه **while** تا زمانی که این فضاهاى خالى بهم پیوسته وجود داشته باشند مراحل بالا را تکرار می کنیم و در صورت عدم برقرارى شرط هاى بالا ، از حلقه خارج می شویم .

Free Space  
before

deallocate

Free Space  
after



## Process\_1

```
import ...
```

```
public class Process implements Runnable {  
    public static Map<Integer, Process> processes = new HashMap<>();  
    public static final ReentrantLock processListLock = new ReentrantLock();  
    private final int pid;  
    private final Map<Integer, Integer> usedSpace; // <beginning address, size>: used space of each block this process has  
    private final long startTime;  
    private long stopTime;
```

یک **ReentrantLock** تعریف می کنیم و برای هر فرآیند  
یک **PID** / یک **Map** از حافظه ی اشغالی هر فرآیند **uesdSpace** / زمان شروع فرآیند **startTime** و  
زمان پایان فرآیند **stopTime** تعریف می کنیم .

```
    public int getPid() { return pid; }
```

```
    public long getStartTime() { return startTime; }
```

```
    public long getStopTime() { return stopTime; }
```

```
    public synchronized static Process getProcess(int pid) { return processes.get(pid); }
```

```
    public Map<Integer, Integer> getUsedSpace() { return usedSpace; }
```

```
    public Process() {  
        processListLock.lock();  
        pid = processes.size() + 1;  
        processes.put(pid, this);  
        processListLock.unlock();  
        usedSpace = new HashMap<>();  
        startTime = System.nanoTime();  
        stopTime = -1;
```

```
    }
```

تابع **getter** متناظر هر متغیر را تعریف کرده و سپس **constructor** را  
تعریف می کنیم

همانگونه که مشاهده می شود به علت دسترسی مشترک همه فرآیندها به **Map**  
**process** برای مقداردهی به این متغیر از قفلی که در بالا  
تعریف کردیم ، استفاده می کنیم .

```
49  
50 public void remove() {  
51     usedSpaceLock.lock();  
52     Map<Integer, Integer> temp = new HashMap<>(usedSpace);  
53     usedSpaceLock.unlock();  
54     if (!usedSpace.isEmpty()) {  
55         for (Integer address : temp.keySet()) {  
56             mmu.deallocate(pid, address);  
57         }  
58     }  
59     processListLock.lock();  
60     processes.remove(pid);  
61     processListLock.unlock();  
62 }  
63
```

وقتی فرآیندی حذف می شود ، باید تمام فضاهایی که از قبل در حافظه داشته خالی کند یا به تعبیری دیگر تمام فضاهای اشغال شده در **usedSpace** را **deallocate** کند .  
در نهایت هم از لیست فرآیندها **Map process** پاک شود .

در تابع **run** متغیرهای :

**address** : آدرس محلی که فرآیند پس از **allocate** در آن قرار می گیرد .

**size** : سایز حافظه ی اختصاص یافته به فرآیند

**num** : متغیری برای ایجاد درخواست های تصادفی **allocate** و **Deallocate**

**count** : تعداد درخواست های هر فرآیند

سپس در یک حلقه به تعداد درخواست ها **count** ، ابتدا یک عدد تصادفی در **num**

ذخیره می کنیم . اگر عدد انتخاب شده به 2 بخش پذیر بود یا حافظه اشغال شده فرآیند صفر

بود (فرآیند در حافظه وجود نداشت) :

یک **size** تصادفی تعریف می کنیم و پس از 1.745 ثانیه ، از کلاس

**MemoryManagementUnit** تابع **allocate** را فراخوانی کرده و

آدرس قرارگیری فرآیند در حافظه را در **address** ذخیره می کنیم .

و اگر عدد تصادفی **num** به 2 بخش پذیر نبود و فرآیند در **memory** وجود داشت :

ابتدا به اندازه 1.745 ثانیه صبر می کنیم سپس به طور تصادفی و با استفاده از

**Map** حافظه های اشغال شده توسط فرآیند (**Map usedSpace**) یکی از حافظه ها را انتخاب کرده و پس از چاپ پیام "فرآیند ...

از حافظه ی ... استفاده کرده است"

مقدار حافظه را با تابع **deallocate** از کلاس **MemoryManagementUnit** پاک می کنیم (**deallocate** میکنیم) .

در نهایت و پس از پاسخدهی به همه ی درخواست های یک فرآیند ، زمان نهایی را ثبت می کنیم .

```
63
64 @Override
65 public void run() {
66     Random random = new Random();
67     int address, size, num, count = random.nextInt( bound: 10);
68     for (int i = 0; i < count; i++) {
69         num = random.nextInt();
70         if (num % 2 == 0 || getUsedSpace().size() == 0) { // allocate
71             try {
72                 Thread.sleep( millis: 1745);
73                 size = random.nextInt( bound: 500) + 20;
74                 address = mmu.allocate(pid, size);
75             } catch (Exception e) {
76                 System.out.println(e.getMessage());
77             }
78         } else { // deallocate
79             try {
80                 Thread.sleep( millis: 1745);
81                 num = random.nextInt(getUsedSpace().size());
82                 address = (int) getUsedSpace().keySet().toArray()[num];
83                 mmu.deallocate(pid, address);
84             } catch (Exception e) {
85                 System.out.println(e.getMessage());
86             }
87         }
88     }
89     stopTime = System.nanoTime();
90 }
91
92 }
```



```
1 package KNTU;
2
3 import ...
4
5
6
7
8
9
10 public class OS implements Runnable {
11
12     @Override
13     public void run() {
14         processListLock.lock();
15         int processes_size = processes.size();
16         processListLock.unlock();
17         while (processes_size > 0) {
18             try {
19                 Thread.sleep(5000);
20             } catch (Exception e) {
21                 e.printStackTrace();
22             }
23             ArrayList<String> outputs = new ArrayList<>();
24             int occupiedSpace = 0, freeSpace = mmu.getMemorySize();
25             int size, internalFrag = 0, processUsed, processOccupied;
26             processListLock.lock();
27             Map<Integer, Process> tempProcesses = new HashMap<>(processes);
28             processListLock.unlock();
29             mmu.usedListLock.lock();
30             if (!mmu.getUsedBlocks().isEmpty()) {
31                 for (Integer used : mmu.getUsedBlocks().values()) {
32                     occupiedSpace += used;
33                 }
34             }
35         }
36     }
37 }
```

در کلاس **OS** هر 5 ثانیه یکبار ، وضعیت فعلی فرایندها را نمایش می دهیم .  
به این منظور با استفاده از ساینز **Map process** در کلاس **Process** ، تعداد فرایندها را در متغیر **process\_size** نگهداری کرده و تا زمانی که فرآیندی برای اجرا داشتیم ، این مراحل را ادامه می دهیم .

به این منظور متغیرهای مناسب را تعریف می کنیم و در ابتدای امر ، با استفاده از متغیر **used** ، مقدار حافظه ی اشغال شده توسط فرایندها یعنی مقدار **UsedBlock** را در متغیر **occupiedSpace** ذخیره می کنیم .

```

35 System.out.println("Total occupied space = " + occupiedSpace);
36 for (Process p : tempProcesses.values()) {
37     processUsed = 0;
38     processOccupied = 0;
39     p.usedSpaceLock.lock();
40     if (!p.getUsedSpace().isEmpty()) {
41         for (Integer address : p.getUsedSpace().keySet()) {
42             size = p.getUsedSpace().get(address);
43             processUsed += size;
44             freeSpace -= size;
45             processOccupied += mmu.getUsedBlocks().get(address);
46             internalFrag += mmu.getUsedBlocks().get(address) - size;
47         }
48     }
49     p.usedSpaceLock.unlock();
50     outputs.add("Process " + p.getPid() + ": \n");
51     outputs.add("\t start time = " + p.getStartTime() + "\n");
52     if (p.getStopTime() == -1) {
53         outputs.add("\t stop time = <still running>\n");
54         outputs.add("\t total runtime = <still running>\n");
55     } else {
56         outputs.add("\t stop time = " + p.getStopTime() + "\n");
57         outputs.add("\t total runtime = " + (p.getStopTime() - p.getStartTime()) + "\n");
58         p.remove();
59         processes_size--;
60     }
61     outputs.add("\t occupied space = " + processOccupied + "\n");
62     outputs.add("\t used space = " + processUsed + "\n");
63 }

```

سپس در یک حلقه برای همه ی فرآیندها ، مقدار حافظه ی اشغالی را در **size** می ریزیم .

-مقدار حافظه ی درخواستی واقعی توسط فرآیندها  
مثلا **processUsed (12KB)**

-مقدار فضای خالی باقی مانده **freeSpace**

-مقدار فضای اشغال شده توسط فرآیند (که ممکن است از تمام آن استفاده نکرده باشد مانند **32KB** برای **12KB processOccupied** )  
و میزان تکه تکه شدن داخلی را محاسبه می کنیم .

سپس اطلاعات فرآیند را چاپ کرده و در صورت کامل شدن اجرای هر فرآیند ، آن را به وسیله تابع **remove** پاک کرده و اندازه **process\_size** را نیز یک واحد کم می کنیم .

```
64 mmu.usedListLock.unlock();
65 System.out.println("Total free space (completely or partially free blocks) = " + freeSpace);
66 System.out.println("Total internal fragmentation = " + internalFrag);
67 System.out.println("Total external fragmentation = " + (mmu.getMemorySize() - occupiedSpace));
68 for (String s : outputs) {
69     System.out.print(s);
70 }
71 }
72 }
73 }
74 |
```

اطلاعات بدست آمده برای هر فرآیند را به ازای هریک ، و یکبار هم اطلاعات کلی شامل فضای آزاد ، تکه تکه شدن داخلی ، تکه تکه شدن خارجی و ... را چاپ می کنیم .

# Main

```
1 package KNTU;
2
3 import java.util.Scanner;
4
5 public class Main {
6     public static MemoryManagementUnit mmu;
7
8     public static void main(String[] args){
9         System.out.print("Enter size of memory in MB: ");
10        Scanner scanner = new Scanner(System.in);
11        int size = scanner.nextInt() * 1024; // in KB
12        mmu = new MemoryManagementUnit(size);
13        System.out.print("Enter number of processes: ");
14        int num = scanner.nextInt();
15        for (int i = 0; i < num; i++) {
16            new Thread(new Process()).start();
17        }
18        new Thread(new OS()).start();
19    }
20 }
21
```

در تابع **Main** ابتدا یک **instance** از کلاس **MemoryManagementUnit** با نام **"mmu"** تعریف می کنیم .

مقدار حافظه (تبدیل آن به **KB** را دستی انجام می دهیم ) و تعداد فرآیندها را از کاربر دریافت کرده و به ترتیب در متغیرهای **size** و **num** ذخیره می کنیم .

سپس در یک حلقه به تعداد فرآیندها ، آن ها را ایجاد می کنیم و به ازای هرکدام یک **Thread** تعریف می کنیم .

سپس یک **instance** از کلاس **OS** ایجاد می کنیم و روند اجرای آن را نیز با **Thread** در پیش می گیریم تا همزمان با عملیات های **Process** ها به اجرای خود بپردازد .



خروجی ها :

```
Enter size of memory in MB: 5
Enter number of processes: 3
allocated 128KB to process 1 starting from address 0
allocated 512KB to process 2 starting from address 512
allocated 512KB to process 3 starting from address 1024
allocated 512KB to process 1 starting from address 1536
allocated 512KB to process 2 starting from address 2048
process 3 deallocated 512KB starting from address 1024
Total occupied space = 1664
Total free space (completely or partially free blocks) = 4095
Total internal fragmentation = 639
Total external fragmentation = 3456
Process 1:
    start time = 403381460999100
    stop time = <still running>
    total runtime = <still running>
    occupied space = 640
    used space = 456
Process 2:
    start time = 403381461151200
    stop time = <still running>
    total runtime = <still running>
    occupied space = 1024
    used space = 569
Process 3:
    start time = 403381461246300
    stop time = <still running>
    total runtime = <still running>
    occupied space = 0
    used space = 0
allocated 512KB to process 1 starting from address 2560
```

**Memory Size = 5MB**  
**NumberofProcess = 3**

```
allocated 512KB to process 1 starting from address 2560
allocated 512KB to process 2 starting from address 1024
allocated 256KB to process 3 starting from address 256
allocated 512KB to process 1 starting from address 3072
allocated 64KB to process 3 starting from address 128
allocated 512KB to process 1 starting from address 3584
allocated 64KB to process 3 starting from address 192
Total occupied space = 4096
process 2 deallocated 512KB starting from address 512
process 2 deallocated 512KB starting from address 2048
process 2 deallocated 512KB starting from address 1024
Total free space (completely or partially free blocks) = 2455
Total internal fragmentation = 1431
Total external fragmentation = 1024
Process 1:
    start time = 403381460999100
    stop time = <still running>
    total runtime = <still running>
    occupied space = 2176
    used space = 1471
Process 2:
    start time = 403381461151200
    stop time = 403386699693900
    total runtime = 5238542700
    occupied space = 1536
    used space = 882
Process 3:
    start time = 403381461246300
    stop time = <still running>
    total runtime = <still running>
    occupied space = 384
```

used space = 312

process 1 deallocated 512KB starting from address 3072

allocated 512KB to process 3 starting from address 2048

process 3 deallocated 64KB starting from address 128

process 1 deallocated 512KB starting from address 3584

Total occupied space = 1984

process 1 deallocated 128KB starting from address 0

process 1 deallocated 512KB starting from address 2560

process 1 deallocated 512KB starting from address 1536

Total free space (completely or partially free blocks) = 3520

Total internal fragmentation = 384

Total external fragmentation = 3136

Process 1:

start time = 403381460999100

stop time = 403395426059500

total runtime = 13965060400

occupied space = 1152

used space = 875

Process 3:

start time = 403381461246300

stop time = <still running>

total runtime = <still running>

occupied space = 832

used space = 725

allocated 256KB to process 3 starting from address 2560

Total occupied space = 1088

process 3 deallocated 256KB starting from address 256

process 3 deallocated 64KB starting from address 192

process 3 deallocated 512KB starting from address 2048

process 3 deallocated 256KB starting from address 2560

Total free space (completely or partially free blocks) = 4238

```
Total internal fragmentation = 384
Total external fragmentation = 3136
Process 1:
    start time = 403381460999100
    stop time = 403395426059500
    total runtime = 13965060400
    occupied space = 1152
    used space = 875
Process 3:
    start time = 403381461246300
    stop time = <still running>
    total runtime = <still running>
    occupied space = 832
    used space = 725
allocated 256KB to process 3 starting from address 2560
Total occupied space = 1088
process 3 deallocated 256KB starting from address 256
process 3 deallocated 64KB starting from address 192
process 3 deallocated 512KB starting from address 2048
process 3 deallocated 256KB starting from address 2560
Total free space (completely or partially free blocks) = 4238
Total internal fragmentation = 206
Total external fragmentation = 4032
Process 3:
    start time = 403381461246300
    stop time = 403397172106800
    total runtime = 15710860500
    occupied space = 1088
    used space = 882

Process finished with exit code 0
```



```
Enter size of memory in MB: 2
Enter number of processes: 4
allocated 512KB to process 2 starting from address 0
allocated 512KB to process 1 starting from address 512
allocated 32KB to process 3 starting from address 1024
allocated 512KB to process 2 starting from address 1536
process 1 deallocated 512KB starting from address 512
cannot allocate memory: process 3 requested 270KB, free spaces: 256KB, 128KB, 64KB, 32KB, 512KB
Total occupied space = 1056
Total free space (completely or partially free blocks) = 1336
Total internal fragmentation = 344
Total external fragmentation = 992
Process 1:
    start time = 403707042104800
    stop time = <still running>
    total runtime = <still running>
    occupied space = 0
    used space = 0
Process 2:
    start time = 403707042274500
    stop time = <still running>
    total runtime = <still running>
    occupied space = 1024
    used space = 684
Process 3:
    start time = 403707042377900
    stop time = <still running>
    total runtime = <still running>
    occupied space = 32
    used space = 28
Process 4:
```

**Memory Size = 2MB**  
**NumberofProcess = 4**

used space = 28

Process 4:

start time = 403707042522400

stop time = 403707043318000

total runtime = 795600

occupied space = 0

used space = 0

process 2 deallocated 512KB starting from address 0

allocated 512KB to process 1 starting from address 0

allocated 512KB to process 3 starting from address 512

process 2 deallocated 512KB starting from address 1536

process 1 deallocated 512KB starting from address 0

allocated 512KB to process 3 starting from address 1536

allocated 128KB to process 2 starting from address 1152

allocated 512KB to process 1 starting from address 0

process 3 deallocated 32KB starting from address 1024

Total occupied space = 1664

Total free space (completely or partially free blocks) = 723

Total internal fragmentation = 339

Total external fragmentation = 384

Process 1:

start time = 403707042104800

stop time = <still running>

total runtime = <still running>

occupied space = 512

used space = 318

Process 2:

start time = 403707042274500

stop time = <still running>

total runtime = <still running>

occupied space = 128

used space = 67

Process 2:

start time = 403707042274500  
stop time = <still running>  
total runtime = <still running>  
occupied space = 128  
used space = 67

Process 3:

start time = 403707042377900  
stop time = <still running>  
total runtime = <still running>  
occupied space = 1024  
used space = 940

process 2 deallocated 128KB starting from address 1152  
allocated 128KB to process 3 starting from address 1024  
cannot allocate memory: process 1 requested 432KB, free spaces: 256KB, 128KB  
allocated 256KB to process 2 starting from address 1280  
cannot allocate memory: process 2 requested 152KB, free spaces: 128KB  
Total occupied space = 1920  
process 1 deallocated 512KB starting from address 0  
process 2 deallocated 256KB starting from address 1280  
process 3 deallocated 512KB starting from address 512  
process 3 deallocated 512KB starting from address 1536  
process 3 deallocated 128KB starting from address 1024  
Total free space (completely or partially free blocks) = 426  
Total internal fragmentation = 298  
Total external fragmentation = 128

Process 1:

start time = 403707042104800  
stop time = 403717523342200  
total runtime = 10481237400  
occupied space = 512  
used space = 310

```
allocated 256KB to process 2 starting from address 1280
cannot allocate memory: process 2 requested 152KB, free spaces: 128KB
Total occupied space = 1920
process 1 deallocated 512KB starting from address 0
process 2 deallocated 256KB starting from address 1280
process 3 deallocated 512KB starting from address 512
process 3 deallocated 512KB starting from address 1536
process 3 deallocated 128KB starting from address 1024
Total free space (completely or partially free blocks) = 426
Total internal fragmentation = 298
Total external fragmentation = 128
Process 1:
    start time = 403707042104800
    stop time = 403717523342200
    total runtime = 10481237400
    occupied space = 512
    used space = 318
Process 2:
    start time = 403707042274500
    stop time = 403721012483800
    total runtime = 13970209300
    occupied space = 256
    used space = 246
Process 3:
    start time = 403707042377900
    stop time = 403717523215700
    total runtime = 10480837800
    occupied space = 1152
    used space = 1058

Process finished with exit code 0
```

```
Enter size of memory in MB: 3
Enter number of processes: 5
allocated 512KB to process 2 starting from address 0
allocated 512KB to process 4 starting from address 512
allocated 64KB to process 1 starting from address 1024
allocated 128KB to process 5 starting from address 1152
allocated 128KB to process 3 starting from address 1280
allocated 512KB to process 4 starting from address 1536
allocated 512KB to process 3 starting from address 2048
process 5 deallocated 128KB starting from address 1152
allocated 32KB to process 1 starting from address 1088
Total occupied space = 2272
process 1 deallocated 64KB starting from address 1024
process 1 deallocated 32KB starting from address 1088
process 2 deallocated 512KB starting from address 0
Total free space (completely or partially free blocks) = 1322
Total internal fragmentation = 522
Total external fragmentation = 800
Process 1:
    start time = 404002608610100
    stop time = 404006162160900
    total runtime = 3553550800
    occupied space = 96
    used space = 78
Process 2:
    start time = 404002608858300
    stop time = 404004414277100
    total runtime = 1805418800
    occupied space = 512
    used space = 509
Process 3:
```

**Memory Size = 3MB**  
**NumberofProcess = 5**



Process 3:

start time = 404002608975900  
stop time = <still running>  
total runtime = <still running>  
occupied space = 640  
used space = 505

Process 4:

start time = 404002609119500  
stop time = <still running>  
total runtime = <still running>  
occupied space = 1024  
used space = 658

Process 5:

start time = 404002609235500  
stop time = <still running>  
total runtime = <still running>  
occupied space = 0  
used space = 0

allocated 512KB to process 5 starting from address 2560  
process 3 deallocated 512KB starting from address 2048  
process 5 deallocated 512KB starting from address 2560  
allocated 128KB to process 3 starting from address 1408  
allocated 256KB to process 5 starting from address 1024  
process 3 deallocated 128KB starting from address 1280  
Total occupied space = 1408  
process 4 deallocated 512KB starting from address 1536  
process 4 deallocated 512KB starting from address 512  
Total free space (completely or partially free blocks) = 2016  
Total internal fragmentation = 352  
Total external fragmentation = 1664  
Process 3:

Process 3:

start time = 404002608975900  
stop time = <still running>  
total runtime = <still running>  
occupied space = 128  
used space = 123

Process 4:

start time = 404002609119500  
stop time = 404007906239800  
total runtime = 5297120300  
occupied space = 1024  
used space = 678

Process 5:

start time = 404002609235500  
stop time = <still running>  
total runtime = <still running>  
occupied space = 256  
used space = 255

allocated 512KB to process 5 starting from address 1536

process 3 deallocated 128KB starting from address 1408

allocated 512KB to process 5 starting from address 2048

Total occupied space = 1280

process 5 deallocated 256KB starting from address 1024

process 5 deallocated 512KB starting from address 1536

process 5 deallocated 512KB starting from address 2048

Total free space (completely or partially free blocks) = 2182

Total internal fragmentation = 390

Total external fragmentation = 1792

Process 5:

start time = 404002609235500  
stop time = 404014888948300

occupied space = 255

Process 5:

start time = 404002609235500

stop time = <still running>

total runtime = <still running>

occupied space = 256

used space = 255

allocated 512KB to process 5 starting from address 1536

process 3 deallocated 128KB starting from address 1408

allocated 512KB to process 5 starting from address 2048

Total occupied space = 1280

process 5 deallocated 256KB starting from address 1024

process 5 deallocated 512KB starting from address 1536

process 5 deallocated 512KB starting from address 2048

Total free space (completely or partially free blocks) = 2182

Total internal fragmentation = 390

Total external fragmentation = 1792

Process 5:

start time = 404002609235500

stop time = 404014888948300

total runtime = 12279712800

occupied space = 1280

used space = 890

Process 3:

start time = 404002608975900

stop time = 404013146006900

total runtime = 10537031000

occupied space = 0

used space = 0

Process finished with exit code 0

|

# تقسیم بندی انجام پروژه

طراحی الگوریتم کد: هر دو نفر به صورت آنلاین  
پیاده سازی کلاس ها و توابع: ثنا نوری  
موازی سازی اجرای ریسمان ها و ایجاد محافظت: سینا خلج  
اجرا و رفع اشکالات کد: هر دو نفر به صورت آنلاین  
تهیه و تنظیم داک: سینا خلج



<https://www.cs.fsu.edu>

<https://en.wikipedia.org>

<https://www.geeksforgeeks.org>

<https://www.transpoco.com>





پایان