**COMP 3410 - Operating Systems (3, 1, 0)**

**Winter, 2018**

**Lab/assignment 7: Main Memory Allocation**

**Problem 1: Memory management and paging**

1. Suppose the page table for a process currently executing on the processor looks like the following. All addresses are memory byte addresses, and addresses in the main memory and processes start from zero. The page size is 512 bytes. (10 marks)

|  |  |  |
| --- | --- | --- |
| **Virtual page number** | **Valid bit** | **Page frame number** |
| 0 | 1 | 4 |
| 1 | 1 | 7 |
| 2 | 0 | 1 |
| 3 | 1 | 2 |
| 4 | 0 | 8 |
| 5 | 1 | 0 |

What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any).

1. 152 – **Page 0 => Frame 4**
2. 1121 – **Page 2** **=> Frame 1 – Page Fault**
3. 2499 – **Page 4 => Frame 8 – Page Fault**
4. Consider the following segment table:

|  |  |  |
| --- | --- | --- |
| **Segment** | **Base** | **Limit** |
| 0 | 219 | 600 |
| 1 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |
| 4 | 1952 | 96 |

|  |  |  |
| --- | --- | --- |
| **Segment** | **Base** | **Limit** |
| 2 | 90 | 190 |
| 0 | 219 | 819 |
| 3 | 1327 | 1907 |
| 4 | 1952 | 2048 |
| 1 | 2300 | 2314 |

All addresses are memory byte addresses, and addresses in the main memory and processes start from zero. What are the physical addresses for the following logical addresses (segment number, offset)? (10 marks)

* 1. 0, 430 = **Segment => 0, Address = 430+219 = 649**
  2. 1, 10 **=** **Segment => 1, Address = 10+2300 = 2310**
  3. 2, 500 **=** **Segment => 2, Address = 500+90 = 590, Error: Exceeds Frame Size ending up in segment 0**
  4. 3, 400 **=** **Segment => 3, Address = 400+1327 = 1727**
  5. 4, 112 **=** **Segment => 4, Address = 112+1952 = 2064, Error: Exceeds Frame Size ending up in segment 1**

1. Consider the following page-reference string in virtual memory management:

1, 2, 8, 3, 4, 2, 1, 5, 6, 2, 1, 3, 7, 6, 3

Assuming four frames, how many page faults would occur for the following replacement algorithms? Remember that all frames are initially empty, so all of your first unique pages will cost one fault each.

**Pages move from top to bottom to display their time spent in queue**

* 1. **LRU replacement**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **8** | **3** | **4** | **2** | **1** | **5** | **6** | **2** | **1** | **3** | **7** | **6** | **3** |
|  | 1 | 2 | 8 | 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 | 3 | 7 | 6 | 3 |
|  |  | 1 | 2 | 8 | 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 | 3 | 7 | 6 |
|  |  |  | 1 | 2 | 8 | 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 | 3 | 7 |
| TO BE REMOVED |  |  |  | 1 | 2 | 8 | 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 | 1 |
| **FAULTS** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **0** | **0** | **1** | **1** | **1** | **0** |

If a page already exists, it is moved to end of queue.

**Total Page Faults = 11**

1. 4 Arrives: At end of queue, 1 Pushed out => 2 8 3 4
2. 2 Arrives: No Replacement, 2 Moves to end of queue => 8 3 4 2
3. 1 Arrives: At end of queue, 8 Pushed out => 3 4 2 1
4. 5 Arrives: At end of queue, 3 Pushed out => 4 2 1 5
5. 6 Arrives: At end of queue, 4 Pushed out => 2 1 5 6
6. 2 Arrives: No Replacement, 2 Moves to end of queue => 1 5 6 2
7. 1 Arrives: No Replacement, 1 Moves to end of queue => 5 6 2 1
8. 3 Arrives: At end of queue, 5 Pushed out => 6 2 1 3
9. 7 Arrives: At end of queue, 6 Pushed out => 2 1 3 7
10. 6 Arrives: At end of queue, 2 Pushed out => 1 3 7 6
11. 3 Arrives: No Replacement, 3 moves to end of queue => 1 7 6 3
    1. **FIFO replacement**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **8** | **3** | **4** | **2** | **1** | **5** | **6** | **2** | **1** | **3** | **7** | **6** | **3** |
|  | 1 | 2 | 8 | 3 | 4 | 4 | 1 | 5 | 6 | 2 | 2 | 3 | 7 | 7 | 7 |
|  |  | 1 | 2 | 8 | 3 | 3 | 4 | 1 | 5 | 6 | 6 | 2 | 3 | 3 | 3 |
|  |  |  | 1 | 2 | 8 | 8 | 3 | 4 | 1 | 5 | 5 | 6 | 2 | 2 | 2 |
| TO BE REMOVED |  |  |  | 1 | 2 | 2 | 8 | 3 | 4 | 1 | 1 | 5 | 6 | 6 | 6 |
| **FAULTS** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **0** |

If a page already exists, it is not moved to top of stack but moved down since its time in memory increases.

**Total Page Faults = 11**

**INITIAL QUEUE = 1 2 8 3**

1. 4 Arrives: At end of queue, 1 Pushed out => 2 8 3 4
2. 2 Arrives: No Replacement
3. 1 Arrives: At end of queue, 2 Pushed out => 8 3 4 1
4. 5 Arrives: At end of queue, 8 Pushed out => 3 4 1 5
5. 6 Arrives: At end of queue, 3 Pushed out => 4 1 5 6
6. 2 Arrives: At end of queue, 4 Pushed out => 1 5 6 2
7. 1 Arrives: No Replacement
8. 3 Arrives: At end of queue, 1 Pushed out => 5 6 2 3
9. 7 Arrives: At end of queue, 5 Pushed out => 6 2 3 7
10. 6 Arrives: No Replacement
11. 3 Arrives: No Replacement
    1. **Optimal replacement**

**Replace the ones that are not used for the longest duration in the future. (LRU but for future)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **8** | **3** | **4** | **2** | **1** | **5** | **6** | **2** | **1** | **3** | **7** | **6** | **3** |
|  | 1 | 2 | 8 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
|  |  | 1 | 2 | 8 | 4 | 4 | 4 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
|  |  |  | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 |
| TO BE REMOVED |  |  |  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **FAULTS** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **1** | **0** | **0** |

1. 4 Arrives: Replaces 8 since 8 is not used again
2. 2 Arrives: No Replacement
3. 1 Arrives: No Replacement
4. 5 Arrives: Replaces 4 since it is not used again
5. 6 Arrives: Replaces 5 since it is not used again
6. 2 Arrives: No Replacement
7. 1 Arrives: No Replacement
8. 3 Arrives: No Replacement
9. 7 Arrives: Replaces 2
10. 6 Arrives: No Replacement
11. 3 Arrives: No Replacement

**Total Faults = 8**

**Problem 2: Main Memory Allocation**

**Goal:** To simulate and evaluate different memory allocation/deallocation techniques, *first fit*, *best fit*, and *worst fit*, when a linked list is used to keep track of memoryusage.

Assume that the memory is 256 KB and is divided into units of 2 KB each. A process may request between 3 and 10 units of memory.

Your simulation consists of three components:

*Memory* component that implements a specific allocation/deallocation technique;

*Request generation* component that generates allocation/deallocation requests;

*Statistics reporting* component that prints out the relevant statistics.

The **Memory component** exports the following functions:

1. int allocate\_mem (int process\_id, int num\_units): allocates num\_units units of memory to a process whose id is process\_id. If successful, it returns the number of nodes traversed in the linked list. Otherwise, it returns -1.

2. int deallocate\_mem (int process\_id): deallocates the memory allocated to the process whose id is process\_id. It returns 1, if successful, otherwise –1.

3. int fragment count (): returns the number of holes (fragments of sizes 1 or 2 units).

You will implement a separate Memory component for each memory allocation/deallocation technique.

The **request generation component** generates allocation and deallocation requests. For allocation request, the component specifies the process id of the process for which memory is requested as well as the number of memory units being requested. For this simulation, assume that memory is requested for each process only once. For deallocation requests, the component specifies the process id of the process whose memory has to be deallocated. For this simulation, assume that the entire memory allocated to a process is deallocated on a deallocation request. You may generate these requests based on some specific criteria, e.g. at random or from a memory allocation/deallocation trace obtained from some source.

There are three performance parameters that your simulation should calculate for all four techniques:

Average number of external fragments;

Average allocation time in terms of the average number of nodes traversed in allocation;

The percentage of times an allocation request is denied.

Generate 10,000 requests using the request generation component, and for each request, invoke the appropriate function of the Memory component for each of the memory allocation/deallocation technique. After every request, update the three performance parameters for each of the techniques. Note we’ll pre load the simulation with 20 tasks with a random selection of between 3 and 10 units of memory.

The statistics reporting component prints the value of the three parameters for all four techniques at the end. What does the simulation tell us about the different memory allocation techniques?

**DRIVER:**

**package** MemoryReplacement;  
  
**import** java.util.Random;  
  
**public class** Driver {  
  
 **static** Random *random* = **new** Random();  
  
 **public static void** main(String[] args) {  
  
 **final int** memorySize = 256;*//KB* **final int** sizePerPage = 2;*//KB  
  
 //WORST FIT MEMORY BLOCKS SIMULATION  
 //-----------------------------------------------------------------------------------------* WorstFitMemoryController worstFitMemoryController = **new** WorstFitMemoryController(memorySize, sizePerPage);  
  
 **int** fails = 0;  
 **int** numberOfHops = 0;  
 **for** (**int** i = 0; i < 10000; i++) {  
 **int** p\_id;  
 **int** requestedSize;  
 p\_id = *generateProcessID*();  
  
 **if** (*toAllocateOrDeallocate*()) {  
 requestedSize = *generateSizeRequest*();  
 numberOfHops+=worstFitMemoryController.findWorstFitMemoryLocation(p\_id, requestedSize);  
 } **else** {  
 **if** (worstFitMemoryController.deallocateFirstFitProcessMemory(p\_id) < 0) {  
 fails++;  
 }  
 **continue**;  
 }  
 *//worstFitMemoryController.printWorstFitMemoryBlocks();* }  
 *//-----------------------------------------------------------------------------------------*

System.***out***.println(**"\n\nBEST FIT FAILS COUNT=> "**+fails);  
 System.***out***.println(**"BEST FIT FRAGMENT COUNT => "**+worstFitMemoryController.fragmentCount());  
 System.***out***.println(**"BEST FIT NUMBER OF HOPS => "**+numberOfHops);  
  
 fails = 0;  
 numberOfHops=0;

*//FIRST FIT MEMORY  
 //-----------------------------------------------------------------------------------------* FirstFitMemoryController firstFitMemoryController =  
 **new** FirstFitMemoryController(memorySize, sizePerPage);  
  
 **for** (**int** i = 0; i < 10000; i++) {  
 **int** p\_id;  
 **int** requestedSize;  
 p\_id = *generateProcessID*();  
  
 **if** (*toAllocateOrDeallocate*()) {  
 requestedSize = *generateSizeRequest*();  
 numberOfHops+=(firstFitMemoryController.findFirstFitMemoryLocation(p\_id, requestedSize));  
 } **else** {  
 **if** (firstFitMemoryController.deallocateFirstFitProcessMemory(p\_id) < 0) {  
 fails++;  
 }  
 }  
 *//firstFitMemoryController.printFirstFitMemoryBlocks();* }  
  
 System.***out***.println(**"\n\nBEST FIT FAILS COUNT=> "**+fails);  
 System.***out***.println(**"BEST FIT FRAGMENT COUNT => "**+firstFitMemoryController.fragmentCount());  
 System.***out***.println(**"BEST FIT NUMBER OF HOPS => "**+numberOfHops);  
 *//-----------------------------------------------------------------------------------------*

*//BEST FIT MEMORY  
 //-----------------------------------------------------------------------------------------* fails = 0;  
 numberOfHops=0;  
 *//NOT BERY EFFECTIVE AND EFFICIENT. MISSE ALLOCATING SOMETIMES WHEN THE MEMORY ALLOCATION WOULD LEAVE A BIG ALLOCATION REMAINDER* BestFitMemoryController bestFitMemoryController = **new** BestFitMemoryController(memorySize, sizePerPage);  
  
 **for** (**int** i = 0; i < 10000; i++) {  
 **int** p\_id;  
 **int** requestedSize;  
 p\_id = *generateProcessID*();  
 **if** (*toAllocateOrDeallocate*()) {  
 requestedSize = *generateSizeRequest*();  
 numberOfHops+=(bestFitMemoryController.findBestFitMemoryLocation(p\_id, requestedSize));  
 } **else** {  
 **if** (bestFitMemoryController.deallocateBestFitProcessMemory(p\_id) < 0) {  
 fails++;  
 }  
 **continue**;  
 }  
 *//bestFitMemoryController.printBestFitMemoryBlocks();* }  
  
 System.***out***.println(**"\n\nBEST FIT FAILS COUNT=> "**+fails);  
 System.***out***.println(**"BEST FIT FRAGMENT COUNT => "**+bestFitMemoryController.fragmentCount());  
 System.***out***.println(**"BEST FIT NUMBER OF HOPS => "**+numberOfHops);  
 }  
  
 **private static int** generateSizeRequest() {  
 **return** *random*.nextInt(8) + 3;  
 }  
  
 **private static boolean** toAllocateOrDeallocate() {  
 *//RETURN TRUE IF TO ALLOCATE* **return** *random*.nextInt() % 2 == 0;  
  
 }  
  
 **private static int** generateProcessID() {  
 **return** *random*.nextInt(21);  
 }  
  
}

**WORST FIT:**

**package** MemoryReplacement;  
  
**public class** WorstFitMemoryController **extends** MemoryController{  
  
 **private** MemoryUnit [] **worstFitMemoryBlocks**;  
 **private int sizePerUnit**;  
  
 **public** WorstFitMemoryController(**int** memorySize, **int** sizePerUnit) {  
 **this**.**worstFitMemoryBlocks** = **new** MemoryUnit[memorySize/sizePerUnit];  
 **this**.**sizePerUnit** = sizePerUnit;  
  
 **for** (**int** i = 0; i < memorySize/sizePerUnit; i++) {  
 **worstFitMemoryBlocks**[i] = **new** MemoryUnit();  
 }  
 }  
  
  
 **public int** findWorstFitMemoryLocation(**int** p\_id,**int** numberOfUnits){  
 *//Variables to measure the startINdex of the empty blocks and its length* **int** worstFitStartIndex = 0;  
 **int** worstFitBlockLength =-1;  
 **int** numberOfJumps = **worstFitMemoryBlocks**.**length**;  
  
 **for**(**int** i = 0; i< **worstFitMemoryBlocks**.**length**; i++){  
 **if** (!**worstFitMemoryBlocks**[i].isAllocated()){  
 **int** j = i;  
 **int** tempBlockLength = 0;  
 *//LOOP THROUGH TO GET THE SIZE OF THE EMPTY BLOCK* **while** (j<**worstFitMemoryBlocks**.**length**&&!**worstFitMemoryBlocks**[j].isAllocated()){  
 tempBlockLength++;  
 numberOfJumps++;  
 j++;  
 }  
 **if**(worstFitBlockLength==-1||tempBlockLength>worstFitBlockLength){  
 worstFitBlockLength = tempBlockLength;  
 worstFitStartIndex = i;  
 }  
 }  
 }  
 *//IF THE LARGEST EMPTY BLOCK IS NOT BIG ENOUGH TO ACCOMODATE REQUIRED UNITS* **if**(worstFitBlockLength<numberOfUnits){  
 **return** -1;  
 }  
  
 **int** index = 0;  
  
 *//ALLOCATE THE MEMORY BLOCKS* **for**(index = worstFitStartIndex;index<worstFitStartIndex+numberOfUnits;  
 index++){  
  
 **worstFitMemoryBlocks**[index].setPARENT\_ID(p\_id);  
 }  
 numberOfJumps+=index;  
 **return** numberOfJumps;  
 }  
 **public int** fragmentCount(){  
 **return super**.fragmentCount(**worstFitMemoryBlocks**);  
 }  
  
 **public int** deallocateFirstFitProcessMemory(**int** pid){  
 **return super**.deallocateProcessMemory(pid,**worstFitMemoryBlocks**);  
 }  
  
 **public void** printWorstFitMemoryBlocks(){  
 **super**.printMemoryBlocks(**worstFitMemoryBlocks**);  
 }  
  
}

**BEST FIT**

**package** MemoryReplacement;  
  
**import** java.util.HashMap;  
  
**public class** BestFitMemoryController **extends** MemoryController {  
  
 **private** MemoryUnit[] **bestFitMemoryBlocks**;  
 **private int sizePerUnit**;  
 **int blocksAllocated** = 0;  
  
 **public** BestFitMemoryController(**int** memorySize, **int** sizePerUnit) {  
  
 **this**.**bestFitMemoryBlocks** = **new** MemoryUnit[memorySize / sizePerUnit];  
 **this**.**sizePerUnit** = sizePerUnit;  
  
 **for** (**int** i = 0; i < memorySize / sizePerUnit; i++) {  
 **bestFitMemoryBlocks**[i] = **new** MemoryUnit(sizePerUnit);  
 }  
  
 }  
  
 **public int** findBestFitMemoryLocation(**int** p\_id, **int** numberOfUnits) {  
 **int** freeBlockStartIndex = 0;  
 **int** prevfreeBlockLength = -1;  
  
 **int** numberOfJumps = **bestFitMemoryBlocks**.**length**;  
  
 **for**(**int** i = 0; i< **bestFitMemoryBlocks**.**length**; i++){  
 **if** (!**bestFitMemoryBlocks**[i].isAllocated()){  
 **int** j = i;  
 **int** tempBlockLength = 0;  
 *//LOOP THROUGH TO GET THE SIZE OF THE EMPTY BLOCK* **while** (j<**bestFitMemoryBlocks**.**length**&&!**bestFitMemoryBlocks**[j].isAllocated()){  
 tempBlockLength++;  
 numberOfJumps++;  
 j++;  
 }  
  
 **if**(prevfreeBlockLength==-1||tempBlockLength<=prevfreeBlockLength&&tempBlockLength>numberOfUnits){  
 prevfreeBlockLength = tempBlockLength;  
 freeBlockStartIndex = i;  
  
 }**else if**(tempBlockLength%numberOfUnits==0||tempBlockLength%numberOfUnits<prevfreeBlockLength%numberOfUnits){  
 **for**(**int** index = freeBlockStartIndex+1;index<=freeBlockStartIndex+numberOfUnits;  
 index++){  
 **bestFitMemoryBlocks**[index].setPARENT\_ID(p\_id);  
 }  
 numberOfJumps+=numberOfUnits;  
 **return** numberOfJumps;  
 }  
 }  
 }  
 *//IF THE LARGEST EMPTY BLOCK IS NOT BIG ENOUGH TO ACCOMODATE REQUIRED UNITS* **if**(prevfreeBlockLength<numberOfUnits){  
 **return** -1;  
 }  
  
 **int** index = 0;  
  
 *//ALLOCATE THE MEMORY BLOCKS* **for**(index = freeBlockStartIndex;index<freeBlockStartIndex+numberOfUnits;  
 index++){  
  
 **bestFitMemoryBlocks**[index].setPARENT\_ID(p\_id);  
 }  
 numberOfJumps+=index;  
 **return** numberOfJumps;  
 }  
  
  
 **public int** deallocateBestFitProcessMemory(**int** pid) {  
 **return super**.deallocateProcessMemory(pid, **bestFitMemoryBlocks**);  
 }  
  
 **public void** printBestFitMemoryBlocks() {  
 **super**.printMemoryBlocks(**bestFitMemoryBlocks**);  
 }  
  
 **public int** fragmentCount() {  
 **return super**.fragmentCount(**bestFitMemoryBlocks**);  
 }  
}

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

**FIRST FIT:**

**package** MemoryReplacement;  
  
**import** java.io.FileNotFoundException;  
**import** java.io.PrintWriter;  
  
**public class** FirstFitMemoryController **extends** MemoryController{  
  
 **private** MemoryUnit [] **firstFitMemoryBlocks**;  
  
 **public** FirstFitMemoryController(**int** memorySize, **int** sizePerUnit) {  
  
 **this**.**firstFitMemoryBlocks** = **new** MemoryUnit[memorySize/sizePerUnit];  
  
 **for** (**int** i = 0; i < memorySize/sizePerUnit; i++) {  
 **firstFitMemoryBlocks**[i] = **new** MemoryUnit(sizePerUnit);  
 }  
 }  
  
  
 **public int** findFirstFitMemoryLocation(**int** p\_id,**int** numberOfUnits){  
 *//printFirstFitMemoryBlocks();* **for**(**int** i = 0;i< **firstFitMemoryBlocks**.**length**;i++){  
  
 **if** (!**firstFitMemoryBlocks**[i].isAllocated()){  
  
 **int** endIndexForAllocation = i + numberOfUnits;  
  
 **if**(endIndexForAllocation>**firstFitMemoryBlocks**.**length**){  
 **return** -1;  
 }  
  
 **int** j;  
 *//Loop through to ensure that there are enough units to accomodate the size* **for**(j =i;j<=endIndexForAllocation&&j<**firstFitMemoryBlocks**.**length**;j++){  
 **if**(**firstFitMemoryBlocks**[j].isAllocated()){  
 **break**;  
 }  
 }  
 *//ENSURE THAT THERE ARE ENOUGH BLOCKS FOR ALLOCATION AND  
 // THEN BEGIN ALLOCATING BACK FROM THE WHERE THE FIRST FREE BLOCK  
 // WAS FOUND* **if**(j>=endIndexForAllocation){  
 **for**(j =i;j<endIndexForAllocation&&j<**firstFitMemoryBlocks**.**length**;j++){  
 **firstFitMemoryBlocks**[j].setPARENT\_ID(p\_id);  
 }  
 **return** i;  
 }  
 }  
 }  
 **return** -1;  
 }  
  
 **public int** deallocateFirstFitProcessMemory(**int** pid){  
 **return super**.deallocateProcessMemory(pid,**firstFitMemoryBlocks**);  
 }  
 **public int** fragmentCount(){  
 **return super**.fragmentCount(**firstFitMemoryBlocks**);  
 }  
 **public void** printFirstFitMemoryBlocks(){  
 **super**.printMemoryBlocks(**firstFitMemoryBlocks**);  
 }  
}

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

*//-------------------------------------------------------------------------------------------*

**MEMORY UNIT:**

**package** MemoryReplacement;  
  
**public class** MemoryUnit {  
  
 *// private int remainder;  
 //private int allocatedSize;* **private int PARENT\_ID**;  
 **private int MAX\_SIZE** = 2;  
 **public** MemoryUnit(){  
 *// remainder = 0;* **PARENT\_ID** = -1;  
 }  
  
 **public** MemoryUnit(**int** sizePerUnit) {  
 **this**.**MAX\_SIZE** = sizePerUnit;  
 *// remainder = MAX\_SIZE;  
 // allocatedSize = 0;* **PARENT\_ID** = -1;  
 }  
  
  
 **public boolean** isAllocated(){  
 **return this**.**PARENT\_ID** !=-1;  
 }  
  
 **public int** getPARENT\_ID() {  
 **return PARENT\_ID**;  
 }  
  
 **public void** setPARENT\_ID(**int** PARENT\_ID) {  
 **this**.**PARENT\_ID** = PARENT\_ID;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Parent = "**+**PARENT\_ID**;  
 }  
}

**MEMORY CONTROLLER:**

**package** MemoryReplacement;  
  
**public class** MemoryController {  
 **public int** fragmentCount(MemoryUnit [] memoryUnits){  
  
 **int** fragmentCount = 0;  
 **for** (**int** i = 0; i <memoryUnits.**length** ; i++) {  
 **if** (!memoryUnits[i].isAllocated()){  
 fragmentCount++;  
 }  
 }  
 **return** fragmentCount;  
 }  
  
 **public void** printMemoryBlocks(MemoryUnit [] memoryUnits){  
 System.***out***.println(**"========================================================================================="**);  
 **for** (**int** i = 0; i < memoryUnits.**length**; i++) {  
 *//if(memoryUnits[i].isAllocated())* System.***out***.println(memoryUnits[i] +**"\t\t\t\tINDEX=> "**+i);  
 }  
 System.***out***.println(**"========================================================================================="**);  
 }  
  
 **public int** deallocateProcessMemory(**int** pid,MemoryUnit [] memoryUnits){  
 **boolean** foundProcessMemory = **false**;  
 **for** (**int** i = 0; i < memoryUnits.**length**; i++) {  
  
 **if**(memoryUnits[i].getPARENT\_ID()==pid){  
 memoryUnits[i].setPARENT\_ID(-1);  
 foundProcessMemory=**true**;  
 }  
 }  
 **return** foundProcessMemory?1:-1;  
 }  
}

**OUT PUT:**

WORST FIT FAILS COUNT=> 2902

WORST FIT FRAGMENT COUNT => 25

WORST FIT NUMBER OF HOPS => 1689345

FIRST FIT FAILS COUNT=> 2818

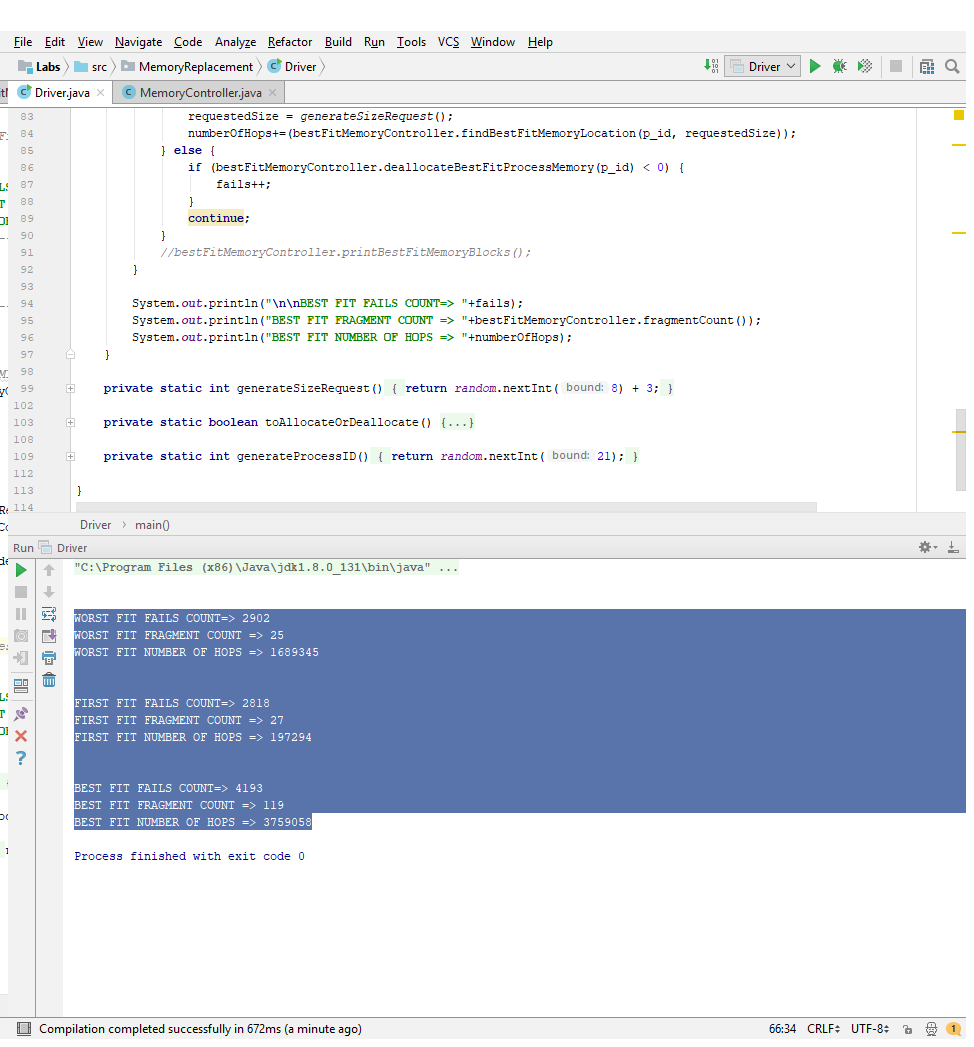
FIRST FIT FRAGMENT COUNT => 27

FIRST FIT NUMBER OF HOPS => 197294

BEST FIT FAILS COUNT=> 4193

BEST FIT FRAGMENT COUNT => 119

BEST FIT NUMBER OF HOPS => 3759058

****