COM1032 Operating Systems
Coursework
60% of the Total Mark

Contents

Deadlines

Date Handed out: **20 March 2023 (Week 7 Monday)**
Deadline: **17 May 2023, 4pm (Week 12)**

Submission: git surrey repo (COM1032) and surrey learn (COM1032)

Late submission penalties will apply as usual according to the regulations.
Academic Misconduct: Coursework will be routinely checked for academic misconduct. The work must be individual. Do not copy.

Overall Task

Your task is to build a simulation, in **Java**, of a **Non-Contiguous Segmented Memory Manager**. You will write a brief report (no more than 5 pages) to explain what you have done, as well as a code submission containing your **Java** project. More information about each of these is provided in their respective section.

- **Non-Contiguous** means that the memory allocator divides memory for a given process into blocks (segments or pages). These can then be placed anywhere in physical (main) memory. The term "main memory" will be used in this document.
- **Segmented**, naturally, means that for those blocks, **segments** will be used instead of pages. The difference between segmentation and paging, is that paging has fixed-sized blocks (every page is always the same size), whereas with segmentation, these blocks may be of various sizes (processes can request segments of some given size).

The following are some tips and information to help you complete this coursework:

- This task is somewhat open-ended. Provided you meet the requirements, you have the freedom to structure your classes and format your logs how you please. **You are encouraged to make use of the labs and lab helpers, which are there to guide you.**
- In **Lab 7 (Part B)** and **Lab 8**, you implemented the base for a *contiguous* memory allocator, you may wish to look at that as a starting point.
- As you are building a simulation, you need to model the main memory in the system, and you will need some way of mapping a process' virtual (logical) memory to main memory. *It is strongly advised* that you make yourself clear on physical (main) and virtual (logical) memory.
- Main memory represents the RAM physically installed in the system; it is separated from virtual memory which is managed by a process. When the process decides that it wants to allocate a segment, it asks the system to place it in main memory, and then stores a reference to it in its segment table. The process' own view of its memory is known as virtual memory because it is an *abstraction* over the real physical memory within the system. The segment table is stored in the process.
- You might need to consider which data structures should be used in places, for example, to represent the system memory, you could use an `ArrayList` or a `HashMap`. There are multiple ways to do things and each may have its own benefits and trade-offs. Select an approach that works for you, and that you understand.
- You may use any code editor or platform to complete your code, however it will be graded using **Eclipse on Windows 10**. It is recommended that you check your work in Eclipse before submission (you can do this on the lab machines).
- **Submit coursework on time using the git repo**. Make sure you submitted (**git push**) the entire coursework content: code and report. If you submit late you will be penalized. You should be familiar on how to use git. **Additionally, you can submit multiple times (using git push) as we'll only check the last submission. Be careful as this is also how we'll decide if your submission is late and if it will be penalized.**
- **Do highlight any issues you have with the git repo well before the coursework deadline**. Your coursework code template will be added to your git repo, make sure you pull before you start. It should already contain your URN (University Registration Number), otherwise known as your candidate number; We'll use it to identify your submission. This is a 7-digit number, usually starting with 6, found on your University ID card, underneath your name.

- **See coursework template**. In the git repo we provide a template code with a parser class. You are free to build on the template code or start from scratch.
- **Finally, if you are unsure of something always ask; do not assume.**

Grading

This coursework has **100 marks** that are split into:

- **Submission** [critical]: surrey git repo and surrey learn.
- **Component A** [25 pts] a report describing the implementation.
- **Component B** [75 pts] a Java program that implements the Segmented Memory Allocation.

Submission

This coursework has TWO components (PDF and code). Your report must be submitted as a **PDF file to SurreyLearn** and your **code must be pushed to the provided GitLab repository** (https://gitlab.surrey.ac.uk/).

For the code, only **your latest commit and push on the main branch will be marked** – so, we strongly encourage you to submit partial solutions as regularly as possible.

Lateness penalties will apply to the entire assessment if either part is submitted late as per university regulations.

Finally, make sure you are using the correct repo (COM1032) when submitting your files.

Component A [25 pts]

You do not need to complete your report first, you would probably begin coding immediately and do your report once you've finished. In general, however, it may be helpful to sketch and plan out on paper how you expect your memory manager to work.

The goal of this report is to explain how your code works, some of your design decisions for your code, and the choices you made. You should comment on what each file in your project does.

Your report must be:

- Submitted as a **PDF** document.
- No more than 5 pages including UML diagram.
- Written in a reasonable font (e.g., Times New Roman or Calibri, 11pt)

Requirements

A.1. **Style [5 pts]** Make your document presentable, e.g., using title, sections or paragraphs

A.2. **Overall Description [10 pts]** Provide an overall description/intuition of the implementation. Describe minimally what we should expect from your implementation, e.g., what is the main file, what each file does (you can use some of the comments in the code), how they link in the system.

The main aim for this is to show how the system looks, and how each class plays a role in the system. You should provide a description for every class, even if that class is trivial. You can however minimally describe methods - for example, for methods used to read and save fields you may simply state that you provide getter and setter methods for that field.

An example, of this is as follows for the `Process.java` class from Lab 8: "`Process.java` is a class that defines a process. There are a number of private fields: `reference_number` which is used to define a process number, `operation` which either allocates (1) or de-allocates (2) the process from within memory, and `argument` for the amount of memory this process will or is currently using. I have defined getter and setter methods for each of these private fields. … "

A.3. **[10 pts]** Use an UML Class diagram to enforce the above point. Be sure you've used the same class names, procedure names, and variable names you mentioned in Task B.1.

Component B [75 pts]

Write a Java program that implements the *Non-contiguous Segmented Memory Allocation Management*.

Note that your submitted program must be self-contained, that means it does not require imports or change of environment paths, etc. Eclipse in Windows 10 will be used for running your program. **Failing to run in Eclipse may result in 0 mark for this component**. It is expected to have **one single package only and one Main.java** for the coursework. That means components B.1, B.2 and B.3 should be in the same package and their examples of should be in the same Main.java.

Component B Tasks

B.1. **[35 pts] Implement a basic Segmented Memory Allocation Management that allows one to add and remove segments from memory.** Here is the overview of tasks to be implemented:

a) Define the main memory with the OS memory and user programs memory; the user program memory is used to store processes/segments.
b) Define processes and segments. Pay attention to also include a segment table for each process.
c) Allocation of processes and segments into memory. Show the changes to the main memory and the process segment table with examples.
d) Deallocation of segments of a process from memory. Show the changes to the main memory and the process segment table with examples.
e) Resize of segments of a process. Show the changes to the main memory and the process segment table with examples.

Marking info. The marking in this section will look at 2 main things: implementation (25pts) and examples (10pts). For the implementation, we look if you have illustrated all the functions needed (see the above note). For the examples, you don't need a separate example, we can base it on the processes used to illustrate the functions; just make sure you allow for a variable number of processes and segments.

In order to differentiate the completed tasks, your program should display messages at the beginning and the end of each task. For task B.1, the following is to be printed. You need to do the same for B.2.1 B.2.2, B.2.3, B.2.4, and B.3.

```
System.out.println ("Start B.1");
… … …
System.out.println ("End B.1");
```

a) Main Memory: the main memory is divided into two parts: Operating System memory and User Programs. For example, a Main `Memory` with 1024 bytes total and OS Memory of 124 can be created. Your program should display the memory state whenever the memory is updated.

```
Memory M = new Memory(1024, 124);
M.memoryState(); // show how the memory looks before operations
```

You should check if the inputs of the Memory constructor are valid, e.g., the size of the OS memory is larger than the whole memory.

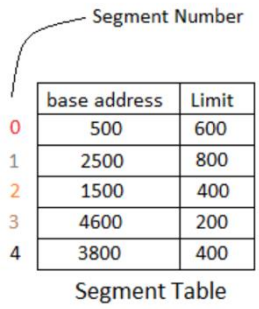
b) Process: A process has unique id (which is a positive number) and is divided into a number of segments. The size of segments is uneven. Each process must handle a variable number of segments – for example to easily change an input `"1, 100, 200, 20"` to `"1, 100, 200, 20, 233"`. For example,

```
Process p1 = new Process("1, 100, 200, 20");
Process p2 = new Process("2, 70, 87, 20, 55");
Process p3 = new Process("3, 10, 260, 40, 10, 70");
```

`Process p1`: "1, 100, 200, 20" refers to Process P1 requiring you to allocate 100 bytes for Segment S0; 200 bytes for Segment S1; 20 bytes for Segment S2. It's expected to have a String as the input of your function, you can use the provided Parser class to read the inputs.

SegmentTable: Segment Table is used to store the information of all segments of the process. Each table entry has:

- Base Address: It contains the starting physical address where the segments reside in memory.
- Limit: It specifies the length of the segment.



	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table

c) Allocation. Define and show an allocation method that adds Processes and segments of process can into the main memory. You need to ensure there is free memory before allocating. You can look at Lab 8 and can potentially use one of the following allocation algorithms: **firstFit, bestFit,** and **worstFit.** Here is an example we provided in the Main function:

```
System.out.println ("Start Component B.1");

// create Main Memory
Memory M = new Memory(1024, 124);
M.memoryState(); // show how the memory looks before operations

// create process examples for tasks B.1
Process p1 = new Process("1, 100, 200, 20");
Process p2 = new Process("2, 70, 87, 20, 55");
Process p3 = new Process("3, 10, 260, 40, 10, 70");

//allocate the first segment from P1 into memory
System.out.println("Add Segment [P1, S0] to Main Memory from Process " + p1.toString());
System.out.println("+ State Before");
```

```
M.memoryState(); // show how the memory looks after the operations
p1.segmentTable(); // show how the segment table looks after the operations

M.allocate (p1, p1.getSegment(0));
// or some variant of this form to add Segment S0 of P1 in main memory.

System.out.println("- State After");
M.memoryState(); // show how the memory looks after the operations
p1.segmentTable(); // show how the segment table looks after the operations


//add Process P1 to Memory. Note that P1.S0 is already in the memory. Each segment can
only be added once.
… … … …
M.allocate(p1); // or some variant of this form to all Segments of P1 in main memory.
… … … …

System.out.println("End B.1");
```

At least 5 processes are created to illustrate the allocation and deallocation of segments and processes and resizing of segments. Remember that each process must handle a variable number of segments – for example to easily change an input "1, 100, 200, 20" to "1, 100, 200, 20, 233". The status of the memory and segment tables should be displayed after the operations.

When your allocation of segments and processes are implemented, your program will produce in the console something of this form:

```
Start Component B.1

Memory State:
[OS 124] | [Hole 900]

Allocating Segment [P1 S0] to Main Memory from Process P1 (1, 100, 200, 20)
+ State before
Mem [OS 124] | [Hole 900]
P1 Segment Table
Sid | base | limit
 0  |      | 100
 1  |      | 200
 2  |      | 20

- State After
Mem [OS 124] | [P1 S0: 100] [Hole 800]
P1 Segment Table
Sid | base | limit
 0  |  124 | 100
 1  |      | 200
 2  |      | 20

… … … … … …
End Component B.1
```

d) Deallocating Processes and Segments. A process or a segment can be deallocated from the memory. You have to check if the process or the segment exists in the memory before the operation. When there is a deallocation operation, the memory and segment table should be updated as required. Deallocation methods can look like these:

```
public int deallocate (Process process) {…} // deallocate a whole process from the memory

public int deallocate (Process process, Segment seg) {…} // deallocate a segment of the process from the memory
```

It returns 1 if successful, -1 otherwise with an error message. The memory state should be displayed to show the updated details. The following provides examples of p1 and a segment deallocated from the memory.

```
System.out.println("Deallocate segment 1 of P1 from the memory");
deallocate(p1, p1.getSegment(0)); // check if segment 0 of p1 is valid and exists in the memory
                    // e.g, deallocate(p1, -1) is not valid
                    // e.g, deallocate(p1, 5) is not valid, as p1 has no segment 5.
M.memoryState();
p1.segmentTable();

System.out.println("Deallocate process " + p1.toString() + "from the memory");
deallocate(p1); // deallocate all segments of process p1 exists in the memory
                    // make sure not to deallocate S0 of P1 again
M.memoryState();
p1.segmentTable();
```

e) Resizing segments of a process. The first allocation of a process creates a baseline, and all other calls to the process will add or remove memory from the segments.

For example, we assume that p2 has been allocated to the memory,

```
p2 = new Process ("2, 70, 80, 20, 50");
```

We then call the function resize ("-30, 40, -10"), which is to update the segments. The updated segments will be: segment S0 size = 70-30=40; segment S1 size = 80+40=120; segment S2 size = 20-10=10; segment S3 = 50.

The size of the segment has to checked, in particular if it becomes zero or negative. It will trigger an error if the size of a segment is less than zero. If the size of a segment is zero, then the segment is to be removed from the memory and update the segment table accordingly.

For another example,

```
p2 = new Process ("2, 100, 200, 10");
```

We then call the function resize ("-40, -200, 10"), the size of segment S1 will be zero, i.e., segment S1 will be removed from p2, whose segment table and memory will be updated.

The following provides an example of resizing a process:

```
System.out.println("Resize Process " + p2.toString());
p2.resize("10, -100, 20, -10");
p2.segmentTable();
M.memoryState();
```

B.2. [20 pts] Implement the following functionalities. B.2 is built on B.1. All functionalities of B.2 can be done independent of each other. **Each subcomponent is 5pts.** We expect for each subcomponent you to provide an example to show this functionality.

Use the following print to highlight what subcomponent you're providing. For example, for B.2.1.

```
System.out.println ("Start B.2.1");
… … …
System.out.println ("End B.2.1");
```

B.2.1. Valid-invalid bit. This is used to indicate if a particular segment is loaded in the Main Memory. This bit is managed by the Process segment table via a new column called **valid-invalid.** When set to `1` (or `true`) for a segment, it shows that segment is loaded into memory (i.e., valid), otherwise if it is `0` (or `false`), that it is not loaded into memory (i.e., the segment is invalid).
This value, **stored in the segment table of a process**, indicates whether a given segment is loaded into main memory (and is therefore a *valid* block of memory that may be used to store information).

If a segment is invalid, (per the valid-invalid bit), then it is not be loaded into main memory. This also means it will not have a memory base address (physical/main memory location) in the segment table.

See the example below.

- **Example**: Process `p1 = new Process ("1, 100, 200");` where S0 is in memory and S1 is not. The segment table of P1 can look something like this:

```
P1 Segment Table
Sid   | base   | limit   | valid-invalid
  0   |    124 | 100     | 1
  1   |        | 200     | 0
```

If you complete this task, don't forget to demonstrate it (by printing the valid-invalid bit for each segment in your memory manager's state, for example).

Tip. If you haven't already, you'll probably want to include methods (e.g., `allocate` and `deallocate`) to load or unload a process' individual segments from memory. One approach might be to add these methods to the Process class, where the method within the Process class would call the Memory class to update the physical memory in the system (i.e., adding or removing the segments from physical memory), followed by the Process class updating its own segment table.

B.2.2. Read-write-execute protection. This is used to limit access to a particular segment. We expect that this information is stored using the segment table. Ensure you provide functions to accept segments with a read, write and execute permissions, check the status of a segment, modify the permission, etc.

To improve security, and reduce the risk of buffer overflows, memory managers introduced the ability for processes to indicate which each segment should be used for. (For instance, this allows a segmentation fault to be thrown if a malicious actor attempts to get a processor to execute its data as code instructions.) -

You should allow each segment to have "permissions" associated with it (and for these to be specified when the segment is created or defined). Where "permissions" refers to '**r**ead**'**, '**w**rite', and/or 'e**x**ecute'. Ideally, you could represent this as a string with "–" for not granting that permission.

- **Example**: Segment [100; r—] means this segment has only read-only permission.
- **Example**: Segment [100; –w–] means this segment has write-only permission.
- **Example**: Segment [100; r–x] has read and execute permission.
- **Example**: Segment [10; rwx] has all permissions.
- **Example**: 1, [100; r—], [200; rw–], [10; rwx] means that Process P1 has 3 segments. Segment S0 has 100 bytes allocated with read-only permission, Segment S1 has 200 bytes under read and write permission. Segment S2 has 10 bytes under read, write and execute permission.

You could trivially check the string for a given permission, by just checking if the string contains the permission you're looking for (either **r**, **w**, or **x**) - or more robustly, checking if the character in a given position within the string matches either a hyphen (–), or the expected character.

You should also provide methods to get and change the segment permissions and to look up a segment in memory (by address) and accept that segment only if it has the correct permissions. (Processes are allowed to change the permissions associated with their segments at any time.)

Hint. It is up to you, but you may wish to consider a segment with no permissions associated (either because the segment had no permissions on creation, or because the segment had all its permissions to be removed) to be an error (as that segment is not useful), however you may wish to consider this as `reserved space' and therefore allow no permissions to be associated.
Whichever you choose, you should document your decision if you considered this.

If you complete this task, don't forget to demonstrate it (by printing the permissions for each segment in your memory manager's state, for example).

B.2.3. Translation Look-Aside Buffer (TLB).

Every time a segment is used, its virtual (logical) memory address (which the process knows and uses), needs to be translated to a physical (main) memory address (that represents the physical location of the data in RAM). For segments that are frequently accessed, this can be *cached* by the memory manager in the Translation Look-Aside Buffer to prevent wasted time on performing this translation.

The memory manager may cache up to *n* (arbitrarily defined) virtual (logical) memory addresses and their equivalent physical (main) memory addresses in the Translation Look-Aside Buffer. (You may choose the value of *n*, for example you might take *n=3* or *n=5* in a system that stores at least *10* segments in total.)

The size of the TLB is a trade-off between the memory or hardware required to hold the TLB vs the performance boost from using the TLB (and for segments that are only accessed once there is no benefit as the translation will need to be calculated for the first time) which is why the memory manager doesn't simply cache every translation that occurs.

Hint. In Java, you would likely choose to implement the TLB as a HashMap (a data structure, often referred to as a *dictionary* in other programming languages, that maps one item - the `key' - to another item - the `value'). Your virtual address (i.e., the key), would probably a composite of the PID and SID (Process ID and Segment ID, respectively). How exactly you choose to do this is up to you.

You will need to demonstrate both a **TLB miss** and a **TLB hit**.

- A **hit** is when a virtual (logical) address (PID and SID) is present as a key within the TLB, and thus a corresponding physical address can be found, avoiding the need to calculate the translation between the virtual (logical) and physical (main memory) addresses.
- A **miss** is, naturally, when this virtual (logical) address (PID and SID) is not in the TLB, meaning the translation between the virtual (logical) address and the physical (main memory) address needs to be calculated. When a miss occurs, you'll add it to the TLB, and if you already have *n* entries in the TLB, you will remove an existing entry (usually, in practice this would be the oldest entry, however you could remove a random entry, as this is arguably just as valid and easier to do in Java).

If you complete this task, don't forget to demonstrate it (by printing TLB miss, followed by TLB hit and TLB miss, for example). Your first TLB miss would be to access some segment, the following TLB hit would be accessing that same segment again, followed by the final TLB miss which would be accessing a different segment.

You would also be expected to show the addresses of the segments for the TLB hit/miss (and/or, ideally, the contents of the TLB.)

B.2.4. Compaction. As segments are allocated and de-allocated, the main memory can become *fragmented* (i.e., the free blocks, or `holes', in memory can be placed sporadically/randomly throughout the physical/main memory).

This could mean that despite there being 550 bytes of memory free in the system, a process would not be able to allocate 550 bytes as there is no *contiguous* block of 550 bytes available. Consider the following example:

```
System Memory: 1024 bytes
OS Reserved Memory: 124 bytes

OS: 124] [P1,S0: 50] [P1,S1: 100] [Hole: 100] [P2, S0: 150][Hole: 300] [P2,S1: 50][Hole: 150]
```

If you add up the size of the `Hole' blocks, you get 550 bytes, but no process can allocate this as a contiguous block of memory because it is split up across physical memory.

To prevent this, and keep all the free memory together so that it can be allocated, by some process, as a contiguous segment (within the overall non-contiguous physical/main memory), **compaction** can be used. This is a means, by which, all of the holes within memory are shifted (i.e., moved to the end) to

form one contiguous block that can then be sub-divided or allocated as a contiguous block by the memory manager.

Your implementation should ensure that this compaction algorithm is called when there are free (but fragmented) memory blocks within the system that can be **compacted** into one free block of memory. ` One example is given as follows:

```
System Memory: 1000 bytes
OS Reserved Memory: 100 bytes

Process 1: 100, 200, 300
Process 2: 100
Process 1: -100, 0, -100 //this is to resize the segments of P1
Process 3: 150, 250
```

The expected result of which, is as follows:

```
System Memory: 1000 bytes
OS Reserved Memory: 100 bytes
[OS: 100] [P1, S0: 100] [P1, S1: 200] [P1, S2: 300] [Hole: 300]
[OS: 100] [P1, S0: 100] [P1,S1: 200] [P1, S2: 300] [P2, S0: 100] [Hole: 200]
[OS: 100] [Hole: 100] [P1,S1: 200] [P1, S2: 200][Hole: 100] [P2, S0: 100][Hole: 200]

Running compaction...
[OS: 100] [P1, S2: 200] [P1, S3: 200] [P2, S1: 100] [Hole: 400]
[OS: 100] [P1, S2: 200] [P1, S3: 200] [P2, S1: 100] [P3, S1: 150] [P3, S2: 250]
```

If you completed this task, do not forget to demonstrate it (by printing your memory state before and after compaction).

B.3. [10 pts] Testing Exceptions.

We will perform two tests, each worth **5 points**. These will check for basic error handling that your implementation should already account for. These are performed on your entire implementation. Some of the exceptions you may find useful are:

1) The OS size with respect to the whole memory size.
2) Enough memory for allocating segments.
3) The size of segment after resizing (its size ≥ 0).
4) The process id should be positive (id ≥ 0).
5) Possible out of bounds (e.g., `ArrayIndexOutOfBoundsException`) – check that when you access an index within an array, that it is indeed a valid index.
6) Possible Reference exceptions – (e.g., `NullPointerException`) – check that you initialise classes/values you should be initializing in a class constructor, and that you initialise classes/values before they're used. (You might, for example, get caught out if you initialise a value in an if-statement, but only in one of the branches, so a value only gets initialised if the condition is true, but not if it is false.)

You can provide the proof of the exception handling within the relevant section. For example, it's in Section B.3:

```
System.out.println ("Start B.3");
… … …
System.out.println ("End B.3");
```

B.4. [10 pts] Useful Code Comments.

You should comment all fields (variables), major methods (functions) – the purpose of getters and setters is trivial and obvious, so you needn't comment those – and classes to explain their purpose and functionality.

To get any marks for this component, you must have first done Component B.1 (Core Program). If any additional components, from B.2 (Additional Features) have been completed, you must also comment those appropriately to get these points. See the Lab 7 solution for an example.

Appendix A – Example Main Method Output

You can download the template code from Surreylearn and modify the code, or start from scratch to complete the coursework. Remember to add comments to describe your code.

The following is an example of a Main class and output we might expect from your implementation. **Your code might look or work differently, your take-away from this should be the example output, and that your code should support the same *functionality*.**

Let's say that your system memory was 1024 bytes total, and your OS memory was 124 bytes, your Main.java might look as follows:

```
package urn0000;
import java.util.Arrays;

public class Main {


    public static void main(String[] args) {

        // you should indicate the start and the end for each task
        System.out.println("Start Component B.1");

        //Create Memory with 1024 bytes total memory, and 124 bytes of OS memory
        // Your code might look differently
        Memory M = new Memory(1024, 124);
        // Show how the memory looks before we do anything
        System.out.println("Initial Main memory state:");
        M.memoryState();

        // create process examples for tasks B.1
        // only 3 examples are shown here for brevity
        Process p1 = new Process("1, 100, 200, 20");
        Process p2 = new Process("2, 70, 87, 20, 55");
        Process p3 = new Process("3, 10, 260, 40, 10, 70");

         // Now, allocate the first segment from P1 into main memory
         // (Segment numbers start at 1.)
         System.out.println("Add Segment [P1, S0] to Main Memory ");
         System.out.println("+ State Before");
        M.memoryState(); // show how the memory looks after the operations
        p1.segmentTable(); // show how the segment table looks after the operations

        M.allocate(p1, p1.getSegment(0));
          // or some variant of this form to add Segment S0 of P1 in main memory

        System.out.println("- State After");
        M.memoryState(); // show how the memory looks after the operations
        p1.segmentTable(); // show how the segment table looks after the operations

        // Next, allocate all segments from P1 into memory
        // (This would skip over the already allocated [P1, S0]
        // Note that P1.S0 is already in the memory. Each segment can only be added once
        System.out.println("Add all segments of P1 to Main Memory .");
        M.allocate(p1);
           // or some variant of this form to all Segments of P1 in main memory.
          // you can add P2 and P3 in here
```

```
            p1.segmentTable();
            M.memoryState(); // show how the memory looks after the operations

            // TODO: continue to complete the rest of the code
        }
}
```

The output, in the console, might look as follows (colours, blank lines, etc., added for clarity):

```
Start Component B.1

Initial Main Memory state
[OS: 124] | [Hole: 900]

Add segment [P1, S0] to Main Memory
[OS: 124] | [P1, S0: 100] | [Hole: 800]
```

(Note, that [P1, S1] does not get re-allocated, as it has already been allocated.)

```
Add all segments from P1 to Main Memory
[OS: 124] | [P1, S0: 100] | [P1, S1: 200] | [P1, S2: 20] | [Hole: 580]
  … more to be done …
End Component B.1
```

This example has been simplified for clarity. In reality, you would be expected to demonstrate this with more processes (at least five) and you would show examples for each of the components (requirements), in turn.