COM1032 Coursework Report

Adam Meor Azlan

For my non-contiguous segmented memory allocation management simulation, I have decided to split up the system into 4 classes: Memory.java, Process.java, Segment.java, and Parser.java. Additionally, the main class is used for testing the previously mentioned classes' implementation.

Class	Description
Memory.java	Used to represent the computer memory, where segments are allocated and de-allocated memory space using a first-fit algorithm.
Process.java	Used to represent a computer process. A process can be split up into multiple variable sized segments, and each segment is stored in the process' segment table.
Segment.java	Used to represent a process' segment in memory, but also used to represent holes in memory, and the computer's OS segment.
Parser.java	Used to parse user input strings into integers. My code relies on integers for class construction and object creation.
Main.java	Used to create memory and process objects, call different methods, to test that the system works as intended.

Short Description of each Class

How the overall simulation works:

Overall, the memory and process classes are meant to be used by the user to create processes and memory. The parser and segment classes are used internally within the system meaning that the user does not have to create any segment or parser objects.

The parser.java class is responsible for turning an input string into a list of numbers. The parser takes in an input string of numbers and characters, separated by a comma or a semicolon which are then converted into numbers. These numbers are then passed into a process or memory object for their own use.

The segment.java class represents the different parts of a process, as well as holes in memory and the operating system's individual segment. A segment can be created by specifying its ID and limit(size). Additionally, there are also setters to set the base address and permissions of the segment if needed. I used segments to represent the holes in memory as well, by setting the limit to 1. For example, if there is a hole in memory of size 100, there will be 100 segment objects with an ID of 'Hole', all with a limit of 1.

The process.java class models a process and can be created by specifying the sizes of its segments through the constructor, which takes in an input string. The string is parsed through the parser.java class, where it turns the input string into a list of numbers which specify the process ID, and

segment sizes and permissions. After a process is created, the user can call on a Memory object and allocate memory space for the process.

The Memory.java class is really the primary class for this simulation. As mentioned before, this class represents the computer memory, where holes are used to represent the free memory space. I chose to represent the memory as an ArrayList, where each ArrayList element represents one byte of memory. The memory starts at memory address 0, which works nicely with the ArrayList as it also starts at index 0. So if the OS requires a segment with 124 bytes, then I will create a segment object and store that segment 124 times in the ArrayList, starting at index 0 and ending at index 123.

segment	OS	OS	OS	OS		OS	OS	
index	0	1	2	3	…	122	123	…

Using an ArrayList to Represent Memory

At each additional index after the OS within the memory's total size, a hole is placed. A hole is just a segment with size 1. For example, if the memory's total size is 1024 and the OS size is 124, every index starting at index 124 until index 1023, will be filled with hole segments.

segment	OS	OS	OS		OS	OS	Hole		Hole	Hole
index	0	1	2	…	122	123	124	…	1022	1023

Using an ArrayList to Represent Memory With Holes

When the program runs, and the user requests for processes and segments to be allocated memory, the memory class will iterate through the ArrayList and find the first space where the segment can be placed in (first-fit algorithm), by counting the number of consecutive holes. The memory class algorithm will iterate through the ArrayList and keep track of the consecutive number of holes. If there exists a space that is large enough for the segment, it will replace the holes with the requested segment object in the ArrayList. For example, if the user requests Segment i (S i) with size 10 to be placed in memory, the memory ArrayList will look like this:

segment	OS	OS		OS	OS	S i	S i		S i	Hole		Hole
index	0	1	…	122	123	124	125	…	133	134	…	1023

Allocating Segment i (S i) in an ArrayList Memory

When the user requests for segment de-allocation, then the memory ArrayList will replace all instances of that segment with holes. The number of holes to place depends on whether the user wants to completely remove the segment from memory, or reduce its memory size. For example, if the user wants to reduce Segment i's memory space by 5, then the memory ArrayList will look like this:

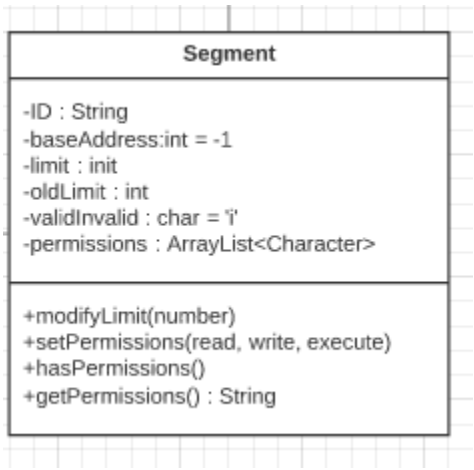
segment	OS	OS		OS	OS	S i	S i		S i	Hole		Hole
index	0	1	…	122	123	124	125	…	128	129	…	1023

De-allocating Memory Space for Segment i (S i) in an ArrayList Memory

Allocating more memory for a segment follows similarly, but it replaces the hole segments at the end with Segment i. If there is no space in memory for a segment to fit in, then the program will print out that the segment has not been allocated memory. A user can print the memory state which will return the memory ArrayList in String format to visualize what is happening.

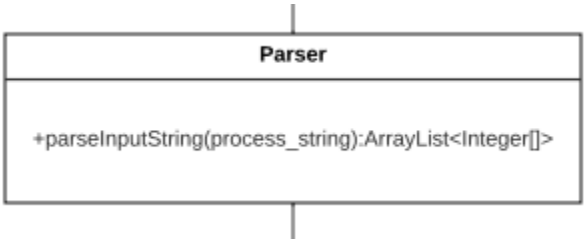
Explaining each class' fields and methods:

Segment.java:



The segment.java class is used internally within the system. It has a number of private variables which are: *ID* - representing the segment ID, *baseAddress* - representing the base address of that segment within the memory, *limit* - representing the segment size, *oldLimit* - representing the previous limit, *validInvalid* - representing the valid-invalid bit, and *permissions* - representing the segment read-write-execute permission. It also has a number of methods which are: *modifyLimit(number)* - used for modifying the size of the segment, *haspermissions()* - used for identifying whether the segment has permissions or not, and also getter and setter for the segment permissions. Note that other getters and setters have been left out from the UML class diagram.

Parser.java:



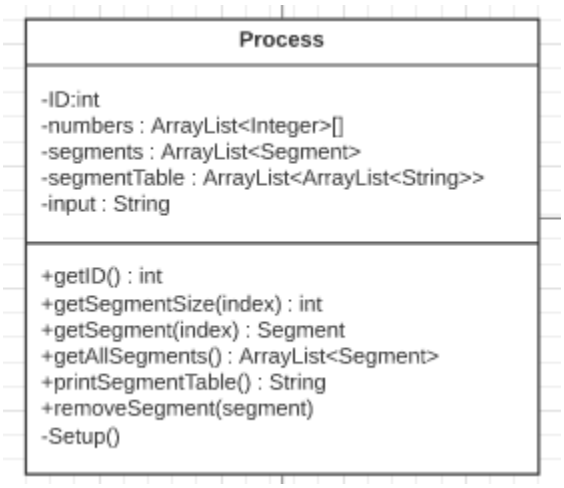
The parser.java class is also used internally within the system. This class only has one method, which is the *parseInputString(process_string)* method. This takes in an input string and returns a 2 dimensional ArrayList of integers. Let's call the 2D ArrayList to be returned, 'comp'. This method splits up the user input on the commas, and splits it again on the semi-colons. Then, each section in between the comma is then transformed into an ArrayList. Essentially 'comp' combines all these ArrayLists making it a 2 dimensional ArrayList. Each individual ArrayList contains a list of numbers, used to represent segment size and permissions. The segment permissions are 3 characters (rwx) which signify the segment's read, write, and execute permissions. If a '-' is present, that means the process does not have that permission. For example, 'rw-' means the process has read and write permissions, but not execute permission. The method adds (to the individual ArrayList within comp) in a 1 if there is permission, and a 0 if there is no permission. For example, for a segment size of 200, with r-x permissions, the individual ArrayList will look like this:

200	1	0	1

Individual ArrayList Example

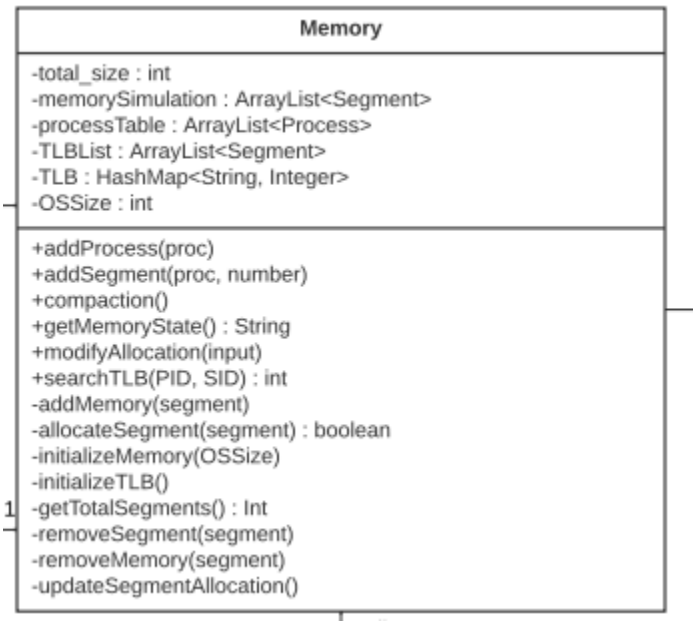
This method also checks on whether or not the given permissions are valid. For example, '-wx' is invalid, as a process cannot execute a segment without having the read permission. In this case, the method will add -1 three times in the individual ArrayList. In the case that the input is unclear, the method will add -3 three times. Overall, the comp ArrayList consists of multiple individual ArrayLists with their own numbers, and the *parseInputString* method returns it.

Process.java:



The process class is used externally, meaning it depends on the user to create process objects. This class has multiple private fields which are: *ID* - representing the process ID, *numbers* - representing the 2 dimensional ArrayList returned by the parser class, *segments* - representing the segments in this process, *segmentTable* - containing information about the segments in this process, *input* - the user input string from the constructor. I have defined some getters, for getting the ID and segments, as well as getting the size of a segment. Other methods include *removeSegment(segment)* - used for removing a segment from this process, *printSegmentTable()* - used for printing the segment table, and *Setup()* - which is a private method used internally within the class for parsing the user input into the *numbers* ArrayList. It then creates segments based on the numbers ArrayList and stores the segments in the *segments* ArrayList.

Memory.java:



The memory class contains a lot of private and public fields and methods which assist it in maintaining a non-contiguous memory type. The private fields are: *total_size* - representing total amount of bytes this memory has, *memorySimulation* - representing the ArrayList for memory, where each element is a byte, *processTable* - containing all the processes currently using memory, *TLBList* - which maintains a list of previously accessed segments, *TLB* - representing the Translation Look-Aside Buffer (TLB), *OSSize* - representing the size of the OS segment. The public methods are *addProcess(proc)* - which allocates memory for all segments in that process which are not already in memory, *addSegment(proc, number)* - which allocates memory for that particular segment, *compaction()* - which performs compaction on the memory and removes external fragmentation, *getMemoryState()* - which prints out the current memory state in a human friendly manner, *modifyAllocation(input)* - which modifies the memory allocation for

certain segments of a process, and *searchTLB(PID, SID)* - which searches the TLB to return the memory location of a particular segment. Additionally, this class has many private methods which assist its public methods. These methods are: *addMemory(segment)* - which allocates more memory for a segment, *allocateSegment(segment)* - which puts in a segment in the *memorySimulation* ArrayList, *initializeMemory(OSSize)* - which initializes the *memorySimulation* by placing in an OS segment and filing in the empty spaces with holes, *initializeTLB()* - which generates the *TLB* HashMap and fills it up with the correct information, *getTotalSegments()* - which returns the total amount of segments in memory, *removeSegment(segment)* - which allocates less memory for a segment, and lastly *updateSegmentAllocation()* - which ensures all segments are correctly placed in *memorySimulation* by checking the segments' base location and limit.

UML Class Diagram: