COM1032 | Operating Systems Report

Simulated Non-Contiguous Segmented Memory Manager

Department of Computer Science

Introduction

The purpose of this coursework was to implement a simulation, in Java, of a non-contiguous segmented memory manager, a program whereby the memory allocator divides memory for a given process into segments, which are placed anywhere in main memory. It utilises segmentation as opposed to paging as paging has fixed-sized blocks, whereas with segmentation, these blocks may be of various sizes. As a result, several classes and methods were created to handle memory allocation, deallocation and segment and process creations

This report will highlight such methods used to create such an implementation and describe the methods and classes I have created to simulate such a program. In order to do this, I intend to go through and describe the functionality of each file, how it operates and the links between classes in order to display them onto a main console.

Creating such a program was no easy task and as such, the report will document key diagrams and modelling and testing techniques used to aid the development of the software.

File Implementation

The program consists of several classes, each responsible for a specific aspect of memory management. These files make up the framework of the program, and each file interacts with another through the use of getters, setters and objects in order to communicate methods and variables across the workspace. There exist six separate classes, each with their own functionality: **Main.java**, **Memory.java, Process.java**, **Parser.java**, **Segment.java**, **SegmentTable.java** and the **Hole.java** file.

The Main class serves as the entry point for the non-segmented memory allocator program. It demonstrates the usage of the memory management operations by creating instances of the Memory, Process, and Parser classes. The class initialises the memory with 1024 bytes and allocates and deallocates memory blocks for different processes and displays the segment table and its contents and the initial memory state. Main.java is also responsible for displaying process and segment creation, memory compaction and resizement of segments within a process and runs tests on memory. This is all done by the initialization of objects to use methods found in other classes.

Memory.java is responsible for managing memory for processes and the segments within. It keeps track of allocated segments, available holes, and a translation lookaside buffer. The class provides methods for allocating and deallocating segments, resizing processes, checking segment validity, setting and getting the valid-invalid bit, and running memory tests. It also includes a compaction algorithm to shift holes and create a contiguous block of memory.

There are a number of private fields. OSsize stores the reserved memory size of the operating system while total_size is used to represent the total size of the memory. A counter exists to monitor translation lookaside buffer hits, counterTLB. The program needs to be able to monitor segments allocated so a map is defined, allocatedSegments that stores the allocated segments, with the key being the segment ID and the value being a boolean true/false flag. Alongside this, listOfHoles is an arraylist that holds the available holes in the memory. I needed a way to map virtual logical memory to physical memory in RAM. Hence, a hashmap TLB was created to represent the TLB entries. A hole object is also initialised to represents a hole in memory.

The memory class holds the majority of methods within the program. It consists of the memory constructor that initialises the memory object with the total memory size and the size of the operating system. I needed a way to allocate and deallocate processes and their individual segments. I decided to use a first fit algorithm due to its simplicity and efficiency, as the search for a suitable block of memory can be performed quickly and easily. Additionally, first fit can also help to minimise memory fragmentation, as it tends to allocate memory in larger blocks.

The method allocate(Process process) is used to allocate a process to memory. It iterates through each segment in the process and calls the allocate(Process process, Segment segment) method for each segment, the method that adds a segment of the process to memory. It searches for a suitable hole, which is free space in memory, to accommodate the segment, updates the allocatedSegments map, and marks the segment as allocated.

Once a process has had all segments allocated and printed onto the segment table, the segments contained within are to be removed to free up space in memory and increase the size of the holes. The deallocate(Process process, Segment segment) method removes a segment of the process from memory. It checks if the segment is allocated, finds the hole containing the segment, and deallocates the segment by updating the data structures, while the deallocate(Process process) method deallocates all segments of a process from memory. It calls the deallocate(Process process, Segment segment) method for each segment and combines adjacent holes after deallocation. The purpose of the addTLB(Process process, Segment segment) method is to declare a TLB hit or miss for a given process and segment. It maintains the TLB hashmap and logs whether the virtual address (logical address) is found in the TLB or not.

Memory is also responsible for resizing and compaction. resizeProcess(Process process) method resizes a process by deallocating its segments and allocating new segments with updated sizes. It also handles the removal of segments with a size of zero. Memory is compacted using the compactMemory() method, a compaction algorithm that shifts all the holes in memory to form a contiguous block. It moves the allocated segments to the beginning of memory, clears the list of holes, and creates a single hole at the end of memory. Adjacent holes are then combined.

The methods isSegmentValid(Process process, Segment segment), setValid(Process process, Segment segment, boolean valid) and isValid(Process process, Segment segment) are all trivial. They check if a segment is valid in the main memory. It verifies if the segment is allocated, loaded into memory, and within the process's address space and returns the corresponding bit value, 0 or 1. Tests are run in memory using runMemoryTests(Process process), a method that runs memory tests to check various conditions and constraints. It performs tests related to the OS size, process IDs, and memory allocation, throwing exceptions or printing error messages if any issues are found. The last method, memoryState(), displays the current state of memory, including the size of the operating system and the available hole sizes to the console.

The Process.java class is responsible for defining a process. A process has a corresponding ID initially zero. with an appropriate getPID() getter and parameters of processString, the string to be processed by the parser. It contains resize(String segments), a method that resizes the segments of the process based on the provided segment sizes. It updates the sizes of the segments in the segmentTable. The class initialises SegmentTable and Segment objects and as result, they have setters and getters respectively. toString() is a method that I made to return a string representation of the process, including its process ID and segment details.

Within the main class, processes are initialised objects with parameters of a string that lists the limit size of each segment and the number of segments allocated within a given process alongside permissions.

Hence, it was important to include a parser class responsible for parsing a process input string and extracting its components by creating a method parseInputString(String process_string) that takes a process input string as input and parses it to extract the process ID and a list of segment sizes with optional parameters and returns an array of ArrayList<String> objects with the first element corresponds to the process ID, and subsequent elements contain the segment sizes and permissions.

The Segment class is responsible for defining a segment, otherwise known as a block of memory that requires allocation. It has five variables with corresponding getters and setters. A segment is defined by an ID S#, with the initial segment within a process starting at 0. It also has a base address initially set to -1 and permissions associated with the segment, initially set to "---" which corresponds to no permissions.
Three methods check to see if the segment has read, write or execute permissions, hasReadPermission(), hasWritePermission() and hasExecutePermission(). A boolean flag isValid that is initially set to false and a corresponding method isValid(), indicating whether the segment is valid or not and returns 1 if the segment is valid and 0 if it is not. Segment.java has two constructors, one for segments that contain no permissions and another for when permissions are present. The toString() method returns a string representation of the segment, including its ID, base address, size, validity, and permissions.

The SegmentTable class is responsible for defining the segment table of a process. It contains one defining variable, an arraylist segment that contains all segments within a process and initialises it within the constructor. SegmentTable has one main method that sets the validity status of the segment given the ID of the segment, and iterates through each segment in the segment table, finds the segment with the matching ID, and sets its validity status. If no segment is found with the given ID, it prints an error message. However, it contains appropriate getters getSegment(int id) and getSegments() that return the list of segments. The most important method within the class is the toString() method which returns a string representation of the segment table by iterating through each segment in the table and concatenating their string representations.

The Hole class is responsible for storing the size and range of a hole in memory. A hole is a free space within memory that processes are allocated to and their respective segments. Three variables define a hole: the start address of the hole, end, the end address of the hole and the segment that occupies the hole if any. These three variables are initialised within the class constructor and have their own respective getters and setters. A crucial method within the class, setRange(int start, int end) sets the range of the start and end addresses of the hole, allowing for holes to vary in size.

File Design and UML Diagram

The files are designed in a way that promotes modularity and separation of concern and are structured in a way to work together to simulate a memory management system. For example, the Main class interacts with the memory class to perform memory operations. The memory class serves as the central component, coordinating operations between processes, segments, and available memory holes while managing the segment tables.. Each Process has its own segment table, and each segment is associated with a process. The parser class assists in parsing the input string and creating Process object and the Hole class represents available memory gaps within the memory class.

However, this can still be difficult to visualise. I have developed a UML diagram to illustrate the design and relationships between the classes in the memory management system. It provides a visual representation of the system's structure, helping to understand the overall architecture and interactions between the components:

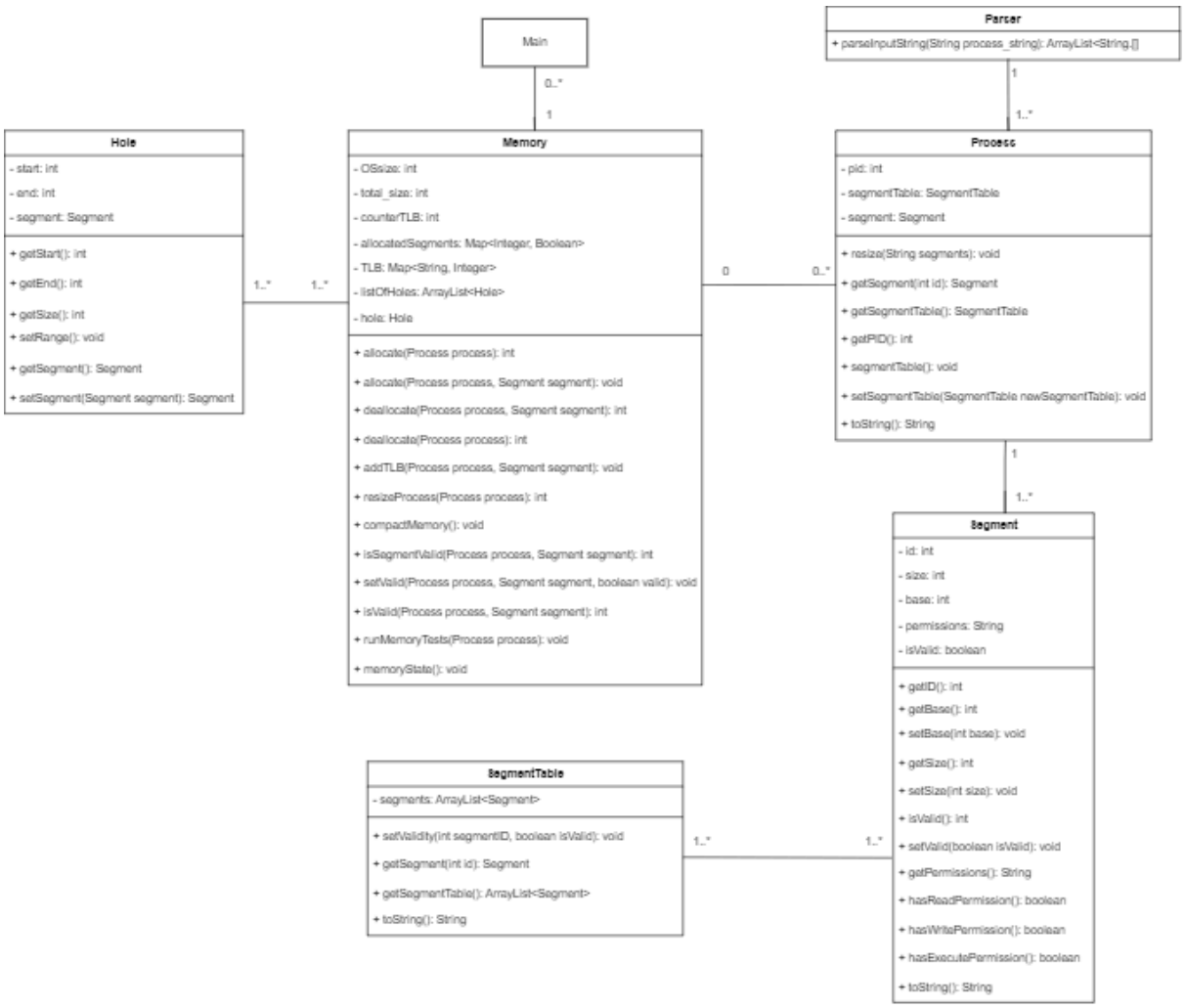

Figure 1: Non-Contiguous Segmented Memory Allocator UML Diagram