COM1032: Operating Systems

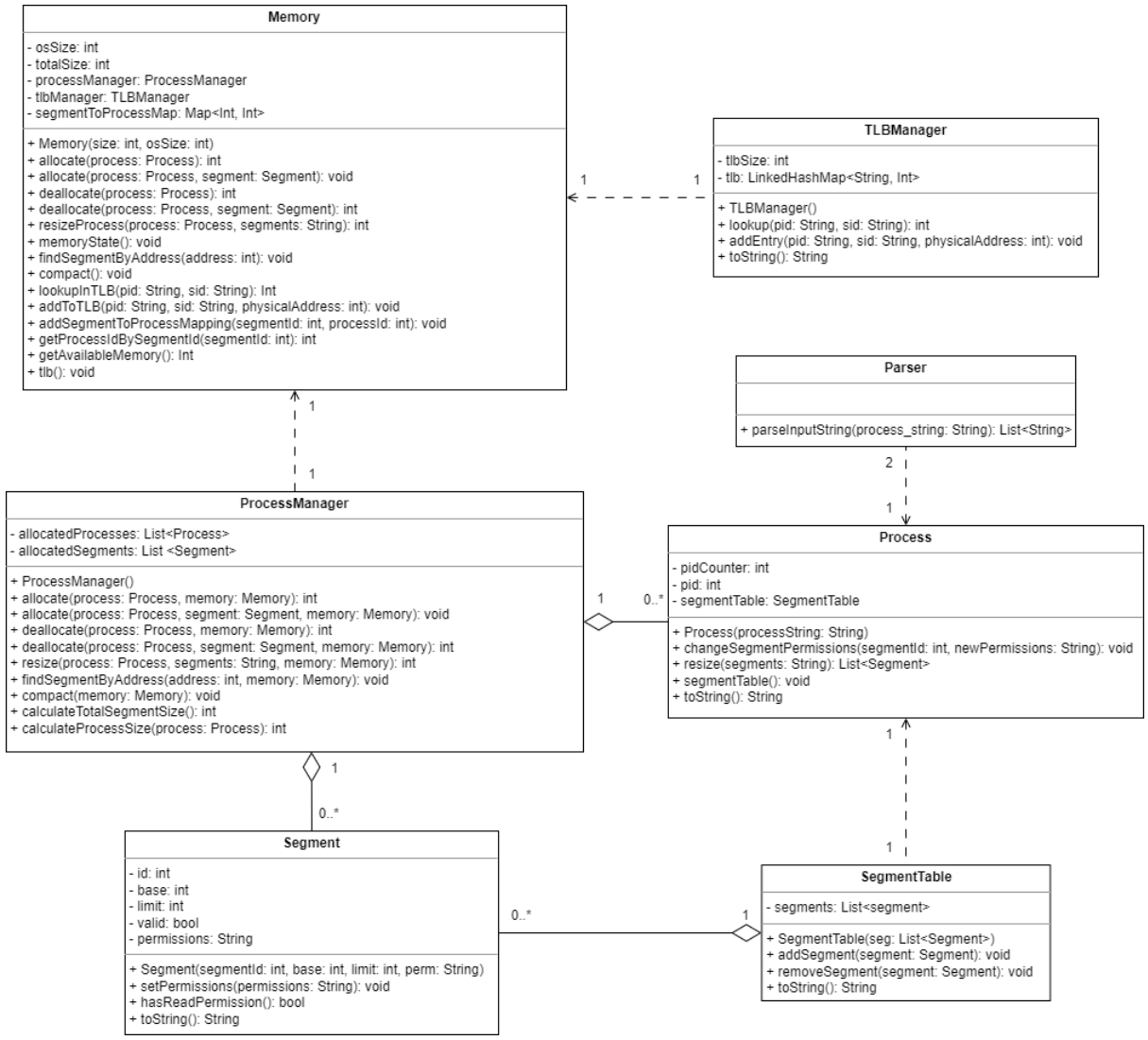
Coursework Assignment

Non-Contiguous Segmented Memory Manager

URN	
Username	
Date	**17/05/2023**

Overall Implementation

My implementation achieves the goal of making a non-contiguous segmented memory manager. I have done this using several classes: 'Memory', 'ProcessManger', 'Segment', 'TLBManager', 'Parser', 'Process', and 'SegmentTable'. My solution is intended to be used in conjunction with a main class that will make the Process and memory objects, and the system is used there. Here is the UML class diagram to help explain my solution. *UML Class Diagram*


Memory
- osSize: int
- totalSize: int
- processManager: ProcessManager
- tlbManager: TLBManager
- segmentToProcessMap: Map<Int, Int>
+ Memory(size: int, osSize: int)
+ allocate(process: Process): int
+ allocate(process: Process, segment: Segment): void
+ deallocate(process: Process): int
+ deallocate(process: Process, segment: Segment): int
+ resizeProcess(process: Process, segments: String): int
+ memoryState(): void
+ findSegmentByAddress(address: int): void
+ compact(): void
+ lookupInTLB(pid: String, sid: String): Int
+ addToTLB(pid: String, sid: String, physicalAddress: int): void
+ addSegmentToProcessMapping(segmentId: int, processId: int): void
+ getProcessIdBySegmentId(segmentId: int): int
+ getAvailableMemory(): Int
+ tlb(): void
TLBManager
- tlbSize: int
- tlb: LinkedHashMap<String, Int>
+ TLBManager()
+ lookup(pid: String, sid: String): int
+ addEntry(pid: String, sid: String, physicalAddress: int): void
+ toString(): String
1
1
Parser
+ parseInputString(process_string: String): List<String>
2
1
ProcessManager
- allocatedProcesses: List<Process>
- allocatedSegments: List
+ ProcessManager()
+ allocate(process: Process, memory: Memory): int
+ allocate(process: Process, segment: Segment, memory: Memory): void
+ deallocate(process: Process, memory: Memory): int
+ deallocate(process: Process, segment: Segment, memory: Memory): int
+ resize(process: Process, segments: String, memory: Memory): int
+ findSegmentByAddress(address: int, memory: Memory): void
+ compact(memory: Memory): void
+ calculateTotalSegmentSize(): int
+ calculateProcessSize(process: Process): int
1
0..*
Process
- pidCounter: int
- pid: int
- segmentTable: SegmentTable
+ Process(processString: String)
+ changeSegmentPermissions(segmentId: int, newPermissions: String): void
+ resize(segments: String): List
+ segmentTable(): void
+ toString(): String
1
1
0..*
Segment
- id: int
- base: int
- limit: int
- valid: bool
- permissions: String
+ Segment(segmentId: int, base: int, limit: int, perm: String)
+ setPermissions(permissions: String): void
+ hasReadPermission(): bool
+ toString(): String
0..*
1
SegmentTable
- segments: List
+ SegmentTable(seg: List)
+ addSegment(segment: Segment): void
+ removeSegment(segment: Segment): void
+ toString(): String

Segment

The Segment class models a segment of a process. The fields include id (or SID), base, limit, valid, and permissions. Various methods are employed to ensure functionality:

Segment(): A constructor that initialises the base address, limit, permissions, and SID from the parameters.

setPermissions(): Sets the permissions new of the segment, validating that new permissions are of the type: 'read-only', 'write-only', 'read and execute', and 'read, write, and execute', which are valid.

hasReadPermission(): Checks if the segment has read permission, returning true if it does and false otherwise.

toString(): Returns the name of the segment in the "S[x]" format, where "S" denotes a segment, and "[x]" is the SID.

Getters and setters: Aid in the functionality of the Segment class, with validation in the setters to ensure the new variables are acceptable.

SegmentTable

The SegmentTable class generates a table of segments belonging to a process. It has one field, segments, and the constructor SegmentTable() initialises segments, a list of them in a process. The SegmentTable class also includes the following methods: addSegment(): Adds segments to the segments list. removeSegment(): Removes segments from the segments list. toString(): Returns the table of segments in a readable format.

Getters: Aid in functionality with other classes.

Process

The Process class models a process in the system. The fields include pidCounter, pid, and segmentTable. The pidCounter ensures each Process has a unique pid, and segmentTable stores the segment table from the SegmentTable class. The Process() constructor initialises all fields and employs the Parser class to deconstruct the process string to instantiate new Segment objects. The Process class also includes the following methods:

changeSegmentPermissions(): Updates the permissions of a specific segment.

resize(): Resizes all segments belonging to the Process, removing any with a new limit of zero.

segmentTable() and toString(): Output the segment table to the console and return the name of the Process, respectively.

Getters: Aid in the functionality of other classes.

Parser

The Parser class, used in the Process class, extracts parts of the process string using commas and semicolons as delimiters.

TLBManager

The TLBManager class manages the TLB (Translation Lookaside Buffer), a memory cache that reduces access time to user memory locations. It includes a tlbSize constant defining the TLB's size, a tlb data structure (LinkedHashMap) that maps a string key to an integer (physical address), and methods for initialising the TLB, retrieving physical addresses, adding new entries, and returning a string representation of the TLB.

ProcessManager

The `ProcessManager` class is a valuable tool for managing the allocation and deallocation of processes and their segments in memory. It has a variety of methods to handle these operations.

During instantiation, using the `ProcessManager()` constructor, two `ArrayList` instances are initialised: `allocatedProcesses` and `allocatedSegments`. These lists store the currently allocated processes and segments, respectively.

The `allocate(Process process, Memory memory)` method allocates a given process and all its segments into memory. First, it checks whether the Process is allocated or enough memory is available. If either condition fails, it returns an error. On the other hand, if the Process can be allocated, it allocates each segment to memory, setting the segment as valid and adjusting its base memory address.

The `allocate(Process process, Segment segment, Memory memory)` method differs from the previous, respective method. It handles the allocation of a specific segment from a given process into memory.

The `deallocate(Process process, Memory memory)` method handles the deallocation of a given process and all its segments from memory. First, it checks if the Process has been allocated, and if so, it removes each segment from memory, invalidating them, removing them from the allocated segments list, and resetting their base memory address. After deallocating the Process, it compacts memory to fill in gaps.

The `deallocate(Process process, Segment segment, Memory memory)` method differs from the previous, respective method. It handles the deallocation of a specific segment from a given process from memory.

The `resize(Process process, String segments, Memory memory)` method resizes a given process based on new limits for its segments. It first checks if the Process has been allocated and then resizes the segments. Segments that need to be removed from memory are dealt with accordingly.

The `findSegmentByAddress(int address, Memory memory)` method uses a memory address to look for a segment in memory. It finds any segment with the given address within its range and checks if it has the proper permissions.

The `compact(Memory memory)` method compacts the loaded processes or segments to fill gaps. It sorts the segments based on their base memory address and then adjusts the base addresses to fill in gaps in memory.

The `calculateTotalSegmentSize()` and `calculateProcessSize(Process process)` methods calculate the total memory size used by all allocated segments and a specific process, respectively.

Finally, the `getAllocatedProcesses()` and `getAllocatedSegments()` methods return the list of currently allocated processes and segments, respectively.

Memory

The Memory class is central to memory management within a system. It contains several attributes to facilitate this, including the Operating System (OS) size, the total size of the system, and a ProcessManager to manage the processes. It also includes a TLBManager to

manage the Translation Lookaside Buffer (TLB) and a Map to associate segment IDs with process IDs.

The class has a constructor that instantiates a Memory object with the system's size and the OS's size. The constructor also performs checks to ensure the validity of these sizes, throwing an IllegalArgumentException if either the system size or OS size is invalid. Upon successful instantiation, ProcessManager and TLBManager objects are created.

The Memory class provides a variety of methods for memory management. The allocate and deallocate methods manage the allocation and deallocation of processes and their respective segments. The resizeProcess method allows for resizing a process with new limits for the segments.

The class also includes a method to output the current state of memory to the console, memoryState, and a method findSegmentByAddress to locate a segment in memory using a memory address. There is also a compact method that compacts loaded processes/segments to fill in any gaps.

The lookupInTLB and addToTLB methods allow for interaction with the TLB, either to lookup a combined Process ID (PID) and Segment ID (SID) in the TLB, or to add a PID, SID, and physical address to the TLB.

The addSegmentToProcessMapping method ensures the correct mapping between segments and processes, while the getProcessIdBySegmentId method retrieves the process ID associated with a specific segment. Lastly, the getAvailableMemory method calculates the available memory by determining the total size minus the used memory, and the tlb method outputs the contents of the TLB.