Contents

Introduction 2

Algorithm and Design 2

Issues 6

Compilation and Execution 7

Sample Output 8

Conclusion 12

Vagner Machado

Professor John Svadlenka

CSC 340

8 May. 2020

Project 4 – Linear Memory Allocation Management and Compaction

# Introduction

Aiming to strengthen the understanding of operating systems’ memory allocation and memory compaction algorithms, students were instructed to design and code an application using the C/C++ languages that simulates those functions. The given instructions detail that the memory to be allocated consists of a linear section of size 1MB. Further, the algorithm should use a best-fit strategy for memory allocation, meaning that processes should be inserted into memory, if enough is available, at a location that results in the lowest number of free bytes between neighboring processes. The following sections of this report contain information related to the development of this assignment. Amongst those are how I designed the algorithm and data structures needed to implement the project, reported issues faced while programming, provided compilation and execution instructions, and produced program output based on provided set of commands.

# Algorithm and Design

During the design step of the algorithm for the proposed memory allocator, I outlined additional classes and data structures need to support an organized and clean programming of the memory allocator application.

First, I recycled and modified the Process and LinkedList classes designed for project three. Nevertheless, those classes have the same fundamental characteristics as in the previous assignment, since the former and latter also describe instance data and methods needed to properly instantiate and manipulate Process and LinkedList objects. For this assignment, a Process has a new set of instance data: firstByte, lastByte and size to enable for tracking of memory locations occupied by a process and its size. This class also tracks the name (or number) of a process and provides a toString()method to access a print out of the instance data for the object. While the instance data for a LinkedList was reduced to only a head pointer to a Process, the LinkedList methods needed for this assignment were reasonably more complex to implemement. More specifically, the methods findSpotInList(args) and addProcessToList(args), called in this order, are responsible for finding a best fit location for a Process and inserting it into a possibly fragmented section of memory. It is best to see their documentation to fully grasp the vital role those methods play in the proper functionality of the application. Another noteworthy method for this class is compactProcesses(args),which iterates through the list of processes and shifts their location towards the low bytes in memory. As consequence, case the memory is not full, compaction causes free memory to be shifted towards the high bytes. The LinkedList class also allows for object removal through the method removeProcessByNumber(args), and provides a toString() method to access a printout of the current processes in memory.

After carefully customizing the Process and the LinkedList classes, I developed the Memory class that, as the name suggests, describes the instance data needed for the instantiation and manipulation of memory in the allocator application. Each Memory object contains only two variables for instance data, a LinkedList object to hold allocated Process objects, and an integer max to indicate the last byte available for the memory. For the starting point, all memory objects have the first available byte for allocation at position zero. Aiming to achieve some encapsulation between classes, the methods available for the Memory class are named similarly to the ones in the LinkedList. Those methods are called findMemorySlot(args), addProcessToMemory(args), removeProcessFromMemory(args), compactMemory() and a toString()method. A method called on a Memory object forwards the call to its instance of a LinkedList object, where all the computing gets done. This extra design feature was not necessary to accomplish this programming assignment, however, it adds some subtle abstraction and encapsulation between objects of different classes.

Lastly, I designed the class Allocator to manage the interaction with the user through its only public function, run(). This class has additional private functions to aid run() validating program arguments, reading and parsing user input, and printing usage information when necessary. The run()method contains the code developed to implement the following algorithm to handle user interaction with the memory allocator application:

1. Validate argument input by the user and print usage if needed.
2. Initiate allocator with valid arguments and initiate a loop.
3. While the allocator command is not QUIT:
   1. Collect and parse user input.
   2. Case first command is RQ and fourth is B
      1. Attempt to parse third argument as an integer for size. If it fails, warn user and reject the command.
      2. If parsed item is an integer, attempt to allocate that memory size. If memory is not available, warn user and reject the command. Else allocate memory for required size Process and set its location in memory.
   3. Case the first command is RL, attempt to release the process named on second word of command.
      1. Case the process name does not match a currently allocated process, user is warned and command is rejected.
      2. Case of a name match, the memory used by the process is set to FREE.
   4. Case the first command is STAT
      1. Print the status of memory, first and last bytes, process allocated in it, if any, or FREE for unallocated sections.
   5. Case the first command is HELP (added functionality).
      1. Prints usage information during runtime without halting program execution.
   6. Case the first argument is C
      1. Memory is compacted by shifting processes towards the low end of memory and free space towards the opposite end.
   7. Case the first command is QUIT.
      1. Allocator application is terminated.
   8. Else the command is invalid, user is warned, command rejected.
   9. Go to top of loop, item (a).

Careful design and implementation of supporting classes were crucial to achieve all functionalities provided by the Allocator::run() method.

# Issues

While not facing any major dead-end during design time or implementation, I would like to illustrate the algorithm for LinkedList::findSpotInList(args),the method that iterates through a possibly fragmented memory and finds the best-fit spot for a Process object to be placed. I am relating the algorithm under this section because it required careful thinking and edge case analysis during design, and lacking those skills could result in a real implementation issue. The algorithm for this method proceeds as follows.

1. Case list is empty and there is enough memory, place process at position 0.
2. Set position to -1; hop counter to 0; bestGap to INT\_MAX;
3. Traverse the list of processes starting at head.
4. Case the firstByte of head is not zero, there is a gap at the start.
   1. Measure the gap
      1. If the process fits, set bestGap to gap, position++.
   2. Set counter to 1
   3. While traverse pointer is not NULL.
      1. If next is NULL, check if last byte of process is the last one available for memory allocation.
         1. If it is not, measure gap between last process and last byte of memory.
         2. If process fits and remaining gap is less than bestGap, set bestGap to gap. Set position to counter.
      2. Else, somewhere in the middle of the list.
         1. Case lastByte of current process is not one less than the firstByte of next process, there is a gap.
         2. Measure gap. If process fits and remaining gap is less than bestGap, set bestGap to gap. Set position to counter.
      3. Increment counter, step to next process, go to top of loop.
      4. When iteration is done, return position.

This is, to my opinion, a well though out algorithm design that finds the best-fit position for a process in memory, leaving the smallest possible free memory between adjacent processes. The result generated by this method is used by addProcessToList(args)to hop forward position number of times before performing a Process insertion and updating its instance data. The intricacy involved in coding this method made it prone to cause issues, which is the reason for it being included in this section.

# Compilation and Execution

Compiling and running this application is very straightforward. Despite the project’s specification not requiring arguments other than an integer to represent memory bytes to be allocated, I decided that as a good practice, to support the -help optional argument to provide users with some information about the application and usage instructions.

To execute this application, the first argument must be either –help or an integer. Case the integer is lower than 1048576, or 1 megabyte, I opted to default the memory available to 1048576 bytes. Case the number passed as argument is higher than that, the memory will be extended. Either case, the memory available will range from 0 to argv[1] – 1. Below are the instruction for compilation and execution.

Compiles the source code in file allocator\_VMachado.cpp into object file: **g++ -c allocator\_VMachado.cpp**

Links the object file allocator\_VMachado.o to executable allocator\_VMachado.exe   
 **g++ -o allocator\_VMachado.exe allocator\_VMachado.o**

Runs the application with default memory size   
 **./allocator\_VMachado.exe 1048576**

Prints useful information about application usage to console and file:  
 **./allocator\_VMachado.exe -help**

Using the compilation and optional arguments during the execution, the application produces the set of outputs shown next.

# Runtime Commands and Sample Output

In this section, the available runtime commands and output produced by the provided set of commands is presented. When running the program, the user must provide either an integer argument for max memory size or run the application with argument -help to print useful information about the application usage. During runtime, the user can type line commands just as the following space-delimited, case-sensitive examples:

1. **RQ P3 1024 B**

Requests memory for process name P3 of size 1024 bytes using the best-fit strategy. Case there is not enough memory, the user is warned and command is rejected.

1. **RL P3**

Attempts to releases the memory allocated to process name P3. Case P3 is not a valid name for an allocated process, the user is warned and command is rejected.

1. **STAT**

Prints status of the memory section, start and final byte, and either process number or FREE for section(s) of not allocated memory.

1. **C**

Compacts the memory by shifting processes towards the low bytes in memory.

1. **QUIT**

Quits the program.

1. **HELP** (extra functionality)

Prints program usage during runtime and program does not quit execution.

Both the **HELP** command entered during runtime and -helppassed as argument during program execution cause the following message to be printed onto the system’s console:

*“ This project simulates a linear memory allocator application. The memory is initially set to have its first byte at location 0 and last available memory byte is represented by the integer passed as program argument during execution, minus 1. When the program loads, the user sees the prompt 'allocator >' to which the user can input commands like the following space separated or tab separated commands examples in a line:*

*a. RQ P3 1024 B Requests memory for P3 using best-fit strategy for 1024 bytes allocation. Case there is not enough memory, user is warned, and request is rejected.*

*b. RL P3 Releases the memory allocated to P3. Case P3 is not a valid name for allocated process, the user is warned and release   
 is rejected.*

*c. STAT Prints status of memory, start and final byte, and either process number or FREE   
 if memory is not allocated.*

*d. C Compacts the memory by shifting free space towards the higher bytes in   
 memory.*

*e. QUIT Quits the program.*

*f. HELP Prints program usage during runtime, program does not quit.*

*Note: All commands are case sensitive and preferably should not contain spaces in the beginning or end of input line for guaranteed execution. For this project, B strategy, best fit, is the only option and any other value passed other than B for the fourth item of RQ will cause allocation to be rejected. Multiple space and tab trimming for input line was implemented, but single space separation between entries in the line is still preferred for optimal execution.*

*Usage: Instruction on how to run the program:*

***g++ -c allocator\_VMachado.cpp*** *Compiles the source code in to object file*

***g++ -c VMapp.exe allocator\_VMachado.o*** *Links the object file to executable.*

***./VMapp.exe 1048576*** *Runs and allocates 1MB for the allocator app.*

***./VMapp.exe -help*** *Prints usage information*

*PS: 1MB is lowest value allowed and will get overwritten to 1048576. Any value over 1048576   
 will extend memory.*

*Memory slot range: [ 0 : argv[1] - 1 ] Where argv[1] is argument passed when executing the program and   
 which lowest value can be 1048576, representing 1MB.”*

Next is the output generated by the application using the provided set of commands as input.

* INPUT

RQ P0 262144 B

RQ P1 262144 B

RQ P2 262200 B

STAT

RL P1

RQ P3 200000 B

STAT

RQ P4 200000 B

STAT

C

STAT

QUIT

* OUTPUT

allocator> RQ P0 262144 B

allocator> RQ P1 262144 B

allocator> RQ P2 262200 B

allocator> STAT

Address [ 0 : 262143 ] Process P0

Address [ 262144 : 524287 ] Process P1

Address [ 524288 : 786487 ] Process P2

Address [ 786488 : 1048575 ] Free

allocator> RL P1

allocator> RQ P3 200000 B

allocator> STAT

Address [ 0 : 262143 ] Process P0

Address [ 262144 : 524287 ] Free

Address [ 524288 : 786487 ] Process P2

Address [ 786488 : 986487 ] Process P3

Address [ 986488 : 1048575 ] Free

allocator> RQ P4 200000 B

allocator> STAT

Address [ 0 : 262143 ] Process P0

Address [ 262144 : 462143 ] Process P4

Address [ 462144 : 524287 ] Free

Address [ 524288 : 786487 ] Process P2

Address [ 786488 : 986487 ] Process P3

Address [ 986488 : 1048575 ] Free

allocator> C

allocator> STAT

Address [ 0 : 262143 ] Process P0

Address [ 262144 : 462143 ] Process P4

Address [ 462144 : 724343 ] Process P2

Address [ 724344 : 924343 ] Process P3

Address [ 924344 : 1048575 ] Free

allocator> QUIT

Allocator terminated

# Conclusion

Students were instructed to design and implement a program that manages the allocation and release of a linear section of memory, and that reports its status when required by the user. To accomplish this task, I implemented additional classes and data structures to support the development of the proposed algorithm. Designing the algorithm to find the best position for a process in a fragmented section of memory proved to be an excellent coding skill development exercise and detail-oriented process. By following the detailed compilation and execution instructions, the user is able to interact with the program, request memory, release memory, ask for help, get a memory status printout, and compact the memory section. Implementing this project enabled me to closely analyze and better understand a system’s intricate memory allocation procedures.