CS420: Compiler Design

Fall, 2017

Term Project #3: Memory Management

(Due date: Dec. 17, 2017)

- **Overview**

  The goal of this project is to experience efficient memory management methods. Students should build own memory management methods to process the given memory allocation and deallocation schedules.

- **Memory Management Policies**

  In this assignment, three memory allocation and deallocation schedules are provided, and students should implement at least two of the methods to process the schedules. The methods should use determination algorithm for the allocation, defragmentation, etc.

  Reference: first fit, best fit, worst fit, learning based, etc.

- **Memory Characteristics**

  - Total memory size is 1Gb.

    The memory management processes various memory allocation and deallocation requests in the memory space.

  - Every allocation starts with a continuous space started from a specific address. No space should be separated into several discontinuous regions.

  - For defragmentation, the actual physical address of an occupied space

changes. You must consider this in this assignment.

- **Memory Allocation Schedules**

  Three schedules made by three different allocation and deallocation patterns are described in the below.

  - sch_random.c: Fully randomized pattern. Randomized sequence of allocation or deallocation is made 10000 times exhaustively utilizing 1Gb memory (up to 90% of total memory). Allocation block sizes are completely random in the range of 4bytes ~ 8Mb.
  - sch_greedy.c: Batch pattern. Allocations are called exhaustively utilizing 1Gb memory(up to 90% of total memory). Then, deallocation calls keep occurring until deallocates 90% of total allocation. And then allocation calls start occur. This cycle repeats up to 10000 calls. Allocation block sizes are 4bytes, 8bytes, 16bytes … 2048bytes, 4096bytes or completely random in range of 4Kb ~ 8Mb.
  - sch_backnforth.c: Alternative pattern. Allocation and deallocation are called alternatively (ex. 3 allocations, 1 deallocation, 2 allocations, 1 deallocation …). Ratio of allocations and deallocation changes in range of 0:1 ~ 4:1 depend on total utility of the memory space. The less utility, the more allocation calls. This repeats up to 10000 calls. Allocation block sizes are 4bytes, 8bytes, 16bytes … 2048bytes, 4096bytes or completely random in range of 4Kb ~ 8Mb.

  And there are 2 kinds of allocation or deallocation function in the schedules.

  malloc(size_t size)
  free(void* ptr)

- **Unit Actions**

  Three unit actions are given, and each unit action has its own cost. The overall performance of each policy is measured by below equation:

$$total\ cost = \sum_{i} (The\ number\ of\ calls\ of\ \boldsymbol{Action\ i}) \times (The\ cost\ of\ \boldsymbol{Action\ i})$$

  Student should implement at least 2 memory management policies using specified unit actions. Table below provides the specification of the unit actions.

Unit Action 1: **allocate**(*allocAddr*, *allocMemSize*);
Input: *allocAddr*: a starting address to allocate contiguous memory. *allocMemSize:* the size of allocated memory in byte unit. **Description:** This action allocates *allocMemSize* bytes of memory at *allocAddr*. For example, **allocate**(0x000000FE, 4) is called, 4 bytes located at 0x000000FE, 0x000000FF, 0x00000100, and 0x00000101 are allocated. **Cost:** The cost of each **allocate** call is 10 for every 32 bytes allocation. ex) cost of **allocate**(0x000000FF, 16) = 10 cost of **allocate**(0x000000FF, 32) = 10 cost of **allocate**(0x000000FF, 64) = 20 cost of **allocate**(0x000000FF, 65) = 30
Unit Action 2: **deallocate**(*deallocAddr*);
Input: *deallocAddr*: a starting address to allocate contiguous memory. **Description:**

This action deallocates the allocated size of memory at *deallocAddr*. For example, **deallocate**(0x000000FE) is called, if the address was allocated with 4 bytes long, the space located at 0x000000FE, 0x000000FF, 0x00000100, and 0x00000101 are deallocated.

Cost:

The cost of **deallocate** call is 5 for every 32 bytes deallocation.

Unit Action 3: **migrate**(*srcAddr*, dstAddr, *migrateMemSize*);

Input:

srcAddr: a starting address of source memory moved to another place.

dstAddr: a starting address of destination memory moved from another place.

Description:

This action migrates the allocated size of memory from *srcAddr* to dstAddr. For example, **migrate**(0x000000FE, 0x00000010) is called, if the address was allocated with 4 bytes long, 4 bytes located at 0x000000FE, 0x000000FF, 0x00000100, and 0x00000101 are moved to 0x00000010, 0x00000011, 0x00000012, and 0x00000013.

Caution! This unit action include both of **allocate**() and **deallocate**(); operations. Allocation and deallocation occur automatically according to the source address and destination address when you call this function.

Cost:

The cost of **migrate** call is 10 for every 32 bytes deallocation.

And the correspondent costs for accompanied allocation and deallocation will be added.

- **What you have to do**

  - Design policies

    You should design at least 2 policies except first fit(provided as an example), last fit(the same as first fit) and random fit(too easy) for memory management. Remember, performance of each policies mainly depends on how much cost taken for defragmentation, in this homework. You should establish your policies to make as less as possible defragmentation occurs. The design concept and their operations should be described in report.

  - Implementation of the policies

    Reading schedule, unit actions and cost calculating parts are provided as pre-implemented. What you have to do is just filling out implementations of malloc, free functions using unit actions in each policy.

    The defragmentation functions are also pre-implemented in the FirstFit.cpp, you can use them with just proto-typing as extern function. Or you can improve the function to achieve the better performance. If you implemented your own defragmentation functions, please mention it on the report.

    Base source code is given in C++, so we recommend you to use C++. If you want to use other languages, you may. But implementing all of the operation would be a tough work, so we don't recommend.

  - Evaluation

    You should sum up the total cost performing each schedule by policies.

	First Fit	Policy 1	Policy 2	...
Random	*cost*	*cost*	*cost*	...
Greedy	*cost*	*cost*	*cost*	...
Back & Forth	*cost*	*cost*	*cost*	...

    Then you should analyze the results and evaluate each performance of policies on each schedule. There should be some descriptions on each case. (Regarding the characteristics of the policy and the pattern of the schedule, why the result shows like that?)

- **Internal structure document re-submission**

Actually, this submission is not related on the current tasks. You have to include the internal structure document again in this submission. If there are not any revisions in implemented internal structure from the one of you submitted before, it is acceptable just submitting that again. Otherwise, please revise the document according to your real implementation and submit.

- **Submit form**

A PDF file report (<5 pages)
Source codes on implementation of your policies (YourPolicy1.cpp, YourPolicy2.cpp)
A PDF file revised document for internal structure (<10 pages)
Compress files above into a zip file

HW4_StudentID_Name.zip
ex) HW4_20173283_Kyuho_Son.zip

If plagiarism is detected, zero-score will be given.
TA will check the details of source code of each student.
If any problem or question, feel free to ask TA with E-mail or face-to-face at office hour in every Friday.

TA (Kyuho Son) : ableman@kaist.ac.kr