CS420: Compiler Design

Fall, 2018

Homework #4: Memory Management

(Due date: Dec. 16, 2018)

- **Overview**

  The term project submission is focusing on efficient memory management. Students should build efficient memory management policies to cover the given memory allocation and deallocation schedules.

- **Memory Management Policies**

  In this assignment, 3 memory allocation and deallocation schedules are provided, and students should establish at least 2 of your own memory management policies to handle the schedules. The policy should contain a determination algorithm on the position of each allocation, defragmentation activity and so on.

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

- **Memory Characteristics**

  - Total memory capacity is 1Gb.
    You should perform each whole schedule with the memory.
  - Every allocation should have a continuous space starting with a distinct address. No space should be separated into several discontinuous regions.
  - For defragmentation, the actual physical address of an occupied space

changes. This matters in a real OS handling addresses of pointers, but never mind on the address change for more simple simulation.

- **Memory Allocation Schedules**

Below 3 schedules made by 3 different allocation and deallocation patterns are provided.

- sch_random.c: Fully randomized pattern. The randomized sequence of allocation or deallocation call occurs 2000 times almost exhaustively utilizing 1Gb memory(up to 90% of total memory). Allocation block sizes are completely random in range of 4bytes ~ 16Mb.
- sch_greedy.c: Batch pattern. Allocation calls keep occurring almost exhaustively utilizing 1Gb memory(up to 90% of total memory). Then, deallocation calls keep occurring until deallocates 90% of the total allocation. And then allocation calls start to occur. This cycle repeats up to 2000 calls. Allocation block sizes are 4bytes, 8bytes, 16bytes … 2048bytes, 4096bytes or completely random in range of 4Kb ~ 16Mb.
- sch_backnforth.c: Alternative pattern. Allocation and deallocation calls occur alternatively (ex. 3 allocations, 1 deallocation, 2 allocations, 1 deallocation …). The ratio of allocations and deallocation changes in the range of 1:1 ~ 5:1 depends on the total utility of the memory space. The less utility, the more allocation calls. Allocation block sizes are 4bytes, 8bytes, 16bytes … 2048bytes, 4096bytes or completely random in range of 4Kb ~ 16Mb.

And there are 3 kinds of allocation or deallocation function in the schedules.

```
malloc(size_t size)
realloc(void* ptr, 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:

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

  The student should implement at least 2 memory management policies using specified unit actions. The 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 memories 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*, *deallocMemSize*);
Input: *deallocAddr*: a starting address to allocate contiguous memory. *deallocMemSize:* the size of deallocated memory in byte unit.

Description:

This action deallocates *deallocMemSize* bytes of memories at *deallocAddr*. For example, **deallocate**(0x000000FE, 4) is called, 4 bytes 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.

migrateMemSize: the size of migrated memory in byte unit.

Description:

This action migrates *migrateMemSize* bytes of memories from *srcAddr* to dstAddr. For example, **migrate**(0x000000FE, 0x00000010, 4) is called, 4 bytes located at 0x000000FE, 0x000000FF, 0x00000100, and 0x00000101 are moved to 0x00000010, 0x00000011, 0x00000012, and 0x00000013.

Caution! This unit action does not include neither **allocate**() nor **deallocate**(); operations. The destination memory space must be allocated before calling **migrate**(). Similarly, the source memory space must be deallocated independently if required.

Cost:

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

- **What you have to do**

  - Designing policies

    You should design at least 2 policies except for first fit(provided as an example), last fit(the same as first fit) and random fit(too easy) for memory management. Remember, the performance of each policy 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 the 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, realloc, free functions using unit actions in each policy.

    The 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 operations would be a tough work, so we don't recommend.

  - Evaluation

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

	Policy 1	**Policy 2**	**...**
Random	*cost*	*cost*	...
Greedy	*cost*	*cost*	...
Back & Forth	*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?)

- **Some hints about the base source code**
  - ✓ What you have to compile : main.cpp (main function), MemoryStructure.cpp, ScheduleProcessor.cpp, FirstFit.cpp, YourPolicy1.cpp, YourPolicy2.cpp
  - ✓ Header files : define.hpp, MemoryStructure.hpp, ScheduleProcessor.hpp, Policy.hpp, FirstFit.hpp, YourPolicy1.hpp, YourPolicy2.hpp
  - ✓ You don't have to compile sch_random.c, sch_greedy.c, sch_backnforth.c. These are allocation / deallocation schdule files.
  - ✓ Make full use of defragmentPrepareEnd and defragmentPrepareFront functions already implemented in FirstFit.cpp. You would be able to implement your schedules just reordering some statements in onMalloc and onFree functions in the FirstFit.cpp example code.

- **Submit form**

A PDF file report (<5 pages)
Source codes on the implementation of your policies
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 the source code of each student.
If any problem or question, feel free to ask TA with E-mail.

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