CS420 Assignment#4 - Memory Allocation

20173245 Chansu Park

Before we enter the main part, notice that I migrated the declaration of defragmentPrepareEnd and defragmentPrepareFront provided in the FirstFit.cpp to the FirstFit.hpp and included “FirstFit.hpp” to the YourPolicy1.cpp and YourPolicy2.cpp to use that methods without reproducing them.

1. Selecting the memory allocation methods

I selected to implement Best Fit allocation and Worst Fit allocation introduced in the description of the Assignment#4.

2. Best Fit Allocation

Best Fit Allocation tries to find the smallest possible free block from the allocation list (ms->getAllocation(i) for each i).

I originally tried to record each possible candidates and select the best one by traversing the candidate list once more.

typedef struct BestCand{ void* prev_end; int size;} BestCand; // records each block position

vector<BestCand> candList; // push_back into candList for each possible block

However, it constantly failed and tried to use slow fragmentation always with the same size of the block. I also tried to change the type of prev_end to int and cast each address to int when save BestCand object, but it still make an unexpected result. Thus I commented out the candidate list method. (You can see the mess on YourPolicy1.cpp)

After that, I changed to just maintain the best address and the size in the method OnMalloc(.).

```
int cand_size = (int)alloc.addr - (int)prev_end;
if(cand_size >= size && min_size > cand_size)
{
  min_size = cand_size;
  best_end = prev_end;
}
```

At the end of the list, which contains the remaining space, it also compares the last block’s size manually.

```
    if(MAX_MEMORY_CAP - (int)prev_end >= size && min_size >
MAX_MEMORY_CAP - (int)prev_end)
    {
      min_size = MAX_MEMORY_CAP - (int)prev_end;
      best_end = prev_end;
    }
```

Then it allocates at the best_end position and return that position.
This method doesn't require the reallocation, so I didn't implemented that one. Also, since free just makes the used block into the available list, I just brought the OnFree(.) function from the FitstFit.cpp into YourPolicy.cpp. Defragmentation also works as same as First Fit at the end of the infinite loop of OnMalloc.

3. Worst Fit Allocation

Worst Fit works similarly. It just tries to find the largest possible block from the allocation list.
(ms->getAllocation(i) for each i).

```
int cand_size = (int)alloc.addr - (int)prev_end;
if(cand_size >= size && max_size < cand_size)
{
  max_size = cand_size;
  worst_end = prev_end;
}
```

```
    if(MAX_MEMORY_CAP - (int)prev_end >= size && max_size <
MAX_MEMORY_CAP - (int)prev_end)
    {
      max_size = MAX_MEMORY_CAP - (int)prev_end;
      worst_end = prev_end;
    }
```

Worst fit also doesn't require the reallocation. OnFree(.) works as same as other policies. Defragmentation also works as same as First Fit at the end of the infinite loop of OnMalloc(.).

4. Evaluation

	First Fit	YourPolicy1(Best)	YourPolicy2(Worst)
Random	23306660180	11962483435	44215755215
Greedy	2599053970	2599053970	4132115030
Back & Forth	31164454230	16913616675	41100209085

This is the table of the cost. For the performance, Best fit performed better than the first fit, and the first fit performed much better than the worst fit. (In other words, the number of the occurence of the fast defragmentation gets much bigger when moving from the best fit to the worst fit.)

a. Random Schedule

Random Schedule tries to make many fragmentations randomly. Unlike other schedules, random schedule doesn't allocate the regular-sized small pieces (4B~4KB), so it can make more irregularly-shaped small pieces than other schedules. Thus the first fit will reduce the size of many medium-sized remaining blocks, which causes some migration for that medium-sized blocks. Whereas, the best fit will make very tiny fragments continuously, so it wouldn't harm the big-pieces. But, the worst fit will make fragmentation both for the small blocks and the large blocks, so it naturally performs as the worst one.

b. Greedy Schedule

Greedy Schedule tries to fill the almost all memory space consequently from the start address to the end address, and tries to free almost all memory space.

For the first fit and the best fit, fast defragmentation didn't occur.

For the best fit, since the batch allocation occurs, after the batch deallocation, we can still find the best fit space for the large blocks (4KB ~ 16MB) without migrating the address. For the small pieces, since the size is regular (4, 4*2, 4*2*2, …, 4*2^10 Bytes), we can always find the best fit after deallocation on such small pieces.

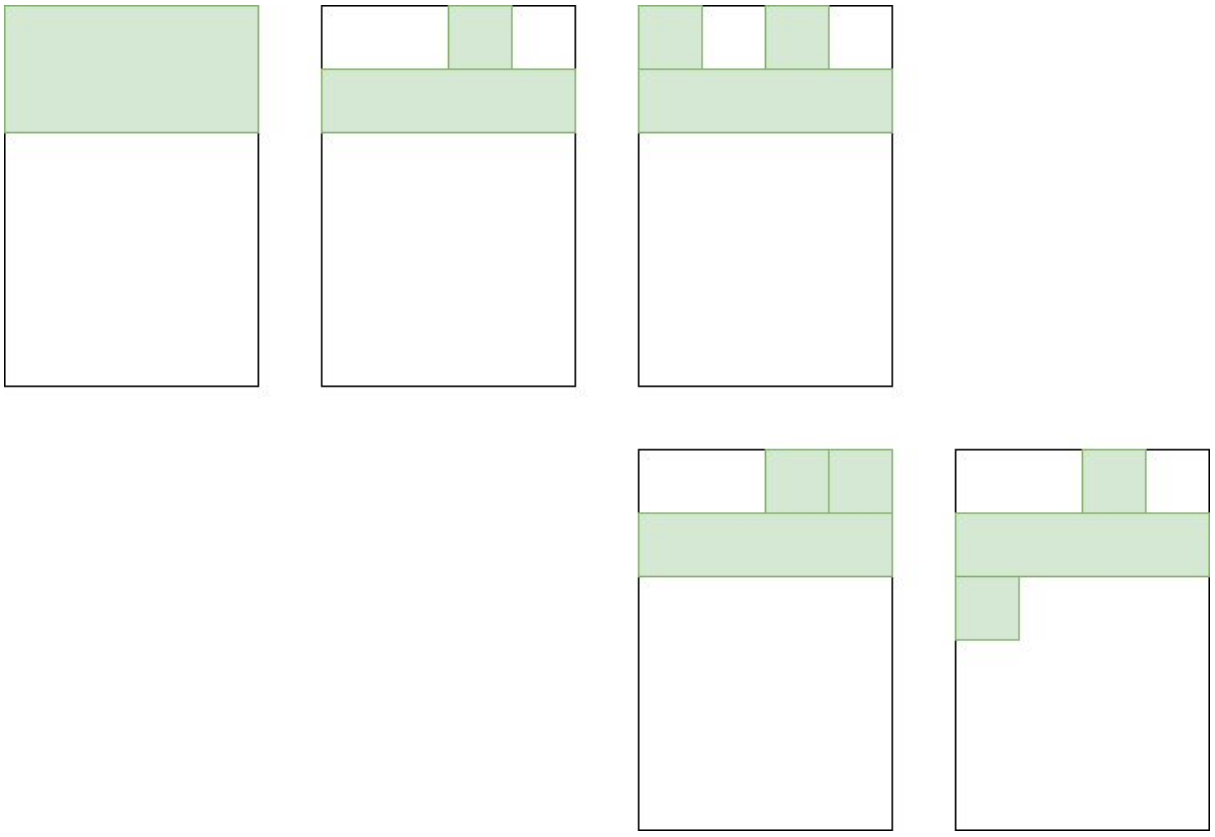
This phenomena occurs similarly for the first fit. For the small pieces, when a small piece find the first fit space which is much larger than the slice itself, it won't make other relatively big pieces to fail to find the remaining space, which makes no migration, since the piece will regularly cut the piece by the power of 2. For the large blocks, especially near the ending address, it can always find the fit before it harms the largest space because the former space were already cleaned by the batch deallocation.

For the worst fit, however, some smaller pieces can harm the larger allocations. After batch deallocation, there will be many pieces of small space and some larger pieces of large space. When we continue the batch allocation, however, those small blocks will just cut off the remaining large blocks, which can make large blocks, which were supposed to be put into that spaces, to be migrated to other space or even make some slow defragmentations.

c. Back & Forth Schedule

Back & Forth Schedule tries to deallocate some formerly-occupied pieces and continue the consumption of remaining spaces. It is a little bit regular than the Random Schedule.

After some deallocation near the start address occurs, first fit policy can make relatively small pieces to cut off the relatively large space near the start position, which will make the fragmentation more frequently than the best fit. Best fit tends to consume the smaller deallocated space for the allocation, so it will make the less defragmentation. For the worst fit, however, even after some initial deallocation, it will just use the rightmost remaining space and doesn't try to utilize some deallocated space at the front of the address, which will make much more failure to allocate large blocks.



This diagram shows the example for all the three schedules. When we fill the Block with size (1)2/(2)1/(3)1/(4)4, suppose that deallocation occured for (1), (3), and now we want to try to put the size 1 block. Then the first fit (third diagram) uses (1) and make the remaining space as 1, which might be uncomfortable for the next blocks with size 2, but still it is okay for the greedy schedule since this space still can be used for the small-sized block like size 1 block. Best fit uses(3) so it wouldn't harm the space for the block size 2. Worst fit, on the other hand, will just use the remaining space.

5. How to execute

In the source folder, I placed the makefile to compile the entire project. The content of this follows:

```
all: main.cpp MemoryStructure.cpp ScheduleProcessor.cpp FirstFit.cpp
YourPolicy1.cpp YourPolicy2.cpp
  g++ --std=c++11 -m32 -g main.cpp MemoryStructure.cpp ScheduleProcessor.
cpp FirstFit.cpp YourPolicy1.cpp YourPolicy2.cpp -o test
```

It works for Ubuntu 18.04 LTS. To compile this, you might need to install

gcc-multilib/g++-multilib.

This report is in the root folder. I hope you to execute this without any problem.