CPSC 457 - Assignment 5

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 23% of the final grade.

Q1 – Programming question (70 marks)

You will write a **best-fit dynamic partition** memory allocation simulator that simulates some of the functionality performed by `malloc()` and `free()` in the standard C library. The input to your simulator will be a page size (a positive integer) and list of allocation and deallocation requests. Your simulator will process all requests and then display some statistics.

Throughout the simulation your program will maintain an ordered list of dynamic partitions. Some partitions will be marked as occupied, the rest marked as free. Each partition will also contain its size in bytes. Further, occupied partitions will have a numeric tag attached to it. Your simulator will manipulate this list of partitions as a result of processing requests. Allocation requests will be processed by finding the most appropriately sized partition and then allocating a memory from it. Deallocation requests will free up any relevant occupied partitions, and also merging any adjacent free partitions.

Start by downloading and compiling the skeleton code:

```
$ git pull https://gitlab.com/cpsc457/public/memsim.git
$ cd memsim
$ make
```

The only file you should modify and submit for grading is `memsim.cpp`. You need to implement your simulator in the function:

```
void mem_sim( int64_t page_size,
              const std::vector<Request> & requests,
              MemSimResult & result);
```

The parameter `page_size` will denote the page size and `requests` will contain a list of all requests to process. The requests are described using the `Request` class:

```
struct Request { int tag; int size; };
```

If `tag>=0`, then this is an allocation request, and the `size` field will then denote the size of the request. If `tag<0` then this is a deallocation request, in which case the `size` field is not used. You will report the results of the simulation via the `result` parameter.

Allocation request

Each allocation request will have two parameters – a tag and a size. Your program will use **best-fit algorithm** to find a free partition, by scanning the list of partitions from the start until the end of the list. If more than one partition qualifies, it will pick the first partition it finds. If the partition is bigger than the requested size, the partition will be split in two – an occupied partition and a free partition. The tag specified with the allocation request will be stored in the occupied partition.

The simulation will start with an empty list of partitions, or, if you prefer, a list containing one free partition of size 0 bytes. When the simulator fails to find a suitably large free partition, it will simulate asking the OS for more memory. The amount of memory that can be requested from OS must be a multiple of `page_size`. The newly obtained memory will be appended at the end of your list of partitions, and if appropriate, merged with the last free partition. Your program must figure out what is the minimum number of pages that it needs to request in order to satisfy the current request.

Deallocation request

A deallocation request will have a single parameter – a tag. In the input list of requests, this will be denoted by a negative number, which you convert to a tag by using its absolute value.

Your simulator will find all allocated partitions with the given tag and mark them free. Any adjacent free partitions will be merged. If there are no partitions with the given tag, your simulator will ignore such deallocation request.

Command line arguments:

The provided skeleton program will accept a single command line argument representing the page size in bytes.

Input:

The provided skeleton will read allocation requests from standard input, until EOF. Lines containing only white spaces will be skipped. Each non-empty line will represent one request, either allocation or deallocation.

Any line with two integers will represent an allocation request. The first integer will represent the tag of the request, and the second one will represent the size of the allocation request in bytes. For example, the line `"3 100"` represents an allocation request for 100 bytes with tag 3.

A line with a single negative integer will represent a deallocation request. The absolute value of the integer will represent the tag to be deallocated. For example, the line `"-3"` will represent a deallocation request for all partitions marked with tag 3.

Output:

At the end of the simulation your simulator must return two numbers via the `result` parameter:

- Set `result.n_pages_requested` to the total number of pages requested during the simulation. Notice that this could be `0`, but only if there are no allocation requests in the input.
- Set `result.max_free_partition_size` to the size of the largest free partition at the end of the simulation. You will set this to `0` if there are no free partitions.

Limits

- `requests.size()` will be in range [0 .. 1,000,000]
- `page_size` will be in range [1 .. 1,000,000]
- `Request::tag` will be in range [-1000 .. 1000]
- `Request::size` will be in range [1..10,000,000]

Make sure your code is efficient enough to process any valid input under 10s. The D2L page contains some hints on the data structures you can use to achieve this.

Overall algorithm

Pseudocode for allocation request:

- *search through the list of partitions from start to end, and find the smallest partition that fits requested size*
  - *in case of ties, pick the first partition found*
- *if no suitable partition found:*
  - *get minimum number of pages from OS*
  - *add the new memory at the end of partition list*
  - *the last partition will be the best partition*
- *split the best partition in two*
  - *mark the first partition occupied, and store the tag in it*
  - *mark the second partition free*

Pseudocode for deallocation request:

- *for every partition*
  - *if partition is occupied and has a matching tag:*
    - *mark the partition free*
    - *merge any adjacent free partitions*

Sample input / output

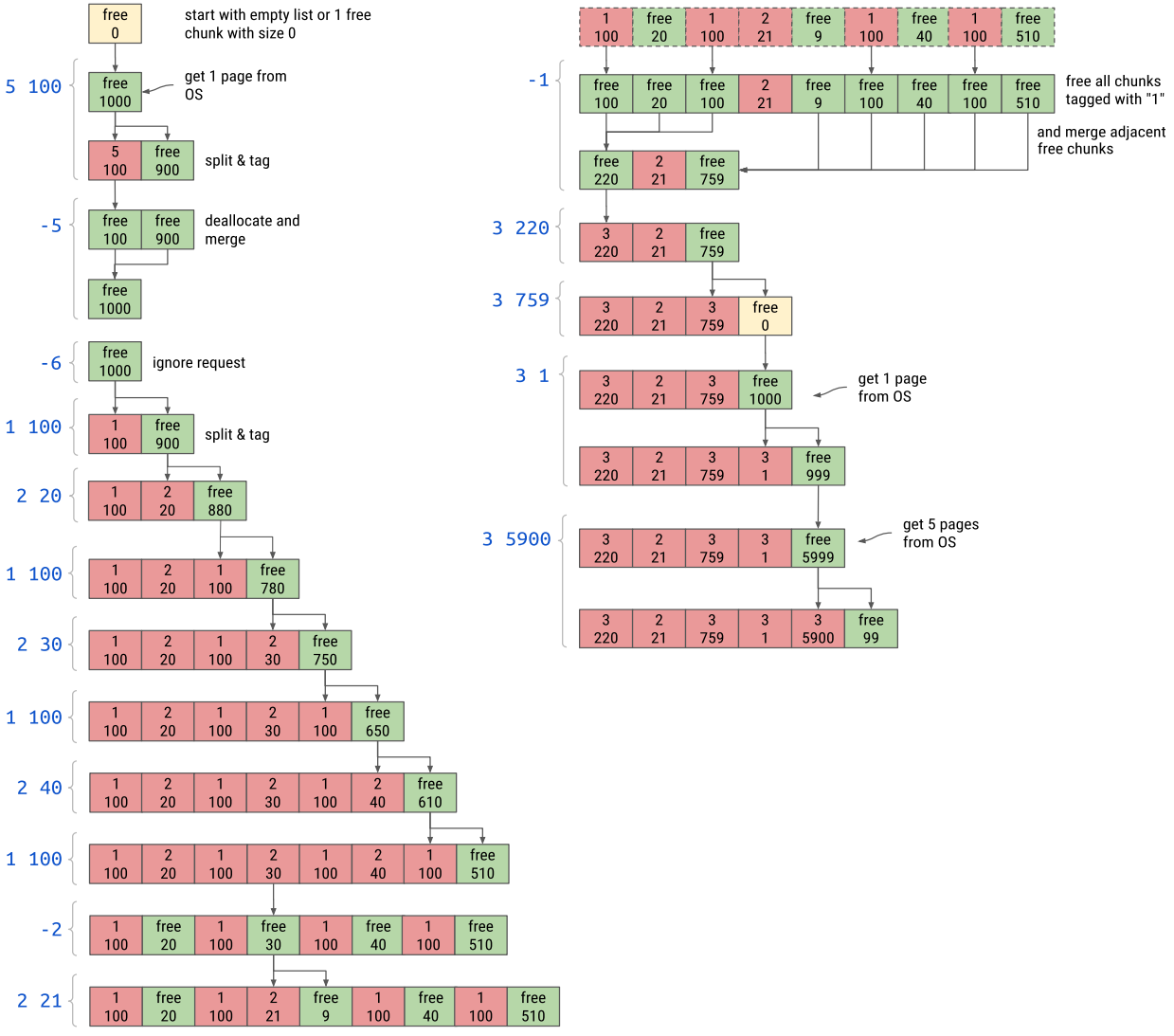
The GitLab repository contains a number of test files and correct results for some test runs. Here is a simple test run:

```
$ cat test1.txt
5 100
-5
-6
1 100
2 20
1 100
2 30
1 100
```

```
2 40
1 100
-2
2 21
-1
3 220
3 759
3 1
3 5900
```

```
$ g++ -O2 -Wall memsim.cpp -o memsim
$ ./memsim 1000 < test1.txt
pages requested: 7
largest free partition: 99
$ ./memsim 1 < test1.txt
pages requested: 6901
largest free partition: 0
$ ./memsim 33 < test1.txt
pages requested: 210
largest free partition: 29
```

The illustrations below show how the simulator processes these requests when `page_size=1000`.


free 0
start with empty list or 1 free chunk with size 0
5 100
free 1000
get 1 page from OS
5 100
free 900
split & tag
-5
free 100
free 900
deallocate and merge
free 1000
-6
free 1000
ignore request
1 100
1 100
free 900
split & tag
2 20
1 100
2 20
free 880
1 100
1 100
2 20
1 100
free 780
2 30
1 100
2 20
1 100
2 30
free 750
1 100
1 100
2 20
1 100
2 30
1 100
free 650
2 40
1 100
2 20
1 100
2 30
1 100
2 40
free 610
1 100
1 100
2 20
1 100
2 30
1 100
2 40
1 100
free 510
-2
1 100
free 20
1 100
free 30
1 100
free 40
1 100
free 510
2 21
1 100
free 20
1 100
2 21
free 9
1 100
free 40
1 100
free 510
-1
free 100
free 20
free 100
2 21
free 9
free 100
free 40
free 100
free 510
free all chunks tagged with "1"
and merge adjacent free chunks
free 220
2 21
free 759
3 220
3 220
2 21
free 759
3 759
3 220
2 21
3 759
free 0
3 1
3 220
2 21
3 759
free 1000
get 1 page from OS
3 220
2 21
3 759
3 1
free 999
3 5900
3 220
2 21
3 759
3 1
free 5999
get 5 pages from OS
3 220
2 21
3 759
3 1
3 5900
free 99

Q2 – FAT simulation (30 marks)

Write a function `fat_sim()` that examines the contents of a FAT table given to it in `fat[]`:

```
void fat_sim(const std::vector<long> & fat,
             long & longest_file_blocks,
             long & unused_blocks);
```

and calculates the following information:

- `longest_file_blocks` – number of blocks in the largest possible file (length of the longest terminated chain of blocks), and if no files are possible, display "0".
- `unused_blocks` – number of blocks that could not belong to any files (because they are not part of a terminated path)

Start by downloading and compiling a skeleton code:

```
$ git pull https://gitlab.com/cpsc457/public/fatsim.git
$ cd fatsim
$ make
```

Only modify and submit the file `fatsim.cpp` by implementing the `fat_sim()` function. Do not modify any other files.

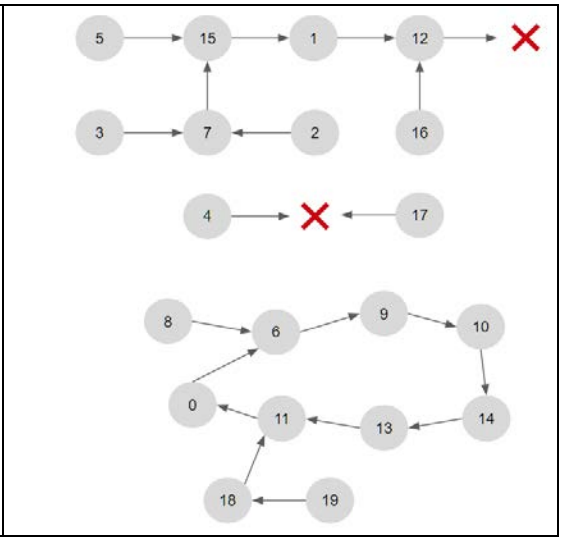
Input:

The FAT entries will be represented by N integers. Each number will be an integer in range [-1 .. N-1], where "-1" represents null pointer (or end of chain), and numbers ≥ 0 represent pointers to blocks. The i-th integer will represent the i-th entry in the FAT.

The skeleton code will read the FAT contents from standard input, parse them and supply them to your function as `std::vector`. Below is a sample test file and expected output.

```
$ cat test1.txt
  6 12  7  7 -1 15  9 15  6 10
 14  0 -1 11 13  1 12 -1 11 18

$ ./fatsim < test1.txt
blocks in largest file: 5
blocks not in any file: 10
elapsed time:           0.000
```



For example, the first integer '6' represents the fact that block 0 is linked to block 6. The graph on the right illustrates how the blocks are linked. There are 3 files possible in the above FAT (since it contains

'-1' three times). The largest file could start either on block 3 or on block 2, but in both cases, it would have a length of 5 blocks. The other two files would start on blocks 4 and 17, respectively, and both would be 1 block long. Notice that no files could start on any of the 10 nodes in the bottom half of the graph, as none of them are part of a terminated chain.

Limits:

- number of entries in FAT will be in the range [1 .. 1,000,000]
- make sure your code is efficient enough to process any valid input under 10s.

Submission

Submit 2 files to D2L for this assignment:

`memsim.cpp`	solution to Q1
`fatsim.cpp`	solution to Q2

General information about all assignments:

1. All assignments are due on the date listed on D2L. Late submissions will be not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
5. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
6. All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
7. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
8. **Assignments must reflect individual work**. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
9. Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
10. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix - hints for Q1:

If you use only basic data structures, such as dynamic arrays or linked lists, you will likely end up with an $O(n^2)$ algorithm. This is going to make your program too slow for many large inputs. To get full marks, you will need to use smarter data structures. I personally used:

- `std::list` - a linked list to maintain all partitions (to make splitting and merging of blocks constant time operation),
- `std::set` - a balanced binary tree to keep track of free blocks, sorted by size
    - the data I store here are pointers to the linked list nodes
    - the tree is sorted by size (primary key), and starting address of partition (secondary key), to make sure I pick the 'first' suitable partition
- `std::unordered_map` - hash table of lists to store all partitions belonging to the same tag
    - the data I store here are just pointers to the linked list nodes

This allows me to process every request in log(n) worst case time. Here are some relevant parts of my code:

```
struct Partition {
  long tag ;
  long size ;
  int64_t addr ;
};

typedef std::list<Partition>::iterator PartitionRef;

struct scmp {
  bool operator()( const PartitionRef & c1, const PartitionRef & c2) const {
    if( c1-> size == c2-> size)
      return c1-> addr < c2-> addr;
    else
      return c1-> size < c2-> size;
  }
};

struct Simulator {
  // all partitions (so that I can get std::prev() and std::next() nodes
  std::list<Partition> partitions;
  // sorted partitions by size/address
  std::set<PartitionRef,scmp> free_blocks;
  // quick access to all tagged partitions
  std::unordered_map<long, std::vector<PartitionRef>> tagged_blocks;

 ...
}
```

You will probably also want to use:

http://www.cplusplus.com/reference/set/set/lower_bound/

to lookup the best match. The easiest way I found to use it was to create a dummy linked list with just one element so that I can get an iterator out of it...:

```
std::list<Partition> dummy { Partition(-1,request.size,0) };
auto sbesti = free_blocks.lower_bound( dummy.begin());
PartitionRef best_free_block_iter = partitions.end();
if( sbesti != free_blocks.end())
   best_free_block_iter = * sbesti;
if( best_free_block_iter == partitions.end())
   ... need to allocate more memory
```

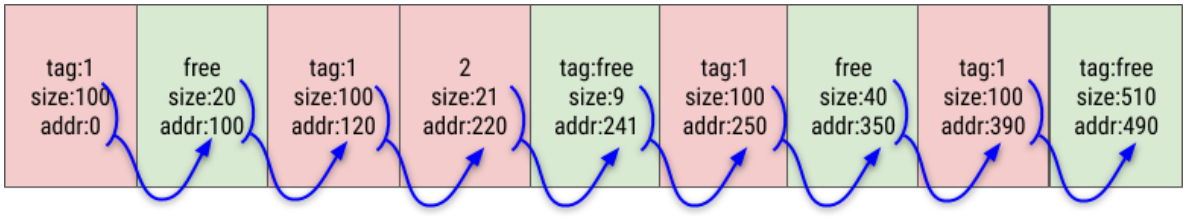
Partition address

The `partition.addr` represents the starting address of the block of memory that the partition represents. The first partition should have `addr=0`.

If a partition is in a linked list, eg. in a `std::list<Partition>`, and if `cptr` is an iterator to this partition then you can calculate its address as:

```
cptr-> addr = std::prev(cptr)-> addr + std::prev(cptr)-> size;
```

In other words, the address of the partition should be previous partition's address plus the previous partition's size. Another way to think about a partition address is that it is the sum of sizes of all partitions preceding it.



The reason you may want to keep track of partition addresses is because of the requirement that the best fit should choose the first partition in the list in case of ties. That means you need to sort your balanced binary tree by size, but in case of ties you sort it by address (secondary key).

Appendix - hints for Q2

No advanced data structures are needed. You may want to recall how depth first search algorithm works.