CPSC 457 - Assignment 5

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

There are 2 programing questions. Both will be marked based on correctness and efficiency. Your mark will be based on the number of tests your solution will pass. Incomplete code, or code that crashes will receive no marks. Small number of test inputs are provided for you, but you should create your own test inputs as well.

Q1 – Worst-fit dynamic partition simulator (70 marks)

You will write a **worst-fit** dynamic partition memory allocation simulator that approximates some of the functionality of `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 simulate processing all requests and then compute some statistics.

Throughout the simulation your program will maintain an ordered list of dynamic partitions. Some partitions will be marked as occupied, the rest will be marked as free. Occupied partitions will have a numeric tag attached to it. Each partition will also contain its size in bytes, and the starting address. The starting address of the first partition should be 0. 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 clone https://gitlab.com/cpsc457/public/memsim-w21.git
$ cd memsim-w21
$ make
```

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

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

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; };
```

When `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 **worst-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.

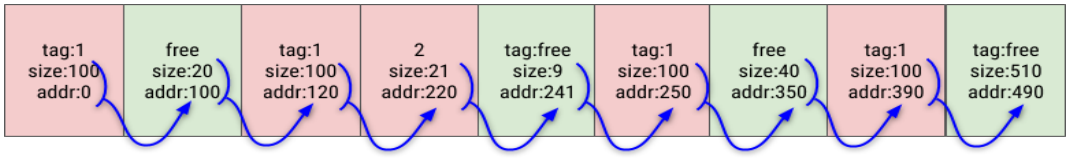
Your simulator will find all allocated partitions with the given tag and mark them free. Any adjacent free partitions will be merged. If there are no partitions with the given tag, your simulator will ignore such deallocation request.

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 driver program (main.cpp)

The included driver will accept a single command line argument representing the page size in bytes.

The driver 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 the `MemSimResult` structure with:

- `n_pages_requested` equal 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.
- Set `result.max_free_partition_address` to the address of the largest free partition at the end of the simulation. You will set this to `0` if there are no free partitions. In case of ties, report the smallest address.

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 [-10,000,000 .. 10,000,000]
- `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 appendix has 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 largest 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, but consider the
      case when last partition is free
    - add the new memory at the end of partition list
    - the last partition will be the best partition
- split the best partition in two if necessary
    - 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

```
$ 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
```

```
$ ./memsim 1000 < test1.txt
pages requested:                8
largest free partition size:    829
largest free partition address: 7171
$ ./memsim 1 < test1.txt
pages requested:                7030
largest free partition size:    129
largest free partition address: 221
$ ./memsim 33 < test1.txt
pages requested:                214
largest free partition size:    129
largest free partition address: 221
```

Additional test files are provided in the GitLab repository. Also, see appendix on how you can obtain correct results on any test files.

Q2 – FAT simulation (30 marks)

Write a function `fat_check()` that examines the contents of a file allocation table `fat[]`:

```
std::vector<long> fat_check(const std::vector<long> & fat);
```

The ends of block chains in the FAT will be represented by '-1'. Your function should find the longest possible chain terminating on each block containing '-1' in the FAT, and report these lengths. Start by downloading and compiling a skeleton code:

```
$ git clone https://gitlab.com/cpsc457/public/fatsim-w21.git
$ cd fatsim-w21
$ make
```

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

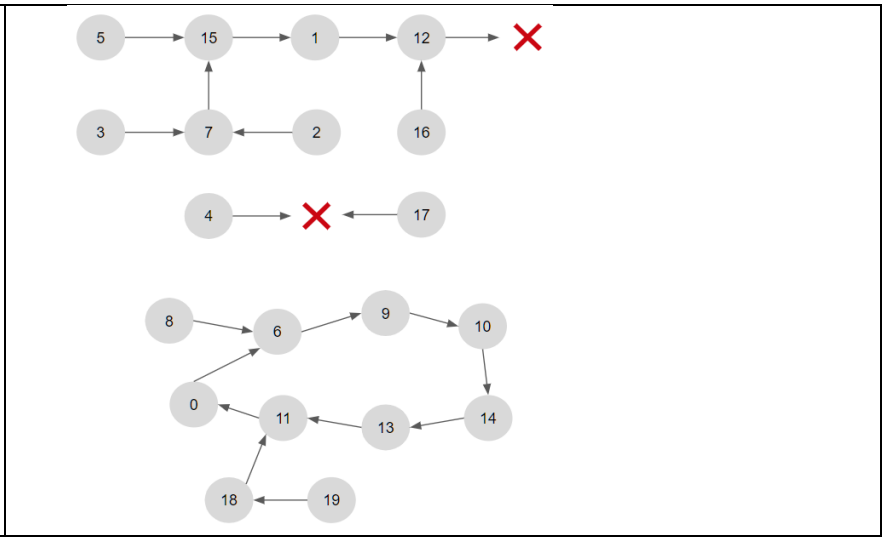
Input:

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
chain sizes: 1 1 5
elapsed time: 0.000s
```



For example, the first integer ‘6’ represents the fact that the next pointer of block ‘0’ is block ‘6’. The graph on the right illustrates how the blocks are linked. There are 3 terminating blocks in the above FAT: blocks 4, 17 and 12. The longest chains ending on blocks 4 and 17 are both of length 1. The longest chain terminating on block 12 either starts on block 3 or 5, but in both cases the length of the chain would be 5. Therefore, for the above input, the `fat_check()` function should return `[1,1,5]`. Order the results in ascending order.

Limits:

- number of entries in FAT will be in the range [1 .. 10,000,000];
- your code should be 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 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**. 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 (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly. This list is not exclusive. 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. 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, which could make your program too slow for large number of requests. To get full marks, you will need to use smarter data structures. I suggest:

- `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
    - you need to store linked list iterators in this tree (aka pointers to the linked list nodes)
    - you should sort the tree by partition size (primary key), and partition address (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

If you use the above data structures correctly, they will allow you to process every request in $log(n)$ worst case time. Here are some relevant parts of my code:

```
struct Partition {
  int tag;
  int64_t size, 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, in a linked list
  std::list<Partition> all_blocks;
  // 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;
 ...
}
```

The `free_blocks` will keep the partition pointers sorted so that `free_blocks.begin()` will always give you the largest partition, and in case of ties, it'll give you the partition with the smallest address.

Before you modify a partition, make sure you `free_blocks.erase()` the partition. If you need to keep it in `free_blocks`, you can `free_blocks.insert()` it back after the modification. If you do not do this, you will likely get a corrupted tree and at some point, your program will output wrong results, or even crash.

Hints for debugging

I suggest you do a consistency check of your data structures after each request:

- make sure the sum of all partition sizes in your linked list is the same as number of page requests * page_size
- make sure your addresses are correct
- make sure the number of all partitions in your tag data structure + number of partitions in your free blocks is the same as the size of the linked list
- make sure that every free partition is in free blocks
- make sure that every partition in free_blocks is actually free
- make sure that none of the partition sizes or addresses are < 1

Also check the results of calls to `free_block.erase()` and `free_block.insert()` to make sure they work as intended.

Appendix – python solution for Q1

The GitLab repository contains a Python solution to Q1 called `memsim.py`. It is a very **inefficient** solution, so do not expect it to run very fast for large inputs. By default, is shows you the partition states after processing each request. To turn off this extra debugging output, you can specify the page size as a negative number on the command line. Here is the output it generates on `test1.txt`:

```
./memsim.py 1000 < test1.txt
alloc 5 100
  ------+-----+-----
    tag |  5  | -1
   size | 100 | 900
   addr |  0  | 100
  ------+-----+-----
free 5
  ------+------
    tag |  -1
   size | 1000
   addr |  0
  ------+------
free 6
  ------+------
    tag |  -1
   size | 1000
   addr |  0
  ------+------
alloc 1 100
  ------+-----+-----
    tag |  1  | -1
   size | 100 | 900
   addr |  0  | 100
  ------+-----+-----
alloc 2 20
  ------+-----+-----+-----
    tag |  1  |  2  | -1
   size | 100 | 20  | 880
   addr |  0  | 100 | 120
  ------+-----+-----+-----
alloc 1 100
  ------+-----+-----+-----+-----
    tag |  1  |  2  |  1  | -1
   size | 100 | 20  | 100 | 780
   addr |  0  | 100 | 120 | 220
  ------+-----+-----+-----+-----
alloc 2 30
  ------+-----+-----+-----+-----+-----
    tag |  1  |  2  |  1  |  2  | -1
   size | 100 | 20  | 100 | 30  | 750
   addr |  0  | 100 | 120 | 220 | 250
  ------+-----+-----+-----+-----+-----
alloc 1 100
  ------+-----+-----+-----+-----+-----+-----
    tag |  1  |  2  |  1  |  2  |  1  | -1
   size | 100 | 20  | 100 | 30  | 100 | 650
   addr |  0  | 100 | 120 | 220 | 250 | 350
  ------+-----+-----+-----+-----+-----+-----
alloc 2 40
  ------+-----+-----+-----+-----+-----+-----+-----
    tag |  1  |  2  |  1  |  2  |  1  |  2  | -1
   size | 100 | 20  | 100 | 30  | 100 | 40  | 610
   addr |  0  | 100 | 120 | 220 | 250 | 350 | 390
  ------+-----+-----+-----+-----+-----+-----+-----
```

```
alloc 1 100
  ------+-----+-----+-----+-----+-----+-----+-----+-----
    tag |  1  |  2  |  1  |  2  |  1  |  2  |  1  | -1
   size | 100 | 20  | 100 | 30  | 100 | 40  | 100 | 510
   addr |  0  | 100 | 120 | 220 | 250 | 350 | 390 | 490
  ------+-----+-----+-----+-----+-----+-----+-----+-----
free 2
  ------+-----+-----+-----+-----+-----+-----+-----+-----
    tag |  1  | -1  |  1  | -1  |  1  | -1  |  1  | -1
   size | 100 | 20  | 100 | 30  | 100 | 40  | 100 | 510
   addr |  0  | 100 | 120 | 220 | 250 | 350 | 390 | 490
  ------+-----+-----+-----+-----+-----+-----+-----+-----
alloc 2 21
  ------+-----+-----+-----+-----+-----+-----+-----+-----+-----
    tag |  1  | -1  |  1  | -1  |  1  | -1  |  1  |  2  | -1
   size | 100 | 20  | 100 | 30  | 100 | 40  | 100 | 21  | 489
   addr |  0  | 100 | 120 | 220 | 250 | 350 | 390 | 490 | 511
  ------+-----+-----+-----+-----+-----+-----+-----+-----+-----
free 1
  ------+-----+-----+-----
    tag | -1  |  2  | -1
   size | 490 | 21  | 489
   addr |  0  | 490 | 511
  ------+-----+-----+-----
alloc 3 220
  ------+-----+-----+-----+-----
    tag |  3  | -1  |  2  | -1
   size | 220 | 270 | 21  | 489
   addr |  0  | 220 | 490 | 511
  ------+-----+-----+-----+-----
alloc 3 759
  ------+-----+-----+-----+-----+------
    tag |  3  | -1  |  2  |  3  |  -1
   size | 220 | 270 | 21  | 759 | 730
   addr |  0  | 220 | 490 | 511 | 1270
  ------+-----+-----+-----+-----+------
alloc 3 1
  ------+-----+-----+-----+-----+-----+------
    tag |  3  | -1  |  2  |  3  |  3  |  -1
   size | 220 | 270 | 21  | 759 |  1  | 729
   addr |  0  | 220 | 490 | 511 | 1270 | 1271
  ------+-----+-----+-----+-----+-----+------
alloc 3 5900
  ------+-----+-----+-----+-----+-----+------+------
    tag |  3  | -1  |  2  |  3  |  3  |  3   |  -1
   size | 220 | 270 | 21  | 759 |  1  | 5900 | 829
   addr |  0  | 220 | 490 | 511 | 1270 | 1271 | 7171
  ------+-----+-----+-----+-----+-----+------+------

----- Results --------------------------------
pages requested:                8
largest free partition size:    829
largest free partition address: 7171
elapsed time:                   0.001
```

Appendix - hints for Q2

No advanced data structures are needed for this question. You should be able to implement a fast solution just using `std::vector`. You may want to make adjacency lists, similar to the way you may have used them in the previous assignment. It is very easy to figure out maximum chain lengths if you start tracing the graph from the '-1' nodes, for example using DFS traversal.

The GitLab repository contains a python solution called `fatsim.py`. This is an inefficient $O(n^2)$ solution, and it will only work for small inputs (it will take too long on large inputs).

Since most of you have a fairly restrictive file system quotas, I made a random FAT generator called `gen.py`. It takes 2 command line arguments: size of the FAT table, and a seed. It should always generate the same output for the same command line arguments. You can use it like this:

```
$ ./gen.py 10000000 33 | ./fatsim
chain sizes: 22562 22929 40123 58711 61844 68366 84036 86491 92003 95352 97758 113158
  116995 125994 128294 152189 153320 162697 169201 169266 184034 187839 198301 215629
  276541 289916
elapsed time: 3.460s
```