CS307 - Fall 2019-2020
Memory Management API - Phase 2

Date Assigned: 18.11.2019
Due Date Time: 28.11.2019 at 23:55 (sharp, according to server's time)
Late Policy: For every 10 minutes of late submission 1 point will be deduced.

1 Introduction

The part of the operating system that manages (part of) the memory hierarchy is called the memory manager. Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

In general terms, there are two ways to keep track of memory usage: bitmaps and free lists. With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).
Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.

In this homework you are expected to implement a memory manager which maintains a linked list for organizing the memory allocation.

2 Program Flow

You are expected to use POSIX threads which will work on the shared data structures. You should use POSIX threads, semaphores, mutex or mutexes, linked list and queue data structures and a main memory array.

2.1 Data Structures

In general case, the memory is divided into fixed size of chunks. In this homework we are assuming one chunk is 1 Byte.

2.1.1 Memory Array

The memory array that will be used in this homework will be a char array with size *M*. Since each chunk is 1 Byte size, total memory will be *M* bytes size. If the memory address is free, there should be *X* in the proper memory index. If it is allocated by a thread, the thread *ID* should be written in the proper memory index.

2.1.2 Linked List

You need to maintain a linked list for free and allocated memory indexes. The purpose of the linked list is tracking the memory state. The linked list should be updated with each memory allocation or deallocation.

ID	SIZE	INDEX

Figure 1: A linked list node

A linked list node should contain; *ID*, *SIZE* and *INDEX* variables.
A hole (free memory) or an allocated thread information could be represented by a linked list node.
For a thread; *ID* denotes the unique integer given to that thread, *SIZE* denotes the random integer created during the execution. *INDEX* denotes the starting memory index of the thread.
For a hole; *ID* equals to -1, *SIZE* denotes the size of free memory, *INDEX* denotes the starting memory index of hole. The linked list should be formed by using the above node.

You can use C++ Standard Library linked list or implement your own linked list class. Also, you are allowed to use any type of linked list, which first fit algorithm can work, as long as it works properly during the memory allocation.

2.1.3 Request Queue

You need to maintain a queue for thread memory requests. The purpose of the queue is ordering the thread requests so that memory server can process them one by one.

ID	SIZE

Figure 2: A queue node

A queue node should contain; *ID* and *SIZE* variables.
ID denotes the unique integer given to that thread, *SIZE* denotes the random integer created during the execution.

You can use C++ Standard Library queue or implement your own queue class.

2.2 Main Thread

You need to initialize any related data structure before starting the memory server thread. Memory server thread will run in a function called **server_function**. After the initialization, you are expected to start all of ordinary threads, which will run in a function called **thread_function**. After that the main thread should sleep for 10 seconds and call **release_function**.

In other words, after the creation of threads, your program should terminate at the end of 10 seconds.

2.2.1 release_function

After 10 seconds, main thread should call release function to terminate the program. Inside of the **release_function**, linked lists should be deallocated and memory should be returned to its initial state. After releasing the allocated memory, your program should terminate.

2.3 Ordinary Threads

After the creation of these threads, they will run in a **thread_function**. A Thread will follow the cycle denoted in Figure 3. The generated size will be between 1 and 1/3 of the total memory size. Thread should call **my_malloc** function. If there is enough space in the memory for a thread, **use_mem** and **free_mem** functions will be called.

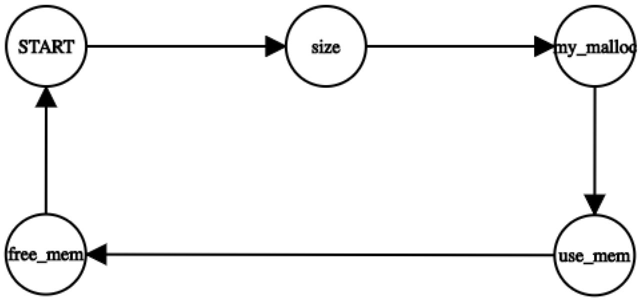

Figure 3: Thread pipeline.

2.3.1 my_malloc

Thread should add its *ID* and *size* to the queue and block on its semaphore. After memory server thread increments the semaphore of the blocked thread, thread should continue and check the linked list to see if it is allocated or not. Thread could use or not use memory based on the return value of the **my_malloc** function.

2.3.2 use_mem

Since threads need to complete their jobs, they need to spent some time in the memory if they are allocated. To simulate this behaviour, threads will sleep inside of this function. A thread needs to wait between 1 and 5 seconds.

2.3.3 free_mem

After a thread is done with using memory, the memory that thread used would be freed by memory manager. For the sake of simplicity in this homework, threads are able to do it by themselves. When a thread is done, thread will update the linked list and it will update the memory array.

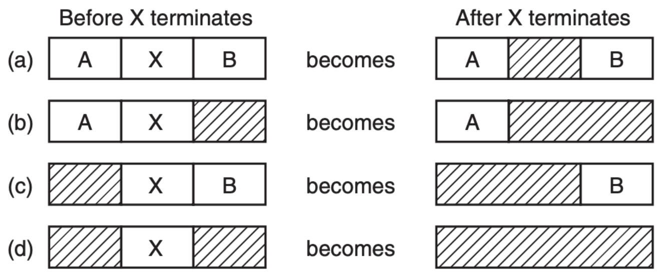

Figure 4: Possible states of the linked list after thread terminates.

Assuming a thread is allocated between two other threads, the termination could result four possible cases of memory alignment as its shown in the Figure 4. If you do not update the linked list holes properly, it will contain consecutive small sized holes. Even though, there is an enough size in the memory for a new thread, memory manager could not allocate memory because of this small holes. Thus, after each thread termination you need to check the linked list and merge the holes to avoid such case.

2.4 Memory Server Thread

After the creation of memory server thread, it will run in a **server_function**. Memory server thread will continuously check the request queue to see if a thread wants to allocate memory. If there is an element inside of the queue, memory server thread will process the element and pop the element from the queue afterwards. This processing is done by checking the linked list. If there is a hole which is big enough for the requested size, memory server thread will grant access by updating the linked list by adding the ID, size and starting index. It should also update the memory array with using the ID of the thread. Afterwards,it will increment the semaphore of the thread to allow its execution and pop the element from the queue. If there is not enough size for the requested size, memory server thread should increment the semaphore of the thread and pop the element from the queue. In order to check the size, you are expected to use first fit algorithm on the linked list.

If memory is granted to the thread, memory server thread will call **dump_memory** function to print out the current state of the linked list and the current state of the memory array.

2.4.1 dump_memory

This function will be used for printing the current state of the memory and the state of the linked list. First you need to print the linked list and then you need to print the memory array. In order to print the linked list and memory array, you are expected to use the following convention.
For representing a single node:

```
[ID][SIZE][INDEX]
```

An example linked list of nodes:

```
[ID][SIZE][INDEX]---[ID][SIZE][INDEX]---[ID][SIZE][INDEX]
```

An example memory array with memory size 5 and 3 threads:

$$[ID_2]\ [ID_2]\ [ID_1]\ [X]\ [ID_0]$$

Where ID_i denotes the *thread ID* and *X* is used for free memory.

3 Sample Runs

Sample Run 1

Below example is a test run for 3 threads, (t_0 t_1 ,t_2) , working on a 10 bytes of memory array. In this example, t_0 allocated memory first, t_1 allocated memory second and t_2 allocated memory third.

After that, t_1 and t_2 finished their jobs and deallocated memory. Since these threads are finished they should create a new request in the queue, and as its seen from the output t_2 sent its request before t_1 and allocated memory before t_1.

After the t_1 allocated the memory, t_0 finished its job and sent another request to the queue with size of 3 bytes. Since the first hole is not big enough for t_0 requested size, the second hole checked and the memory index 5 is granted to t_0 as starting index.

Then t_0 and t_2 finished their jobs and added another request to the queue. Since t_2 was faster in this step, t_2 pushed its request to the queue faster than t_0. Thus, it is allocated before t_0.

```
List:
[0][1][0]---[-1][9][1]
Memory Dump:
0XXXXXXXXX
********************************
List:
[0][1][0]---[1][2][1]--[-1][7][3]
Memory Dump:
011XXXXXXX
********************************
List:
[0][1][0]---[1][2][1]---[2][2][3]--[-1][5][5]
Memory Dump:
01122XXXXX
********************************
List:
[0][1][0]---[2][2][1]--[-1][7][3]
Memory Dump:
022XXXXXXX
********************************
List:
[0][1][0]---[2][2][1]---[1][2][3]--[-1][5][5]
Memory Dump:
```

```
02211XXXXX
********************************
List:
[-1][1][0]-[2][2][1]---[1][2][3]---[0][3][5]-----[-1][2][8]
Memory Dump:
X2211000XX
********************************
List:
[2][3][0]---[1][2][3]--[-1][5][5]
Memory Dump:
22211XXXXX
********************************
List:
[2][3][0]---[1][2][3]---[0][2][5]--[-1][3][7]
Memory Dump:
2221100XXX
********************************
.
.
.
(Print until main terminates)
```

Sample Run 2

Below example is a test run for 10 threads working on a 20 bytes of memory array.

```
List:
[0][3][0]---[-1][17][3]
Memory Dump:
000XXXXXXXXXXXXXXXXX
********************************
List:
[0][3][0]---[3][5][3]---[-1][12][8]
Memory Dump:
00033333XXXXXXXXXXXX
********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[-1][11][9]
Memory Dump:
000333331XXXXXXXXXXX
********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[4][4][9]---[-1][7][13]
Memory Dump:
```

```
0003333314444XXXXXXX
*********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[4][4][9]---[5][2][13]---[-1][5][15]
Memory Dump:
00033333144445 5XXXXX
*********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[4][4][9]---[5][2][13]---[2][1][15]---[-1][4][16]
Memory Dump:
0003333314444552XXXX
*********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[4][4][9]---[5][2][13]---[2][1][15]---[8][1][16]
---[-1][3][17]
Memory Dump:
00033333144445528XXX
*********************************
List:
[0][3][0]---[3][5][3]---[1][1][8]---[4][4][9]---[5][2][13]---[2][1][15]---[8][1][16]
---[6][3][17]
Memory Dump:
00033333144445528666
*********************************
List:
[-1][3][0]---[3][5][3]---[-1][1][8]---[4][4][9]---[-1][2][13]---[2][1][15]---[-1][1][16]
---[6][3][17]
Memory Dump:
XXX33333X4444XX2X666
*********************************
List:
[7][3][0]---[3][5][3]---[-1][1][8]---[4][4][9]---[-1][2][13]---[2][1][15]---[-1][1][16]
---[6][3][17]
Memory Dump:
77733333X4444XX2X666
*********************************

.
.
.
(Print until main terminates)
```

4 Testing

In order to understand that threads are working as intended, you need to compare the linked list and the memory array. Both data structures have to be consistent with each other. Your program should work on different number of threads and different sizes of memory.
Also, you need to use semaphores and mutexes properly to get full grade. You should not use unnecessary semaphores/mutexes, or mutexes to lock any actions. You should definitely avoid using other means of aligning your threads. Having a proper output without using semaphores and mutexes will result an invalid solution. Thus, you need to be careful with mutual exclusion and race conditions.

Submission

Solutions should be submitted in a zip archive, name your zip archive as:
YourNameSurname_ID_hw4.zip and submit to **SUcourse**.
You need to add comments to the your code to make it more understandable.

Note that, your system time and SUcourse server's time may not be synchronized so do not wait the last minutes to submit your solution. Only the solutions in the SUcourse system will be graded. Other submissions, such as emailing to instructor or assistants, will not be graded.

Late Policy:
You are allowed to late submit your homework. Our policy for late submission depends on how late you submit. For every 10 minutes you are late, 1 point from your grade will be deducted. Please do not ask for any relaxation. Below you can find the table for the point deduction system:

Submission Time	**Deduction points**	**Max Grade**
23:56 - 00:05	1	99
00:06 - 00:15	2	98
00:16 - 00:25	3	97
00:26 - 00:35	4	96
00:36 - 00:45	5	95
00:46 - 00:55	6	94
00:56 - 01:05	7	93
01:06 - 01:15	8	92
...	...	...
...	...	...
...	...	...
15:46 - 15:55	96	4
15:56 - 16:05	97	3
16:06 - 16:15	98	2
16:16 - 16:25	99	1
16:26 - 16:35	100	0