CS 307 – Operating Systems

Fall 2019 - Homework 3

Memory Management API - Phase 1

This homework is prepared by Barış Altop and Berkant Deniz Aktaş under the supervision of Yücel Saygın.

Date Assigned: 04.11.2019
Due Date Time: 14.11.2019 at 23:55 (sharp, according to server's time)
Late Policy: For every 10 minutes of late submission 1 point will be deduced. (Details at the end)

Introduction

In this project, you are going to implement a memory heap management API, and we will provide a code for you to test your API. You are expected to create a shared memory, in order to manage memory requests coming from multiple threads and grant or decline requests based on free space available. You are also expected to manage all mutexes and semaphores for shared memory access.

Program Flow

The main program, that we will give you, will call various functions in order to initialize, access, change and dump the memory. Also there will be an array of threads, an array of semaphores, a mutex, a memory server thread and a char array representing the memory given.

The **`init()`** function that is also provided will initialize the memory array and the request queue and any related data structures. Afterwards it will start the memory server thread.

After initialization, you are expected to start all of threads, which will run in a function called **`thread_function(int id)`**. This function will generate a random size and call the **`my_malloc(int thread_id, int size)`** function, which you will also implement. My_malloc will utilize the semaphores and mutex for synchronization of access to shared data structures. The **`thread_function`** will wait for the answer and -if memory is granted- access the memory and set all the bytes allocated to it to character value **`'1'`**. The size of the memory will be randomly generated by the **`thread_function`** function.

The memory server thread is expected to handle all requests for memory access. All access requests will be appended to the shared queue. The server thread will read from this queue and depending on the size and availability will return an answer to the requesting thread.

- If there is enough space it should return the start address of the memory array.
- If there is not enough space it should return -1.

For this phase of the assignment you will only maintain an index of the next available space. You are not expected to manage the memory allocations. The access will be based on the index and the space remaining in the memory array. If the requested size is smaller than `(memory_size - index)`, then you should grant access.

The thread which makes the request will be blocked till the memory server processes its request

- If space is allocated: Gains access to the shared memory and set all the bytes allocated to it to character value **'1'**
- If space is not allocated: Prints an error message "Thread ID: Not enough memory" and exits.

Expected Functions

my_malloc(int thread_id, int size): This function should be called by a thread with a size value no bigger than the memory itself. It will gain access to the shared queue and write the requesting **thread_id** and the amount.

dump_memory(): This function is used to test memory allocation. It will print the entire contents of the memory array onto the console.

thread_function(int id): This function should run as a thread. The function should first create a random memory size. It will then call the **my_malloc(id, size)** function and block until the memory request is handled by the memory server thread. Once the thread is unblocked by the memory server thread it will take appropriate action. The memory server thread will store the value in a shared array called **thread_message**. The value is -1 indicates there is no available memory, hence only an error message must be printed. If the value is bigger than -1 then it will be interpreted as the starting point in the memory. It will set all the bytes allocated to it to character value **'1'.** This entire process will only run once, i.e. each thread will only make one memory request.

server_function(): This thread will run until **release_memory** function is called. It will check the queue for the requests and depending on the memory size it should either grant or decline them. The answers to the requests will be written to the **thread_message** array and the requesting thread should be unblocked. It should return either -1 for declining the request due to no available size, or the start_point of the memory location that will be allocated to the requesting thread.

release_function(): This function will be called whenever the memory is no longer needed. It will kill all the threads and deallocate all the data structures.

thread_id: This value will be incrementally generated and passed to the thread_function as a parameter starting from 0 (zero), while the threads are being created.

Shared Data

```
char memory[MEMORY_SIZE];
pthread_mutex_t sharedLock;
sem_t semlist[NUM_THREADS];
int thread_message[NUM_THREADS];
queue<node> memory_queue;
```

Submission Guides:

Solutions should be submitted in a zip archive, name your zip archive as: ***YourNameSurname_ID_hw3.zip*** and submit to **SUcourse**.

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.

You will be graded on the correctness of your solution and your use of semaphores based on the code we have sent. You shouldn't use unnecessary semaphores/mutexes, or mutexes to lock any actions. You should definitely avoid using other means of aligning your threads. All of these dirty tricks may actually make your code look like it has "somehow finally started working" but in fact they will cause your solution to be invalid!

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