UĞUR KAĞAN ÇAKIR - 27058

p.s.: I have used semaphores in my implementation. In order to have semaphores, I've included semaphore library.

#include <semaphore.h>

I think my code can only run when we link pthread library (by using "-lpthread") when we execute my code. But, since we've already used pthreads in the sample runs, I thought that the pthread library will be linked when you compile our codes. I hope this won't be a problem, I have tried my code in virtual ubuntu machine, and it worked correctly when I had used -lpthread.

PA4- REPORT

I have used semaphores as the locking algorithm in my code. But they work as mutexes since, the allowed number of threads in a semaphore protected code is one. I have used in total 2 semaphores, 1 of them is shared between myMalloc and myFree functions, and the other one is for the print() function. To show how the locking mechanism is working, I will first show my definition of HeapManager class.

```
#include <semaphore.h>

struct memo_chunk

{

        int id;

        int size;

        int index;

        memo_chunk * next;

}


class HeapManager

{

public:

        int initHeap( int );
```

```
        int myMalloc(int, int);
        int myFree(int, int);
        void print();
        ~HeapManager();
        HeapManager();

private:
        memo_chunk * head;
        sem_t mutex; // the mutex used in myMalloc and myFree
        sem_t print_mtx; // the mutex used in print function
};
```

As you can see the semaphore objects are given as the private objects of the class. We initialize these semaphores inside the default initializer.

```
HeapManager()
        Head = null
        Initialize mutex semaphore with the maximum of threads allowed at a time is set to 1
        Initialize print_mtx semaphore with the maximum of threads allowed at a time is set to 1
```

But inside the initHeap function, the semaphores are not used, only the heap is initialized.

```
initHeap( int size)
        initialize heap with the given size
```

Then, after initializing the heap, threads can call myMalloc and myFree functions concurrently to allocate and deallocate spaces from the heap. However, we shouldn't let them update the heap concurrently, since the stage of the heap changes after each operation. Whenever one thread tries to allocate memory from this heap, the other threads should wait until the allocation is complete. Similarly, whenever a thread tries to free a memory that is already allocated none of the other threads should be

able to neither allocate a memory nor free a memory from this heap. Therefore, the access to these functions should be done one by one. For this reason, I have used a single mutex (I've implemented as a semaphore, but it basically works as a mutex) for both of the functions, surrounding the codes inside these functions.

```
myMalloc( int id, int size)
        sem_wait (&mutex) // lock the mutex
        bool isSpaceFound = false // this is the variable I've used to keep the information whether we found a suitable space for allocation or not
        try to allocate memory
        if ( isSpaceFound )
                then print success message, print the heap
                        sem_post( & mutex) //release the lock
                        return
        else
                then print failure message, print the heap
                sem_post( &mutex ) //release the lock
                return


myFree (int id, int index)
        sem_wait( & mutex ) // lock the mutex
        bool doesNodeExists = false // this has a similar function to isSpaceFound
        try to find the chunk of memory with id = id and index = index
        if ( doesNodeExists )
                then print success message, print the heap
                sem_post (&mutex) //release the lock
                return
        else
                then print failure message, print the heap
```

```
        sem_post (&mutex) //release the lock
        return
```

As it can be seen from the pseudocode, the mutex is locked at the start of the functions and it is released right before the return statements. Therefore, since everything is surrounded between sem_wait and sem_post the functions will be executed atomically. Hence, we can't have two threads that one using one of the myMalloc or myFree functions and the other one also using one of the myMalloc or myFree functions, at the same time. Therefore, the state of the heap will always be updated atomically. Moreover, the print function is called before releasing the mutex which is also very important since by doing this we guarantee to print out the updated version of the heap after we allocate or deallocate a memory from the heap. If we have released the lock and then tried to print the heap the state of the heap might had been changed.

Lastly, the print function also uses a mutex, which is exclusive to the print function. This mutex is implemented in order to prevent threads to print out the table at the same time.

```
Print()
        Sem_wait( & print_mtx )
        Print the table
        Sem_post ( & print_mtx )
```

Finally, when the main operation terminates, the destructor function is called. Inside the destructor the nodes of the heap are deleted, and the semaphore variables are destroyed.

```
~HeapManager()
        Iteratively delete each node of the heap
        Destroy mutex
        Destroy print_mtx
```