Kemal Yılmaz 31097

PA4 Report

In my project I used a built-in C++ linked list for simplicity purposes. The list is a private member of the HeapManager class.

As my locking mechanism I used a simple mutex lock to ensure atomicity of the operations. The details of the implementation will be in the report.

struct linkNode: Holds id, size and index of our virtual heap node.

HeapManager: Creates an empty link node and pushes it to the linked list.

initHeap: Initializes the empty link node with the heap size and prints the heap.

myMalloc: Looks for available space for the memory demand of the thread. Using a while loop and an iterator. Checks two conditions whether the available space is bigger or the same size as the demanded size. Updates the heap list in accordance with the homework requirements.

myFree: Identifies the space to be freed and frees it then checks the neighbor nodes for possible coalescing operations. It is also done using a while loop and an iterator but includes additional checks to ensure the iterator doesn't go out of bounds when nodes are deleted after coalescing.

print: It is a private function called by only the member functions of the HeapManager class to prevent data races. It prints the current state of the heap list.

Locking algorithm:

class HeapManager{

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

def struct linkNode;

public:

//constructor

def initHeap(int size){

```
 lock(mutex)
// initialize the first node with free space size
print()
unlock(mutex)
}

def int myMalloc(params){
lock(mutex);
// iterate the list and allocate space update the list
print();
unlock(mutex);
}

def int myFree(params){
lock(mutex)
// iterate the list, free the space node, coalesce if possible, update the list
print();
unlock(mutex);
}

private:

list<linkNode> list_;
def print(){
//print contents of the list_
}
}
```

As can be seen in the pseudo code in initHeap, myFree and myMalloc functions, all operations are done within the bounds of the globally declared mutex lock. This ensures that firstly the heap is initialized safely, even though it is probably going to be called once. malloc and free operations will also be done atomically and safely as any function that accesses the list in any way is mutex locked so when malloc of a thread is in progress no other thread can access any operation that reads or writes to the heap list. An issue that I faced during the implementation was the print() function because if it was mutex locked it'd create a deadlock as it is called within mutex locked functions. As a way to solve this, I made print() a private function so that it can not be called outside of the class and all print calls are done by mutex locked functions.