SLAB ALLOCATOR

OS - II

OM SITAPARA
CS16BTECH11036

HARSHIT PATEL
CS16BTECH11017

TABLE OF CONTENTS

LIBMYMEM.HPP

Data-Structures:

1) **struct bucket:**
   a) **Int objectsize :** Size of each slab object.
   b) **struct slab* firstSlab :** Pointer to first slab of that bucket

2) **struct object :**
   a) **struct slab* slabptr :** Pointer to slab which this object belongs to.
   b) **void* data :** user defined data
   c) **mutex mtx :** a mutex lock to make library thread safe.

3) **struct slab :**
   a) **int totalObj :** total number of objects which slab can hold.
   b) **int freeObj :** total number of free objects in the slab.
   c) **bitset<16000> bitmap :** A bitmap for finding the free object in a slab.
   d) **struct bucket* bucketPtr :** A pointer of slab pointing to the bucket which it belongs to.
   e) **struct slab* nextSlab :** A pointer to next slab.
   f) **void* offset :** This pointer shows the point from which allocation of objects should start.

VOID* MYMALLOC(UNSIGNED SIZE):

This function first finds the designated bucket for that object size. Now it creates an object of given size. Then it finds the free memory in the slab and assigns it to this newly created object using bitmap and finally returns a void* pointer pointing to this object.

VOID MYFREE(VOID* PTR):

This function takes a void pointer pointing to the memory address which is to be freed. In this function first we take out the slab pointer from the void* ptr and then we find at which position this object in slab is allocated. If there are more objects in the slab then we just make the bitmap entry from 1 to 0 and if the last object in the slab is being deleted than we munmap the whole slab.

LIBMYMEM.CPP

Define and global Variables:

1) **#define slabSize 65536 :** Defines that slab size if of 64KB.

STRUCT SLAB* SLABCREATOR(UNSIGNED SIZE, INT INDEXBUCKET):

The motive of this function is to create a new slab using mmap and return its pointer.
First we create a binary file of specified size using **truncate –s 1G 1mfile.** If we get error while opening it we print that error and return the program. **1mfile is of 1GB.**
Now we create a void* ptr and create a memory of 64KB using mmap using the following command:

```
(ptr1 = mmap(NULL, slabSize, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0)
```

So, by this we create a private memory of 64KB which has read and write access. If the following expression fails than we print mmap failed and return the program. This created memory writes to this 1mfile which was created earlier.
Now we create a slab pointer pointing to ptr1 and set the attributes of this slab pointer.
We set totalObjects by dividing the size of slab – headerSize to size of one object
Size of one object = sizeof(struct slab*) + user size.
We initially set freeobjects to total objects and all entries of bitmap as 0 along with nextSlab pointer to NULL. We set the offset pointer to slabPointer + sizeOfheader where

sizeOfheader = sizeof(data structers of slab).

Finally, we return **ptr** which is the pointer pointing to the newly created slab.

VOID* MYMALLOC(UNSIGNED SIZE):

First we find which bucket this specified size belongs using `int index = ceil(log2(size)) - 2;`
First of all it locks that bucket so that two threads cannot write to same bucket at same time.
Now we check that the bucket at this index has slab or is null. If it is null then we create a slab and make `bucket[index].firstSlab` to point at newly created slab.
Now we traverse thru each slab finding the free memory in the first available slab. If the slab are full and there is no new slab then we create one and make current slab to point at that slab.

```
if(current -> nextSlab == NULL)
            {
                current -> nextSlab = slabCreator(size,index);
            }
                current = current -> nextSlab;
```

Now when we get a free point in a slab at index i we set the bitmap of that position 1 we decrement the freeobj of that slab by one. Now we create a pointer of void* allocationPoint which gives the position in memory at which the object struct should be written. Then we make new struct object* pointer pointing to this allocation pointer and write the data as:

```
            newObj = (struct object*) allocationPoint;
            newObj->data = i;
            newObj->slabptr = current;
```

Then we return the void* at user data by incrementing the allocationPoint by the sizeof(struct slab*).

So, now the user can write its content from the returned pointer.

VOID MYFREE(VOID* T_PTR):

When a *ptr is given for deleting the memory first we extract the object in which this was written by creating a struct object* pointing to this pointer – sizeof(struct slab*). **We do this because the ptr is pointing to the user data and the object in which this user data is stored has a slab pointer before this data that's why we need to subtract the size of struct slab*.**

As we have the pointer to the slab in which this object was allocated we find the index at which this object was allocated.

Now we use mutex lock to lock this current bucket so some other thread may not read or write at this block while it is deleting.

We use memset to write garbage value at this point so that when this pointer is used again for allocating user data the user cannot trick and start reading the data.

```
 long int x = (char*)((char*)t_ptr-sizeof(struct slab*))-(char*)((t_came->slabptr)-
>offset);
int noOfObjectsOffset = x/(t_came->slabptr->bucketPtr->objectSize + sizeof(struct
slab*));
```

As we get the index in slab as at which it was allocated we make its bitmap to 0 and increase the number of freeobjects in that slab. Now if the totalObjects are equal to freeObjects than we need to delete the whole slab and unmap its memory. This operation is same as deleting the node from a linked list. We unmap the memory using

```
munmap((void*)(t_came->slabptr) ,slabSize);
```

MEMUTIL.CPP :

SEQUENTIAL:

This program randomly choses size from 1 to 8192 bytes and allocates it into the memory and makes the program to sleep for some random amount of time and frees the earlier allocated memory. The allocation and freeing is done with the library implemented by us using the functions of mymalloc() and myfree(). This is done for n number of iterations.

PARALLEL:

This program creates t threads for which each thread does n iterations. In each iteration the thread randomly chooses a bucket and tries to allocate that object. Then it sleeps for a random amount of time and then frees that allocated object. The allocation and freeing is done with the library implemented by us using the functions of mymalloc() and myfree().

PROBLEMS FACED:

The main problem faced was the management of pointers and the arithmetic operation on the pointers which led to bunch of warning messages which took some amount of work to remove them. Also understanding the problem how to write at a particular memory address took some time. Else it was pretty normal.