Saksham Mittal
CS16BTECH11032

Sudhanshu Chawhan
CS16BTECH11037

23 April 2018

Operating Systems II

Programming Assignment 5: Linux Slab Allocator

Problem statement :

Implementing a thread safe and re-entrant Linux slab header.

Slab allocation :

A slab is a collection of objects of the same size. It avoids fragmentation by allocating a fairly large block of memory and dividing it into equal-sized pieces

In our implementation, a bucket (hash table) is used which hashes the size of required memory into their respective closest memory objects in the table entry. Each entry contains pointer to the first slab in the linked list. Each slab in the linked list of a particular entry has same object size.

Slab consists of a header and contiguous objects. Header keeps track of total objects, free objects, bitmap, pointer to bucket and other useful metadata. The required memory address is returned from the contiguous object memory whichever is free. Bitmap keeps track of the free objects.

Code Definitions :

Struct header consists of :-

```
struct header {
    int totobj;
    int freeobj;
    bitset<16000> bitmap;
    struct bucket* buck;
    struct slab* nxtSlab;
};
```

totobj : total objects in the slab

freeobj : total free objects in the slab

bitmap : bitwise mapping where 0 means free and 1 means used

buck : pointer to the bucket entry

nxtSlab : pointer to the next slab in the linked list

Struct object consists of :-

```
struct object{
    struct slab* slbPtr;
    int data;
};
```

slbPtr : pointer to the slab header

data : data to be filled in the memory

Struct slab consists of :-

```
struct slab {
    struct header slabHeader;
};
```

slabHeader : Header of the slab

Struct bucket consists of :-

```
struct bucket {
    int objSize;
    struct slab* firstSlab;
};
```

objSize : size of the objects to which the entry corresponds

firstSlab : pointer to the first slab in the linked list

Code Design :

void* mymalloc(unsigned size) :-

Allocates memory of the required size in the appropriate slab and returns pointer to the allocated memory address.

Working

* Finds the bucket entry index

* If the first slab is null then make a new slab and initialise its metadata. Make the first object's (from the contiguous memory) slab pointer point to the current slab and return the address of the data memory.

* Else traverse the slabs in the linked list until traversed slab has at least one free object or end is reached.

* If end was reached then there were no slabs free. So create a new slab and follow the same steps as done in upper section.

* If a slab is found that is free then find the first object in the bitmap that is 0. This gives us the position of the free object. now find the address of this object and make the object's slab pointer point to the current slab and return the address of the data memory.

* At the end of each allocation, update the metadata of the current slab.

void myfree(void* ptr) :-

Frees the memory of the occupied object from the slab and updates the slab's metadata. If slab is now free then deallocates the entire slab.

Working

* Finds the address of the object to free and uses slab pointer to get the current slab's header.
* Finds the index of the object to free using pointer arithmetics.
* Updates the bitmap and other metadata using this index
* If current slab is completely free then deallocates this slab safely by not harming the linked list

Commands for compiling the library and running the memutil

```
truncate -s 1G 1mfile
g++ -std=c++11 -c libmymem.cpp -pthread
g++ -Wall -Werror -fpic -c libmymem.o -I . libmymem.cpp -std=c++11
-lpthread
g++ -shared -o libmymem.so libmymem.o

g++ -I . -L . -Wall -o memutil memutil.cpp -l mymem -std=c++11 -
lpthread
./memutil -n 5 (./memutil1 -n 100 -t 2)
```