OPERATING SYSTEM

SLAB ALLOCATOR

April 22, 2018

CS16BTECH11035

CS16BTECH11027

Overview.

The main purpose of slab allocator is to minimise the fragmentation and avoid compaction. As the kernel demands for objects ,we look into the hash table for the nearest size and then start traversing into the list of slabs. The slabs can be of three types :-

1. Full. All objects in the slab are marked as used

2. Empty. All objects in the slab are marked as free.

3. Partial. The slab consists of both used and free objects.

Goals

1. **Implementation of library libymem.so**
2. **To make Linux Slab allocator re-entrant & thread-safe**

Specifications

When a first demand is made for the object size , we have to allocate 64kb to the slab through mmap . The slab will then be used for the further demands of the objects until it is full . The basic structure of slab consists of :-

1. Header :-contains entities like bitmap ,totobj ,freeobj .(max size)
2. Objects :- The rest of the slab is divided into chunks of object size .

If the user demands for an object size and the corresponding slab is full ,we have to add a new slab to the corresponding linked list . After allocation ,the object request will be satisfied by this slab.

Thus, whenever a demand for object is made ,we traverse the corresponding linked list searching for a slab .

If the object has served it's purpose then we can set it free .The corresponding chunk can then be reused (overwritten) again.But if all the objects of a slab are set free ,then we deallocate the slab .Deallocation of slab is similar to deleting a node from linked list .We use munmap in this case .

Initialisation:

In the initialisation function ,the hash table is initialised .Here the hash table is an array of buckets.Bucket is a structure with the following entities :- 1) a (object size) 2) totobj(total objects) 3) pointer (pointer to the first slab) buck_lock(to protect simultaneous access to given bucket) .

Allocation Process:

This section consists of two functions :

void* mymalloc (uint64_t size) ,

void* myallocator(uint64_t size).

Void* mymalloc (uint64_t size)

Returns the void pointer to the chunk of object size

In this function ,the initialisation function .This intialises the hash table as decribed previously .Now the myallocator function is called with the size as arguement . We know that the hash table is to be initialised only once .Therefore,the initialisation() function is called only once .

Next the myallocator function output is returned .

Void *myallocator(uint64_t size)

Firstly we search the hash table with the size just greater than or equal to the object size .

Once we get the desired index of hash table ,we check if the hash_table[i].pointer is NULL or not .If it is NULL,it means that this is the first demand and no slab has been allocated yet . Therefore we have to invoke mmap which will provide a mapping of contiguous pages .The desired size and other parameters are passed to mmap function and it returns a void pointer to the slab .

```
f ((buffer = mmap(NULL, SZ, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0)) == MAP_FAILED)
        {
                perror("mmap");
                exit(1);
        }
```

Since the hash table pointer(pointer in structure) should point to the first slab , the pointer is typecasted to uint64_t .

It is clear from this line of code :

```
hash_table[i].pointer=(uint64_t)buffer ;
```

Buffer is again type casted into slab* .This is because we have to access the slab .On the way ,we also set the entities of header section of the slab . We know that header consist of bitmap ,the pointer to next slab ,pointer to previous slab etc.The total maximum size of the header will be 1048 bytes (1024 +24).1024 bytes are required for bitset .How? The minimum size of the object is (4B + 8B).4B =object size and 8B =size of slab ** pointer .

Maximum no of objects = 64kB/(4+8)= 5246 (approx).The nearest power of 2 is 8192 .Each bit stands for one object . Therefore 8192 bits =1028 bytes.

Here for traversing across the slab ,the buffer is type casted to bool as bool is 1 BYTE.It is again type casted to void* .

If the hash_table[i].pointer is not NULL ,it means that there is already a slab . Thus we then traverse the linked list of slabs and check if it is partial or full . For checking if the slab is partial or full ,we use the bitmap count function (here foo is bitmap) .If the count is totobj then we know if it is full . Count refers to no. of ones in the bitset.

If while traversing we coe at the end (i.e slabptr is NULL) then you have to allocate a new slab in the same manner described earlier .

If the salbptr is not NULL , it means we have a partial slab ,now what we have to do is to check the first 0 in the bitmap .The index of the first 0 is used in getting the desired chunk location .After simple loop ,we get a count .The slabptr is then again typecasted to bool and then we traverse and obtain the memory location .The void pointer is then returned .

Deallocation Process:

This involves the following function

void myfree(void * ptr)

When a object has completed it's use , it's slot can be freed .Here the slot can be freed means the chunk can be overwritten by some other object in future. In this function ,the (void) pointer to the object which is to be freed is passed as arguement . Once the ptr is obtained it is typecated to bool* so to traverse the slab . It is decremented by 8 bytes and we reach the memory location containing the start address of slab .Therefore it is typecasted to slab** and

we reach the starting point of the slab . Now we reach the corresponding bit of the bitmap and just set it to 0.The exact index of the bit can be obtained by a simple arithemtic of the pointers . We also have to check if the total slab has became free (all bits set to 0) . If the slab is free ,we have to remove it from the linked list and then munmap it .

The poniters to the prev slab ,the nxt slab have to be adjusted .

In case if the slab is the first ,we also have to adjust the bucket pointer (hash_table[i].pointer) .

Use of Bitmap:

Total no of bits = total no of objects .

Here maximum no of bits = 64kB /(4B +8B) = 5246 .Therefore the no of bits =8192 (nearest power of 2).

Initially all the bits are 0 . In case of allocation , the corresponding indexed bit is set to 1 . In case of freeing ,the bit is again reset to 0 .

ENSURING THREAD SAFETY AND REENTRANCY:-

The main difficulty in allowing two or more threads for accessing a particular bucket is race condition .Suppose two threads demanding same size object are allowed access simultaneously ,then they might set the same bit and thus overwriting each other . A major problem occurs when a thread allocates at the same slot which other thread is releasing .All other slots are free .So the thread releasing the object will see all other bits to be 0 Therefore the thread will munmap the whole slab .But the other thread has lost its data because it was writing at the same slot simultaneouslly .

The shared entity (list of slabs) has to be protected thus allowing a mutual access .It can be achieved by using locks like semaphore ,mutex.Basically we set one lock for each memory size .Sdo as there are 12 entries in our hash table ,we have 12 mutex locks.The lock will be part of the structure of the bucket . So a thread requesting a object size has to wait if a thread is already accessing the slabs .