CS314 : Operating Systems Laboratory

Lab 3 Report Group 10

Utkarsh Prakash - 180030042 Shriram Ghadge - 180010015

---

1. Introduction

In this lab we were expected to implement `malloc()` like functionality i.e. build a memory manager. In the first part the heap size was fixed to **4KB** i.e. only one page was there. In the second part the memory manager was made **'elastic'** i.e. the pages of the heap should be allocated on demand. In other words, the size of the heap should grow on demand.

2. ' alloc.c ' Implementation :

a. init_alloc() :

In `init_alloc()`, function call `mmap()` provides the virtual memory page of size 4KB (Heap size - 4KB). If `mmap()` fails it will throw error MAP-FAILED (-1) and on success it will assign the virtual address to pageAddress. Whereas `memset()` will reset all values to Zero in memoryBluePrint. memoryBluePrint is simple blueprint of heap indicating different memory allocations

```
int init_alloc()
{
    pageAddress = (char *)mmap(NULL, PAGESIZE, PROT_READ |
                    PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,0,0);

    if (pageAddress == MAP_FAILED)
        return -1;

    memset(memoryBluePrint, (long long)EMPTY,
                                        sizeof(memoryBluePrint));
    return 0;
}
```

b. alloc() : heuristic - FIRST FIT

In `alloc()` we first check whether the demanded memory size is a multiple of 8 bytes or not. If no, we return NULL. We maintain an array of available memory slots of size 8 bytes (memoryBluePrint). If the value at a position is 0 it means that the memory slot is available. Using this array we search for the first chunk of continuous memory which can satisfy the demand i.e. we implement the first fit heuristic. Then we store the address of the first block in this chunk at all the blocks in the chunk. This will help us to track how much memory was allocated using `alloc()` system call. We then return the address of the first block in the allocated chunk.

c. dealloc() :

In `dealloc()` we search over the entire memoryBluePrint array to find all the slots which have the content same as the address passed as argument to the `dealloc()` function. We simply change the content at those positions as 0 to indicate that they are now available.

```
void dealloc(char *chunkAddress)
{
    for (int i = (chunkAddress - pageAddress) / MINALLOC;
                                    i < BLOCKSPERPAGE; i++)
    {
        if (memoryBluePrint[i] == (long long)chunkAddress)
        {
            memoryBluePrint[i] = EMPTY;
        }
        else
        {
            break;
        }
    }
}
```

d. cleanup() :

```
int cleanup()
{
    int status = munmap(pageAddress, PAGESIZE);

    memset(memoryBluePrint, (long long)EMPTY,
                                        sizeof(memoryBluePrint));
    return status;
}
```

In `cleanup()`, we return the memory mapped page to the OS using the `munmap()` system call. Moreover, reinitialize all of the memoryBluePrint array to o to indicate that all the memory is available for allocation.

3. ‘ ealloc.c ’ Implementation :

a. init_alloc() :

Here the memoryBluePrint array has the same functionality as previously i.e. it acts as a simple blueprint of heap indicating different memory allocations. However, we maintain a separate array for each page. `memset()` will reset all values to Zero in memoryBluePrint.

```
void init_alloc()
{
    memset(memoryBluePrint, (long long)EMPTY,
                                        sizeof(memoryBluePrint));
}
```

b. alloc() : heuristic - FIRST FIT

The `alloc()` works similar to that described in the previous section. However, subtle differences are there such as whenever the first time memory is demanded we demand a page from the OS. Moreover, for every subsequent request we check whether the demand for the memory can be satisfied using pages using the `memoryBluePrint` array. If not, then we demand for a new page from the OS. Here also, we use the first fit heuristic to find the continuous chunk of memory which can satisfy the request.

c. dealloc() :

In `dealloc()` we search over the entire memoryBluePrint array (across all memory-pages used) to find all the slots which have the content same as the address passed as argument to the `dealloc()` function. We simply change the content at those positions as o to indicate that they are now available.

```
void dealloc(char *chunkAddress)
{
    bool found = false;
    for (int p = 0; p < PAGEUSED; p++)
    {
        for (int i = (chunkAddress-pageAddress[p])/MINALLOC;
                                        i < BLOCKSPERPAGE; i++)
        {
            if (memoryBluePrint[p][i] == (long
                                            long)chunkAddress)
            {
                memoryBluePrint[p][i] = EMPTY;
                found = true;
            }
        }

        if (found)
            break;
    }
}
```

d. cleanup() :

In `cleanup()`, we reinitialize all of the `memoryBluePrint` array to 0 to indicate that all the memory is available for allocation.

```
void init_alloc()
{
    memset(memoryBluePrint, (long long)EMPTY,
                                        sizeof(memoryBluePrint));
}
```