OS Lab
Lab 3 Report
Dynamic memory management

Vibhuti Raman 180010040
Vipul Nikam 180010041

Part A: Building a simple memory manager

In this part of the lab, we have written code for a memory manager, to allocate and deallocate memory dynamically. our memory manager manages 4KB of memory, by requesting a 4KB page via mmap from the OS. we must support allocations and deallocations in sizes that are multiples of 8 bytes
We have implemented four basic functions here:

- **init_alloc() ->** It initializes the memory manager, including allocating a 4KB page from the OS.

```
int init_alloc(){
        p = (char *) mmap(0, PAGESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

        if (p == (void *) -1){
                return -1;
        }
        for (int i = 0; i < arr_size; i++){
                starts[i] = 0;
                ends[i] = 0;
        }
        return 0;
}
```

- **cleanup() ->** it should clean up any state of your memory manager and memory is not returned back to OS.

```
int cleanup() {
        for (int i = 0; i < arr_size; i++){
                starts[i] = 0;
                ends[i] = 0;
        }
        return munmap(p, PAGESIZE);
}
```

- **alloc(int) ->** it takes an integer buffer size that must be allocated, and returns a char * pointer to the buffer on a success. This function returns a NULL on failure (e.g., requested size is not a multiple of 8 bytes, or insufficient free space). When successful, the returned pointer should point to a valid memory address within the 4KB page of the memory manager.

```
char *alloc(int size) {
        if (size % MINALLOC != 0){
                return NULL;
        }
        int size_units = size / MINALLOC;

        int state = 0;
        int probable_start = 0;
        int free_count = 0;
        for (int i = 0; i < arr_size; i++) {
                state += starts[i];

                if (state == 0) {
                        free_count += 1;
                } else{
                        probable_start = i+1;
                        free_count = 0;
                }

                if (free_count == size_units) {
                        starts[probable_start] = 1;
                        ends[i] = 1;
                        return p + probable_start*MINALLOC;
                }
                state -= ends[i];
        }
        return NULL;
```

- **dealloc(char *) ->** it takes a pointer to a previously allocated memory chunk, and frees up the entire chunk**.**

```
void dealloc(char * addr){

        char * temp = p;
        int found = 0;
        for (int i = 0; i < arr_size; i++,temp += MINALLOC) {
                if (temp == addr) {
                        starts[i] = 0;
                        found = 1;
                }

                if (found == 1 && ends[i] == 1){
                        ends[i] = 0;
                        return;
                }
        }
        return;
}
```

Output after running the test program:

```
raman@LAPTOP-H3SVD4FV: /mnt/f/6th semester/os-repeat/lab3/try this
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab3/try this$ g++ alloc.cpp test_alloc.c
raman@LAPTOP-H3SVD4FV:/mnt/f/6th semester/os-repeat/lab3/try this$ ./a.out
Hello, world! test passed
Elementary tests passed
Starting comprehensive tests (see details in code)
Test 1 passed: allocated 4 chunks of 1KB each
Test 2 passed: dealloc and realloc worked
Test 3 passed: dealloc and smaller realloc worked
Test 4 passed: merge worked
Test 5 passed: merge alloc 2048 worked
```

Part B: Expandable heap

In this part, we built a custom memory allocator over memory mapped pages, much like we did in the previous part of the lab. However, now our memory allocator is “elastic”, i.e., it memory maps pages from the OS only on demand,

We have implemented four basic functions:

- **init_alloc() ->** it should initialize our memory manager, However we have not memory mapped any pages from the OS yet, because we are supposed to allocate memory only on demand.

```
void init_alloc(){
        for (int i = 0; i < arr_size; i++){
                start1[i] = 0;
                start2[i] = 0;
                start3[i] = 0;
                start4[i] = 0;
                end1[i] = 0;
                end2[i] = 0;
                end3[i] = 0;
                end4[i] = 0;
        }
        return;
}
```

- **cleanup() ->** it should clean up any state of our memory manager.

```
void cleanup() {
        for (int i = 0; i < arr_size; i++){
                start1[i] = 0;
                start2[i] = 0;
                start3[i] = 0;
                start4[i] = 0;
                end1[i] = 0;
                end2[i] = 0;
                end3[i] = 0;
                end4[i] = 0;
        }
}
```

- alloc(int) -> it should take an integer buffer size that must be allocated, and must return a char * pointer to the buffer on a success. This function should return NULL on failure.we have made the following assumptions to simplify the problem. Buffer sizes requested are multiples of 256 bytes, and never longer than 4KB (page size). The total allocated memory will not exceed 4 pages, i.e16KB. we have assumed that every allocated chunk fully resides in one of the

4 pages.

```
char *alloc(int size) {
        if (size % MINALLOC != 0){
                return NULL;
        }
        int size_units = size / MINALLOC;

        if (p1 == NULL) {
                p1 = (char *) mmap(0, PAGESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

                if (p1 == (void *) -1){
                        return NULL;
                }
        }

        int state = 0;
        int probable_start = 0;
        int free_count = 0;
        for (int i = 0; i < arr_size; i++) {
                state += start1[i];

                if (state == 0) {
                        free_count += 1;
                } else{
                        probable_start = i+1;
                        free_count = 0;
                }

                if (free_count == size_units) {
                        start1[probable_start] = 1;
                        end1[i] = 1;
                        return p1 + probable_start*MINALLOC;
                }
                state -= end1[i];
        }

        if (p2 == NULL) {
                p2 = (char *) mmap(0, PAGESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

                if (p2 == (void *) -1){
                        return NULL;
                }
        }

        state = 0;
        probable_start = 0;
        free_count = 0;
        for (int i = 0; i < arr_size; i++) {
                state += start2[i];

                if (state == 0) {
                        free_count += 1;
                } else{
                        probable_start = i+1;
                        free_count = 0;
                }

                if (free_count == size_units) {
                        start2[probable_start] = 1;
                        end2[i] = 1;
                        return p2 + probable_start*MINALLOC;
                }
                state -= end2[i];
        }
```

```
    if (p3 == NULL) {
            p3 = (char *) mmap(0, PAGESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0

            if (p3 == (void *) -1){
                    return NULL;
            }
    }
    state = 0;
    probable_start = 0;
    free_count = 0;
    for (int i = 0; i < arr_size; i++) {
            state += start3[i];

            if (state == 0) {
                    free_count += 1;
            } else{
                    probable_start = i+1;
                    free_count = 0;
            }

            if (free_count == size_units) {
                    start3[probable_start] = 1;
                    end3[i] = 1;
                    return p3 + probable_start*MINALLOC;
            }
            state -= end3[i];
    }

    if (p4 == NULL) {
            p4 = (char *) mmap(0, PAGESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0

            if (p4 == (void *) -1){
                    return NULL;
            }
    }

    state = 0;
    probable_start = 0;
    free_count = 0;
    for (int i = 0; i < arr_size; i++) {
            state += start4[i];

            if (state == 0) {
                    free_count += 1;
            } else{
                    probable_start = i+1;
                    free_count = 0;
            }

            if (free_count == size_units) {
                    start4[probable_start] = 1;
                    end4[i] = 1;
                    return p4 + probable_start*MINALLOC;
            }
            state -= end4[i];
    }
    return NULL;
```

- dealloc (char *) -> it takes a pointer to a previously allocated memory chunk (that was returned by an earlier call to alloc), and frees up the entire chunk.

```
void dealloc(char * addr){

        if (p1 == NULL) {
                return;
        }
        char * temp = p1;
        int found = 0;
        for (int i = 0; i < arr_size; i++,temp += MINALLOC) {
                if (temp == addr) {
                        start1[i] = 0;
                        found = 1;
                }

                if (found == 1 && end1[i] == 1){
                        end1[i] = 0;
                        return;
                }
        }

        if (p2 == NULL) {
                return;
        }
        temp = p2;
        found = 0;
        for (int i = 0; i < arr_size; i++,temp += MINALLOC) {
                if (temp == addr) {
                        start2[i] = 0;
                        found = 1;
                }

                if (found == 1 && end2[i] == 1){
                        end2[i] = 0;
                        return;
                }
        }

        if (p3 == NULL) {
                return;
        }
        temp = p3;
        found = 0;
        for (int i = 0; i < arr_size; i++,temp += MINALLOC) {
                if (temp == addr) {
                        start3[i] = 0;
                        found = 1;
                }

                if (found == 1 && end3[i] == 1){
                        end3[i] = 0;
                        return;
                }
        }

        if (p4 == NULL) {
                return;
        }
        temp = p4;
        found = 0;
        for (int i = 0; i < arr_size; i++,temp += MINALLOC) {
                if (temp == addr) {
                        start4[i] = 0;
                        found = 1;
                }

                if (found == 1 && end4[i] == 1){
                        end4[i] = 0;
                        return;
                }
        }
        return;
}
```

Output after running the program for part B: