西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab | Report

Report Subject: OS Experiment - Lab 7

Student ID : 2019380141

Student Name : ABID ALI

Experiment Name : Memory Management

Objective:

- Learn to design and to implement a custom memory allocator:
  The operating system provides mechanisms for dynamic allocation in contiguous memory spaces. There are multiple criteria to assess how well these mechanisms work, including speed of execution and the minimization of external fragmentation (to that end, you learned of different policies for allocation decisions, such as first- fit, best-fit, and worst-fit). In this lab, you will design and implement a memory allocation system to work in user space and apply the concepts you have studied so far.
- In this lab, you will write your own dynamic memory allocator called allocator() that you

should be able to use in place of the standard malloc() utility.

Equipment:

VMware with Ubuntu Linux

Methodology:

2.1 Experiment 1: customed memory allocation

The implementation of a doubly-linked list abstract data type (ADT) in C is given to you. The solution that is being given to you has undergone a good amount of testing, but there are no guarantees that it is absolutely perfect. You may want to read the code to understand the implementation, you may want to test it further, or do anything else you find appropriate to develop the confidence you have in it.

Be sure to put the files you create for this lab in the appropriate directories in this source code tree! That is, new header files should go in include/, new source files should go in src/, etc. Note that you may want to modify the code of this doubly-linked list so as to have a little more information in the node structure than what is given to you in the original module. (See more about this point in the description of the deallocate function below.) If you modify the doubly-linked list code, you should want to modify the test suite in src/dlisttest.c and run these tests to ensure that the module runs correctly after your changes.

For this problem, you will create a custom memory allocator. Since your code will not be part of the operating system, it will execute in user mode. The general idea is that one who uses your allocator will include a header file which declares your functions and also link with the code that defines/implements them. A program using your allocator will have to get it initialized before any calls are made to use dynamically allocated memory; our standard prototype for the initialization is:

```
int allocator_init(size_t size);
```

This function will create and initialize two doubly-linked lists used for memory management: one which keeps track of the memory that is available (call it free_list) and another which keeps track of memory which is already in use (call it allocated_list). All the memory manipulated by your allocator will reside in the heap of the process that uses it. When allocator_init starts, it will call the standard malloc to request a contiguous space of size bytes from the underlying operating system. The pointer received from malloc will be used to initialize a single node in your free_list; your allocated_list must start out empty. If the malloc call does not succeed, this function must return -1, otherwise, it must return 0.

Ultimately, a custom memory allocator will need an API similar to what you have for malloc and free, functions that provide dynamic memory allocation in C. With that in mind, we define the API for a function to allocate memory and another to deallocate.

```
void *allocate (size_t size);
```

Equivalent to malloc. This function will traverse the free_list and find a contiguous chunk of memory that can be used to satisfy the request of an area of size bytes. If the caller makes a request for memory that is larger than what your allocator can honor, this function must return NULL. Your allocate function must be flexible enough to allow for different allocation policies, namely first-fit, best-fit, and worst-fit. You should probably create three functions, one for each of these policies, which are used internally by allocate and are not visible to the user of your custom memory allocator. Having those functions allows you to make easy modifications to the policy used by allocate. You will want to design your code so that it is easy run experiments with different allocation policies, so think carefully about how you will define your functions prototypes. The design of these functions' API and their implementation (obviously) is up to you.

```
int deallocate(void *ptr);
```

Equivalent to free. This function will use ptr to find the corresponding node in the allocated_list and free up that chunk of memory. If the caller attempts to deallocate a pointer that cannot be found in the allocated_list, this function must return -1, otherwise it must return 0. To make your development process easier, at first, you can simply move the deallocated memory from the allocated_list to the free_list. Note that just as the C library's free, deallocate does not ask you for the size of the memory you are returning to the system. Think about how you can make your custom allocator keep track of the size of each allocated chunk of memory — the cleanest solution might require you to change the code of the doubly-linked list given to you.

The entire code of your custom memory allocator will be encapsulated in two files:

```
allocator.h

allocator.c
```

The allocator.h file is shown below.

Run Terminal Help

C dlisttest.c C allocator.h ×

include > C allocator.h > ...

```c
#ifndef _ALLOCATOR_H_
#define _ALLOCATOR_H_

#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include "dnode.h"
/*
  initializes two dlists: free_list and allocated_list. allocated_list is
  initialized as empty. free_list is initialized with one dnode. For this dnode,
  the size is equal to the size passed, and the data pointer points to the start
  of allocated memory, which is allocated using malloc(size).
*/

int allocator_init(size_t size);

/*
  This is a simple redirecting function, where depending on the int policy
  passed, one of the three policies will be called.
*/
void *allocate(int policy, size_t size);

/*
  Iterates through the nodes in free_list, starting from the front of the list.
  A temporary node, curNode is declared, which is set equal to the iterator of
  free_list. A do-while loop is used to test if curNode is has enough memory
  to allocate. Each iteration of the loop sets the iterator to the next node.
  When a suitable node is found, a new node is added to the back of dlist, and
  curNode's size is decreased and it's data pointer is increased by size. If
  curNode's size is = 0 after this decrease in size, it is removed from
  free_list. If the list is empty or no node has enough memory to allocate,
  NULL is returned.
*/
void *firstFit(size_t size);

/*
  Iterates through the nodes in free_list, starting from the front of the list.
  A temporary node, curNode is declared, which is set equal to the iterator of
  free_list. Another temporary node, smallestNode is declared, which is by
  default set equal to the first node in free_list. A do-while loop is used to
  iterate through every node of free_list. If the current node has enough
  memory and is smaller than smallestNode, then smallestNode is set to the
  currentNode. After every node is checked, the loop terminates. smallestNode is
  added to the back of the allocated_list. If smallestNode's size = 0, then it
  is removed from free_list. Otherwise, the size and data pointer of
  smallestNode are adjusted accordingly. If free_list is empty or no node has
  enough memory to allocate, NULL is returned.
*/
void *bestFit(size_t size);

/*
  Iterates through the nodes in free_list, starting from the front of the list.
  A temporary node, curNode is declared, which is set equal to the iterator of
  free_list. Another temporary node, largestNode is declared, which is by
  default set equal to the first node in free_list. A do-while loop is used to
  iterate through every node of free_list. If the current node has enough memory
  After every node is checked, the loop terminates. largestNode is added to the
  back of allocated_list. If largestNode's size = 0, then it is removed from
  free_list. Otherwise, the size and data pointer of smallestNode are
  adjusted accordingly. If free_list is empty or no node has enough memory to
  allocate, NULL is returned.
*/
void *worstFit(size_t size);
```

```
*/
void *worstFit(size_t size);

/*
  Iterates through the nodes in allocated_list, starting from the front of the
  list. A temporary node, curNode is declared, which is set equal to the
  iterator of allocated_list. A boolean variable 'exists' is set to false as
  well. A while loop iterates through allocated_list. The loop will run as long
  as curNode is not NULL. If curNode's data address equals the address pointer
  passed, called ptr, then exits is set to true and the loop terminates. After
  the loop terminates, if exits is false, the function exits and returns -1.
  Otherwise, the node pointed to is removed from allocated_list and added to the
  back of free_list.
*/
int deallocate(void *ptr);

/*
  Simply put, this function prints both free_list and allocated_list. A
  temporary node, curNode is declared, and is set equal to the iterator of
  free_list. The iterator is set to the first node in the list. If curNode is
  NULL to start, then free_list is empty. Otherwise, a do-while loop is used
  to iterate through every node in free_list and print out the node's size
  and memory address that data points to. After free_list is printed, curNode
  is set equal to the iterator of allocated_list, and the same process to print
  free_list applies to allocated_list.

*/
void allocator_print();

/*
  Returns a node in allocated_list. Simply put, a helper function for deallocate.
  A temporary node, curNode is declared, and is set equal to the iterator of
  allocated_list. The iterator is set to the first node in allocated_list. If
  curNode is NULL right after it is set to the iterator, allocated_list is empty
  and NULL is returned. A counter for a while loop, count, is initialized to 1.
  A while loop iterates through allocated_list until the iterator is equal to the
  'ath' node of allocated_list. After the loop exits, the function returns the
  node that the iterator points to. All in all, this function returns the 'ath'
  node of allocated_list.
*/
void *getIterAllocatedList(int a);



#endif /* _ALLOCATOR_H_ */
```

Requirements：

1) You must include the **allocator.h** and implement **allocator.c** using first fit, best fit and worst fit algorithms respectively.

```c
#include "dlist.h"
#include "dnode.h"
#include "allocator.h"
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>

struct dlist *free_list;
struct dlist *allocated_list;



int allocator_init(size_t size){
    free_list = dlist_create();
    allocated_list = dlist_create();
    void *data;
    data = malloc(size);
    if(data == NULL){
        printf("ERROR WITH MALLOC()\n");
        return -1;
    }
    dlist_add_front(free_list, data, size); //added size
    return 0;
}


void *allocate(int policy, size_t size){
    void *ret;
    if(policy == 0){
        ret = firstFit(size);
    }
    else if(policy == 1){
        ret = bestFit(size);
    }
    else if(policy == 2){
        ret = worstFit(size);
```

Included the **allocator.h** and implement **allocator.c**

```c
void *firstFit(size_t size){

    dlist_iter_begin(free_list); // set iter to start of free_list
    struct dnode *curNode; // declare curNode as a dnode
    do{
        curNode = free_list->iter; // current node is iterator
        if(curNode == NULL){ // case free_list is empty
            break;
        }
        if(curNode->size > size){ // if size of node is large enough
            //printf("ADDRESS: %lld\n", (long long) curNode->data);
            dlist_add_back(allocated_list, curNode->data, size);
            curNode->size -= size;
            curNode->data += size;
            void *retVal;
            retVal = curNode->data;
            if (curNode->size <= 0){
                dlist_find_remove(free_list, retVal);
            }
            return retVal;
        }

    }while(dlist_iter_next(free_list) != NULL);
    printf("No node is large enough to store memory!\n");
    return NULL;


}
```

Fig:First Fit

```c
void *bestFit(size_t size){
    dlist_iter_begin(free_list); // set iter to start of free_list
    struct dnode *curNode; // declare curNode as a dnode
    struct dnode *smallestNode;

    curNode = free_list->iter;

    if (curNode == NULL){
        return NULL;
    }

    smallestNode = curNode; // set default smallest node to first node

    do{
        curNode = free_list->iter;
        //printf("curNode size: %d\n", curNode->size);
        //printf("size: %lld\n", (long long) size);
        //printf("smallNode size: %d\n\n", smallestNode->size);
        if(curNode->size >= size && curNode->size < smallestNode->size){
            //printf("SET NEW SMALL\n\n");
            smallestNode = curNode; // set smallestNode
        }

    }while(dlist_iter_next(free_list) != NULL);

    if(smallestNode->size < size){ // if no node is large enough
        printf("No node is large enough to store memory!\n");
        return NULL;
    }

    dlist_add_back(allocated_list, smallestNode->data, size);
    smallestNode->size -= size;
    smallestNode->data += size;
    void *retVal;
    retVal = smallestNode->data;
    //printf("SMALL SIZE: %d\n", smallestNode->size);
    if (smallestNode->size <= 0){ // remove node from free_list if size is 0
        dlist_find_remove(free_list, retVal);
    }
    return retVal;


}
```

Fig:Best Fit

```c
void *worstFit(size_t size){
    dlist_iter_begin(free_list); // set iter to start of free_list
    struct dnode *curNode; // declare curNode as a dnode
    struct dnode *largestNode;


    curNode = free_list->iter;

    if (curNode == NULL){
        return NULL;
    }

    largestNode = curNode; // set degault largest node to first node

    do{
        curNode = free_list->iter;
        if(curNode->size >= size && curNode->size > largestNode->size){
            largestNode = curNode; // set largestNode
        }

    }while(dlist_iter_next(free_list) != NULL);

    if(largestNode->size < size){ // if no node is large enough
        printf("No node is large enough to store memory!\n");
        return NULL;
    }

    dlist_add_back(allocated_list, largestNode->data, size);
    largestNode->size -= size;
    largestNode->data += size;
    void *retVal;
    retVal = largestNode->data;
    if (largestNode->size <= 0){
        dlist_find_remove(free_list, retVal);
    }
    return retVal;


}
```

Fig:Worst Fit

2) To test your implementation, you will need to create one or more test programs, each with their own main() function, to exercise your memory allocator. If you have a single file with all your tests, submit it as memory-test.c.

```c
#include "allocator.h"
#include "dnode.h"
#include "dlist.h"
#include <stdio.h>
#include <stdlib.h>
int main(){
    int init;
    size_t size;
    int sizes[5] = {100, 200, 250, 200, 100};
    int i;
    size = 2000;
    init = allocator_init(size);
    allocator_print();
    if(init == -1){
        exit(-1);
    }
    // test first-fit policy
    for(i = 0; i < 5; i++){
        allocate(0, sizes[i]);
    }
    allocator_print();
    //remove a node
    struct dnode *curNode;
    curNode = getIterAllocatedList(2);
    printf("Removed Node size = %d, points to %lld", curNode->size, (long long) curNode->data);
    deallocate(curNode->data);
    allocator_print();
    // allocate space for mem size 300; will result in two free nodes, since we use first fit
    /*
    allocate(0, 250);
    allocator_print();
    */
    //allocate space for mem size 250; will result in one free node, since we use best fit
    ///*
    allocate(1,250);
    allocator_print();
    //*/
    //attempt to allocate memory too large
    allocate(0,1300);

    //allocate more memory
    allocate(0, 300);
    allocate(0,320);
    allocate(0,340);
    allocator_print();

    //remove some data
    curNode = getIterAllocatedList(2);
    deallocate(curNode->data);
    curNode = getIterAllocatedList(1);
    deallocate(curNode->data);
    curNode = getIterAllocatedList(2);
    deallocate(curNode->data);
    allocator_print();
    //test first-fit - WORKS
    //allocate(0,20);
    //test best-fit - WORKS
    //allocate(1,20);
    //test worst-fit - WORKS
    allocate(2,20);
    allocator_print();
    // test if node is removed from allocated_list
    allocate(1,190);
    allocator_print();
```

Fig: Created memory-test.c

```
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./src/dlisttest.c -o ./bin/dlisttest
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/allocator.c -o ./obj/allocator.o
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./obj/allocator.o ./src/memory-test.c -o ./bin/memory-test
abid@ubuntu:~/Lab-7/try/1$ ls
bin  doc  include  Makefile  obj  src
abid@ubuntu:~/Lab-7/try/1$ cd bin/
abid@ubuntu:~/Lab-7/try/1/bin$ ls
dlisttest  memory-test
abid@ubuntu:~/Lab-7/try/1/bin$ ./memory-test

Free List:
Node size = 2000, points to 94668181766912
Allocated List is empty!

Free List:
Node size = 1150, points to 94668181767762

Allocated List:
Node size = 100, points to 94668181766912
Node size = 200, points to 94668181767012
Node size = 250, points to 94668181767212
Node size = 200, points to 94668181767462
Node size = 100, points to 94668181767662
Removed Node size = 250, points to 94668181767212
Free List:
Node size = 1150, points to 94668181767762
Node size = 250, points to 94668181767212

Allocated List:
Node size = 100, points to 94668181766912
Node size = 200, points to 94668181767012
Node size = 200, points to 94668181767462
Node size = 100, points to 94668181767662

Free List:
Node size = 1150, points to 94668181767762

Allocated List:
Node size = 100, points to 94668181766912
Node size = 200, points to 94668181767012
Node size = 200, points to 94668181767462
Node size = 100, points to 94668181767662
```

Fig: Execution process

```
abid@ubuntu:~/Lab-7/try/1/bin$ ./dlisttest
dlisttest running...

testing dlist_add_back
forward traversal
string = animal
string = barnacle
string = carnage
string = demented
string = error
backward traversal
string = error
string = demented
string = carnage
string = barnacle
string = animal

list destroyed
forward traversal
backward traversal

traversal of empty list completed

testing dlist_add_front
forward traversal
string = jelly
string = ignorant
string = hospital
string = gunk
string = folly
backward traversal
string = folly
string = gunk
string = hospital
string = ignorant
string = jelly

testing remove front
removed string = jelly
list length= 4

forward traversal
string = ignorant
string = hospital
string = gunk
string = folly
backward traversal
string = folly
string = gunk
string = hospital
string = ignorant

testing remove back
removed string = folly
list length= 3

forward traversal
string = ignorant
string = hospital
string = gunk
backward traversal
string = gunk
string = hospital
string = ignorant
```

```
testing find_remove
removed string = hospital
contents of the list
forward traversal
string = ignorant
string = gunk
backward traversal
string = gunk
string = ignorant
abid@ubuntu:~/Lab-7/try/1/bin$
```

Fig: Created by dlisttest.c

Test1

Unsorted Array

```c
int init;
size_t size;
int sizes[5] = {100, 200, 250, 200, 100};
int i;
size = 2000;
init = allocator_init(size);
allocator_print();
if(init == -1){
    exit(-1);
}
```

When we give unsorted value with few similar elements in the array.

```
abid@ubuntu:~/Final Lab-7/1$ make clean
/bin/rm -rf ./bin/* ./obj/* core* *~
abid@ubuntu:~/Final Lab-7/1$ make all
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/dnode.c -o ./obj/dnode.o
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/dlist.c -o ./obj/dlist.o
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./src/dlisttest.c -o ./bin/dlisttest
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/allocator.c -o ./obj/allocator.o
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./obj/allocator.o ./src/memory-test.c -o ./bin/memory-test
abid@ubuntu:~/Final Lab-7/1$ cd bin
abid@ubuntu:~/Final Lab-7/1/bin$ ls
dlisttest  memory-test
abid@ubuntu:~/Final Lab-7/1/bin$ ./memory-test

Free List:
Node size = 2000, points to 94100433949440
Allocated List is empty!

Free List:
Node size = 1150, points to 94100433950290

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 250, points to 94100433949740
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190
Removed Node size = 250, points to 94100433949740
Free List:
Node size = 1150, points to 94100433950290
Node size = 250, points to 94100433949740

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190

Free List:
Node size = 900, points to 94100433950540
Node size = 250, points to 94100433949740

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190
Node size = 250, points to 94100433950290

Free List:
Node size = 900, points to 94100433950540

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190
Node size = 250, points to 94100433950290
Node size = 250, points to 94100433949740
No node is large enough to store memory!
No node is large enough to store memory!

Free List:
Node size = 280, points to 94100433951160

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190
Node size = 250, points to 94100433950290
Node size = 250, points to 94100433949740
Node size = 300, points to 94100433950540
Node size = 320, points to 94100433950840
```

We can see that,we have fixed to 2000 .Then,we notice ,how the values are distributed from Free List to Allocated List.

Node Size =2000

Allocated list was empty at that time.

Then,in next step values are allocated based on different policies.

There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes

```
Free List:
Node size = 1150, points to 94100433950290

Allocated List:
Node size = 100, points to 94100433949440
Node size = 200, points to 94100433949540
Node size = 250, points to 94100433949740
Node size = 200, points to 94100433949990
Node size = 100, points to 94100433950190
Removed Node size = 250, points to 94100433949740
Free List:
Node size = 1150, points to 94100433950290
Node size = 250, points to 94100433949740
```

In this way,the process keep continuing.

- First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Test2

Sorted Array

```
int init;
size_t size;
int sizes[5] = {100, 200, 250, 300, 400};
int i;
size = 2000;
init = allocator_init(size);
allocator_print();
if(init == -1){
    exit(-1);
}
// test first-fit policy
for(i = 0; i < 5; i++){
    allocate(0, sizes[i]);
}
```

```
abid@ubuntu:~/Final Lab-7/1/bin$ ls
dlisttest  memory-test
abid@ubuntu:~/Final Lab-7/1/bin$ ./memory-test

Free List:
Node size = 2000, points to 94680211256064
Allocated List is empty!

Free List:
Node size = 750, points to 94680211257314

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Removed Node size = 250, points to 94680211256364
Free List:
Node size = 750, points to 94680211257314
Node size = 250, points to 94680211256364

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914

Free List:
Node size = 500, points to 94680211257564
Node size = 250, points to 94680211256364

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314

Free List:
Node size = 500, points to 94680211257564

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314
Node size = 250, points to 94680211256364
No node is large enough to store memory!
No node is large enough to store memory!
No node is large enough to store memory!

Free List:
Node size = 200, points to 94680211257864

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211257564
```

```
Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314

Free List:
Node size = 500, points to 94680211257564

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314
Node size = 250, points to 94680211256364
No node is large enough to store memory!
No node is large enough to store memory!
No node is large enough to store memory!

Free List:
Node size = 200, points to 94680211257864

Allocated List:
Node size = 100, points to 94680211256064
Node size = 200, points to 94680211256164
Node size = 300, points to 94680211256614
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211257314
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211257564

Free List:
Node size = 200, points to 94680211257864
Node size = 300, points to 94680211256614
Node size = 200, points to 94680211256164
Node size = 250, points to 94680211257314

Allocated List:
Node size = 100, points to 94680211256064
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211257564

Free List:
Node size = 160, points to 94680211257904
Node size = 280, points to 94680211256634
Node size = 200, points to 94680211256164
Node size = 250, points to 94680211257314

Allocated List:
Node size = 100, points to 94680211256064
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211257564
Node size = 20, points to 94680211257864
Node size = 20, points to 94680211257884
Node size = 20, points to 94680211256614
No node is large enough to store memory!

Free List:
Node size = 160, points to 94680211257904
Node size = 280, points to 94680211256634
Node size = 200, points to 94680211256164
Node size = 250, points to 94680211257314

Allocated List:
Node size = 100, points to 94680211256064
Node size = 400, points to 94680211256914
Node size = 250, points to 94680211256364
Node size = 300, points to 94680211257564
Node size = 20, points to 94680211257864
Node size = 20, points to 94680211257884
Node size = 20, points to 94680211256614
```

2.2 Experiment 2: observation the fragmentation

Create a new function in your allocator to compute the average fragmentation created in memory after repeated, randomly interleaved calls to your allocate and deallocate functions. Before you get down to coding this function, though, think about how you will implement it and write here your algorithm in the form of an algorithm in pseudo-C code. The signature for this function should be something like:

double average_frag();

```c
double average_frag(){
    //void *curNode;
    double totalSize = 0;
    int totalNodes = 0;
    /*curNode = */dlist_iter_begin(free_list); // set curNode to first node in free_list
    while(free_list->iter != NULL){
        totalSize += free_list->iter->size;
        totalNodes += 1;
        dlist_iter_next(free_list);
    }
    if(totalNodes == 0){
        printf("free_list is empty!\n");
        return 0.0;
    }
    printf("total size: %lf\n", totalSize);
    printf("total nodes: %d\n", totalNodes);
    return totalSize/totalNodes;
}
```

You can get the average fragmentation using below:

average_frag = totalFreeSpace / the # holes;

Finally, once you have the algorithm for average_frag, implement the corresponding function in the C module that contains your allocator. That is, you will be changing your **allocator.h** and **allocator.c** files to include the new function.

You need to implement frag-eval.c. Modify your new file so that its main function accepts command line parameters as in the usage guide below:

frag-eval [algorithm] [requests]

```
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 2 10 10
```

where:

- algorithm (integer) can take only the tree values which select different allocation policies:0 (means first-fit), 1 (means best-fit), 2 (means worst-fit)
- requests (integer) specifies the number of dynamic memory allocation requests to simulate before the evaluation of fragmentation is performed.

Your main function must implement some pattern of allocate and deallocate requests to exercise the functionality of your allocator. Consider the algorithm given below in pseudo-C code:

```c
srand(seed);
count = 0;
while(count < requests){
    ran = rand();
    ran = 100 + 9900*((double)ran/RAND_MAX); // generate random number between 100 & 10000
    //printf("Random Memory: %d\n", ran);
    data = allocate(algorithm, ran);
    deallocate(data);
    //allocator_print();
    count++;
}
allocator_print();
```

Terminal panel

```
abid@ubuntu:~/Final Lab-7/2$ make all
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/dnode.c -o ./obj/dnode.o
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/dlist.c -o ./obj/dlist.o
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./src/dlisttest.c -o ./bin/dlisttest
gcc -I ./include -std=gnu99 -Wall -g  -c ./src/allocator.c -o ./obj/allocator.o
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./obj/allocator.o ./src/memory-test.c -o ./bin/memory-test
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./obj/allocator.o ./src/frag-eval.c -o ./bin/frag-eval
abid@ubuntu:~/Final Lab-7/2$ cd bin
abid@ubuntu:~/Final Lab-7/2/bin$ ls
dlisttest  frag-eval  memory-test
```

Requirements:

1) Choose a value of R of runs to perform and also choose R different seeds to provide for the generation of pseudo-random numbers. (Consider only values of R >= 30)

```
bid@ubuntu:~/Final Lab-7/2$ cd bin
bid@ubuntu:~/Final Lab-7/2/bin$ ls
listtest  frag-eval  memory-test
bid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
nter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 0 30 30
```

Fig: First Fit

```
gcc -I ./include -std=gnu99 -Wall -g  ./obj/dnode.o ./obj/dlist.o ./obj/allocator.o ./src/frag-eval.c -o ./bin/frag-eval
abid@ubuntu:~/Final Lab-7/2$ cd bin
abid@ubuntu:~/Final Lab-7/2/bin$ ls
dlisttest  frag-eval  memory-test
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 1  30 30
```

Fig: Best Fit

```
abid@ubuntu:~/Final Lab-7/2$ cd bin
abid@ubuntu:~/Final Lab-7/2/bin$ ls
dlisttest  frag-eval  memory-test
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 2 30 30
```

Fig: Worst Fit

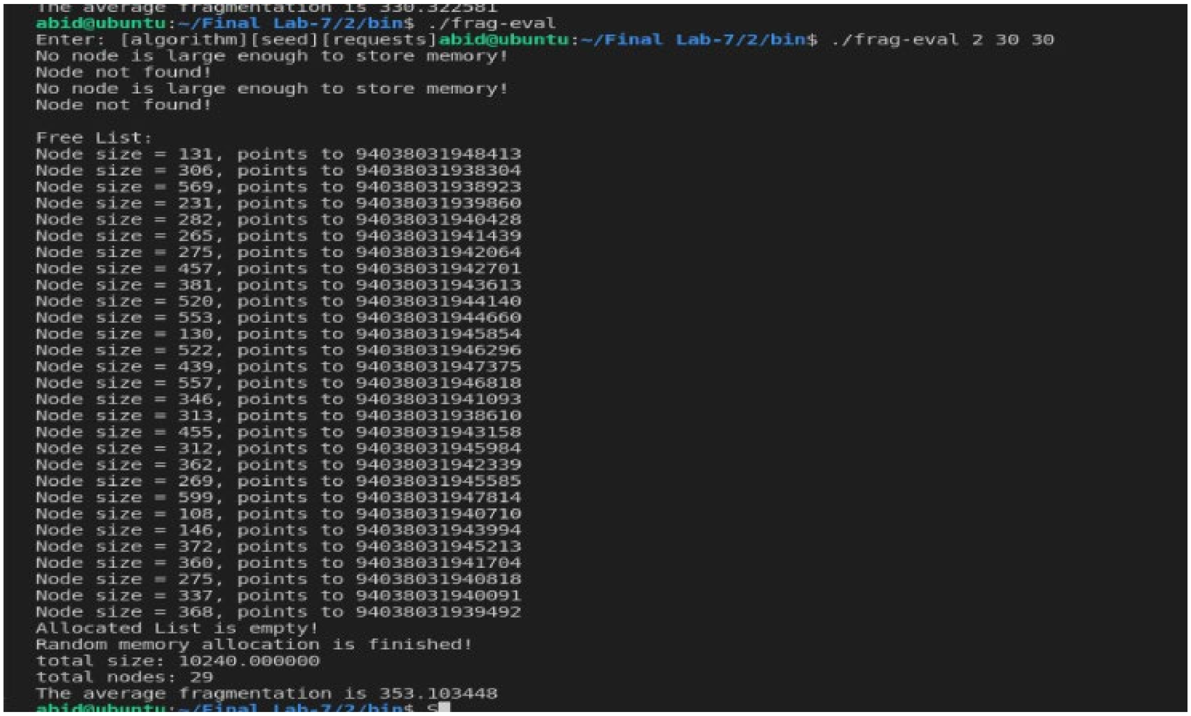
2) For each of the three allocation policies (first-fit, best-fit, and worst-fit), complete R runs using the selected seeds and compare the total fragmentation and the average fragmentation.

```
abid@ubuntu:~/Final Lab-7/2$ cd bin
abid@ubuntu:~/Final Lab-7/2/bin$ ls
dlisttest  frag-eval  memory-test
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 0 30 30

Free List:
Node size = 65, points to 94143188081343
Node size = 160, points to 94143188071314
Node size = 153, points to 94143188072203
Node size = 11, points to 94143188072944
Node size = 164, points to 94143188073410
Node size = 112, points to 94143188074456
Node size = 323, points to 94143188074880
Node size = 85, points to 94143188075937
Node size = 195, points to 94143188076663
Node size = 67, points to 94143188077457
Node size = 193, points to 94143188077884
Node size = 66, points to 94143188078782
Node size = 132, points to 94143188079550
Node size = 996, points to 94143188079682
Node size = 557, points to 94143188080678
Node size = 729, points to 94143188071474
Node size = 313, points to 94143188072356
Node size = 455, points to 94143188072955
Node size = 882, points to 94143188073574
Node size = 312, points to 94143188074568
Node size = 362, points to 94143188075203
Node size = 641, points to 94143188076022
Node size = 599, points to 94143188076858
Node size = 108, points to 94143188081235
Node size = 146, points to 94143188071168
Node size = 372, points to 94143188075565
Node size = 360, points to 94143188077524
Node size = 275, points to 94143188072669
Node size = 337, points to 94143188078077
Node size = 702, points to 94143188078848
Node size = 368, points to 94143188078414
Allocated List is empty!
Random memory allocation is finished!
total size: 10240.000000
total nodes: 31
The average fragmentation is 330.322581
abid@ubuntu:~/Final Lab-7/2/bin$
```

Fig: First Fit

```
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 1 30 30

Free List:
Node size = 3954, points to 94767099620238
Node size = 306, points to 94767099613952
Node size = 8, points to 94767099615132
Node size = 46, points to 94767099614811
Node size = 62, points to 94767099615697
Node size = 46, points to 94767099616707
Node size = 36, points to 94767099616358
Node size = 7, points to 94767099617565
Node size = 65, points to 94767099618343
Node size = 25, points to 94767099617394
Node size = 98, points to 94767099614713
Node size = 42, points to 94767099618301
Node size = 834, points to 94767099618408
Node size = 6, points to 94767099620232
Node size = 185, points to 94767099615512
Node size = 27, points to 94767099618274
Node size = 1, points to 94767099616706
Node size = 93, points to 94767099614620
Node size = 882, points to 94767099619242
Node size = 312, points to 94767099616394
Node size = 2, points to 94767099614618
Node size = 641, points to 94767099616753
Node size = 599, points to 94767099615759
Node size = 108, points to 94767099620124
Node size = 146, points to 94767099617419
Node size = 4, points to 94767099615508
Node size = 23, points to 94767099614595
Node size = 275, points to 94767099614857
Node size = 337, points to 94767099614258
Node size = 702, points to 94767099617572
Node size = 368, points to 94767099615140
Allocated List is empty!
Random memory allocation is finished!
total size: 10240.000000
total nodes: 31
The average fragmentation is 330.322581
```

Fig: Best Fit



```
The average fragmentation is 330.322581
abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval
Enter: [algorithm][seed][requests]abid@ubuntu:~/Final Lab-7/2/bin$ ./frag-eval 2 30 30
No node is large enough to store memory!
Node not found!
No node is large enough to store memory!
Node not found!

Free List:
Node size = 131, points to 94038031948413
Node size = 306, points to 94038031938304
Node size = 569, points to 94038031938923
Node size = 231, points to 94038031939860
Node size = 282, points to 94038031940428
Node size = 265, points to 94038031941439
Node size = 275, points to 94038031942064
Node size = 457, points to 94038031942701
Node size = 381, points to 94038031943613
Node size = 520, points to 94038031944140
Node size = 553, points to 94038031944660
Node size = 130, points to 94038031945854
Node size = 522, points to 94038031946296
Node size = 439, points to 94038031947375
Node size = 557, points to 94038031946818
Node size = 346, points to 94038031941093
Node size = 313, points to 94038031938610
Node size = 455, points to 94038031943158
Node size = 312, points to 94038031945984
Node size = 362, points to 94038031942339
Node size = 269, points to 94038031945585
Node size = 599, points to 94038031947814
Node size = 108, points to 94038031940710
Node size = 146, points to 94038031943994
Node size = 372, points to 94038031945213
Node size = 360, points to 94038031941704
Node size = 275, points to 94038031940818
Node size = 337, points to 94038031940091
Node size = 368, points to 94038031939492
Allocated List is empty!
Random memory allocation is finished!
total size: 10240.000000
total nodes: 29
The average fragmentation is 353.103448
abid@ubuntu:~/Final Lab-7/2/bin$
```

Fig: Worst Fit

Solution:

To solve those problems I looked for information in internet. In order to understand some questions and procedure I also asked the teacher to help me understand them. And provided instructions helped to solve some of my errors during the experiment.

Problems:

The problem that I faced during was how to use different kinds of algorithms and implementing those algorithms.I was having problem to understand the question at the beginning .

Conclusion:

At the beginning, I was unfamiliar with those algorithm and how to make simulation. Gradually, reading lot of article and reading teachers ppt then I solved those problem one by one. In this experiment ,small helps and suggestions from the teacher was very helpful that saved my time.I enjoyed the practical and learned lot of interesting things.

Attachments:

1) ABID ALI_2019380141_OS(Lab 7).docx

2) ABID ALI_2019380141_OS(Lab 7).pdf
3)Code_ABID ALI_2019380141(Lab-7).zip