CMP-5013A Coursework Assignment 2

zmf18gwu (100237137)
vfs18heu (100263300)

Fri, 6 Dec 2019 15:17

PDF prepared using PASS version 1.17 running on `Windows 10 10.0` (`amd64`).

☑ I agree that by submitting a PDF generated by PASS I am confirming that I have checked the PDF and that it correctly represents my submission.

Contents

part1.h

```

#ifndef CW2_PART1_H
#define CW2_PART1_H

typedef struct NodeStruct{

    int free;
    size_t size;
    void *memory;
    struct NodeStruct* prevNode;
    struct NodeStruct* nextNode;

}Node;


void *allocate(size_t);
void deallocate ( void * memory );
void initialise ( void * memory , size_t size);


#endif //CW2_PART1_H
```

part1.c

```
/* -------------------------------------------------------------------------------------------
/
    /
/   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory Manager.              /
/
    /
/                                                  part1                                     /
/
    /
/   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.                        /
/
    /
/   Last edit date: 06/12/19                                                                    /
/
    /
/   Description: Program to simulate a thread-safe memory manager using a linked list                   /
/                   implementation.                                                             /
/
    /
  -------------------------------------------------------------------------------------------
    */


#include <stdio.h>
#include <stdlib.h>
#include "part1.h"



//global variables.
Node *HEAD = NULL;

/*
 * Inputs: Node* nodePointer, size_t bytes.
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a struct.
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
```

```c
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *allocate(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated.\n",bytes);

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("found node of exact size.\n");

            //Set free to 0 as node is taken.
            nodePointer->free = 0;
            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the initial HEAD node.
*/
void initialise(void *memory ,size_t size){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);

    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory address.
 */
void deallocate ( void * memory ){

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;

    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful\n");
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size + sizeof(Node)+connectedHolesPointer->nextNode->size;


                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }

    }

}
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                       point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                       point->memory);
            //look at next node.
```

```c
            point = point->nextNode;
        }

    }
}
```

5013ACW02-100237137-file.c

Filename scrubbed (one or more forbidden characters found). Original name:

`part1_test.c`

```
/*
   ----------------------------------------------------------------------------------------

/

   /
/   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory
   Manager.             /
/

   /
/                                           part1
                                                                /
/

   /
/   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.
                                    /
/

   /
/   Last edit date: 06/12/19
                                                                                  /
/

   /
/   Description: Program to simulate a thread-safe memory manager using a linked
   list                   /
/                 implementation.
                                                                              /
/

   /
 ----------------------------------------------------------------------------------------
    */


#include <stdio.h>
#include <stdlib.h>
#include "part1.h"



//global variables.
Node *HEAD = NULL;

/*
 * Inputs: Node* nodePointer, size_t bytes.
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free
     node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating
       node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a
        struct.
```

```c
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *allocate(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated.\n",bytes);

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("found node of exact size.\n");

            //Set free to 0 as node is taken.
            nodePointer->free = 0;

            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the initial HEAD node.
*/
void initialise(void *memory ,size_t size){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);

    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory address.
 */
void deallocate ( void * memory ){

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;


    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful\n");
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;



                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }

    }

}
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                       point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
```

```c
                    (int)point->size,point->free,(point->memory-sizeof(Node)),point->memory);
            //look at next node.
            point = point->nextNode;
        }

    }
}


int main() {

    //TEST HARNESS FOR EACH ALGORITHM COMMENTED OUT.

    //Set the initial heap size.
    size_t size = 1024;

    //Allocate memory for the heap.
    void *memory = malloc(size);

    //Algorithm test harness
    initialise(memory,size);

    void *x = allocate(100);
    printf("allocate x output: %d\n===============================\n\nVisual Node output:\n\n",(char*)x);
    output();

    void *y = allocate(50);
    printf("allocate y output: %d\n===============================\n\nVisual Node output:\n\n",(char*)y);
    output();

    void *z = allocate(100);
    printf("allocate z output: %d\n===============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    void *t = allocate(100);
    printf("allocate t output: %d\n===============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    deallocate(x);
    deallocate(z);

    printf("De-allocate x & z: \n===============================\n\nVisual Node output:\n\n");
    output();

    void *test1 = allocate(70);
    printf("test allocation of 70: \n===============================\n\nVisual Node output:\n\n");
    output();

    printf("Allocate all nodes.\n");
    void *test2 = allocate(10);
    void *test3 = allocate(100);
    void *test4 = allocate(574);
    output();

    printf("Try to allocate when all nodes are not free.\n");
    void *test5 = allocate(50);

    //Test coalace.
    printf("Test coalace, free test4 and t.\n");
    deallocate(test4);
    deallocate(t);
    output();

    //test invalid values.
    void *test6 = allocate(-5);

    free(memory);
    return EXIT_SUCCESS;

}
```

part2.h

```

#ifndef CW2_PART2_H
#define CW2_PART2_H

typedef struct NodeStruct{

    int free;
    size_t size;
    void *memory;
    struct NodeStruct* prevNode;
    struct NodeStruct* nextNode;

}Node;


void*(*allocate)(size_t);
void deallocate ( void * memory );
void initialise ( void * memory , size_t size, char* algorithm);

void *firstFit(size_t bytes);
void *nextFit(size_t bytes);
void *bestFit(size_t bytes);
void *worstFit(size_t bytes);

#endif //CW2_PART2_H
```

part2.c

```
/*
   ---------------------------------------------------------------------------------

/


   /
/   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory
   Manager.              /
/


   /
/                                        part2
                                                   /
/


   /
/   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.
                                  /
/


   /
/   Last edit date: 06/12/19
                                                                      /
/


   /
/   Description: Program to simulate a thread-safe memory manager using a linked
   list                 /
/                  implementation.
                                                                   /
/


   /
 -----------------------------------------------------------------------------------
    */


#include <stdio.h>
#include <stdlib.h>
#include "part2.h"
#include <string.h>


//global variables.
Node *HEAD = NULL;
Node *NEXTFITNODE = NULL;


//Function Set as a function pointer so that the different algorithms can be
   applied.
/*
 * Inputs: size_t variable.
 * Outputs: NONE.
 * Description: function pointer to other algorithm allocate functions.
 */
void*(*allocate)(size_t);

/*
 * Inputs: Node* nodePointer, size_t bytes.
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free
```

```
      node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating
       node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a
        struct.
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *firstFit(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }


    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated.\n",bytes);
```

```c

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("found node of exact size.\n");

            //Set free to 0 as node is taken.
            nodePointer->free = 0;

            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the NextFit algorithm to allocate memory from the heap.
 */
void* nextFit(size_t bytes){

    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial start location to the NEXTFITNODE node.
    Node *nodePointer = NEXTFITNODE;

    //Set the end node to the start node, therefore it will only loop once and not infinitely.
    Node *limitNode = NEXTFITNODE;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node *returnNode = allocateNodeWithHole(nodePointer,bytes);

            //update NEXTFITNODE.
            NEXTFITNODE = returnNode->nextNode;

            //return the allocated nodes memory address.
            return nodePointer->memory;
        }

            //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            //Set free to 0 as node is taken.
            printf("Found node of exact size.\n");
            nodePointer->free = 0;

            //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
                NEXTFITNODE = nodePointer->nextNode;
            }
            else{
                NEXTFITNODE = HEAD;
            }

            //return the allocated nodes memory address.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {

            //Look at next node.
            if (nodePointer->nextNode == NULL) {

                //loop through whole list
                nodePointer = HEAD;
            }
            else{
                nodePointer = nodePointer->nextNode;
            }
            if(nodePointer->memory == limitNode->memory){

                printf("!!!! No free nodes, returning NULL !!!!\n");

                //update NEXTFITNODE.
                NEXTFITNODE= HEAD;

                //Return NULL as no valid nodes found.
                return  NULL;
            }


        }
    }
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the BestFit algorithm to allocate memory from the heap.
 */
```

```
void* bestFit(size_t bytes){
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the best node that fits.
    int found = 0;
    int bestFit = 1;
    Node *bestNode = NULL;
    Node *nodePointer = HEAD;

    while(bestFit){
        if(nodePointer->free){

            //IF the node is of exact size.
            if (nodePointer->size == bytes){
                bestNode = nodePointer;
                found = 1;
                break;
            }

            //IF the node is greater than bytes
            if(nodePointer->size > (bytes + (sizeof(Node)))){
                found = 1;
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                if(nodePointer->size < bestNode->size){
                    bestNode = nodePointer;
                }

            }
            //If there is enough size to create a new node but not assign a size
                > 0 to it.
            if(nodePointer->size == (bytes +(sizeof(Node)))){
                printf("Invalid node to use. searching next\n");

            }
            //If there is a next node.
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                break;

            }
        }
        //Look at next node.
        else{
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Return NULL as no valid nodes found.
                return NULL;
```

```c
            }
        }


    }

    //Allocate the best node found with the desired bytes.
    if(found) {
        //If the node is free and is large enough for the allocated bytes.
        if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(bestNode,bytes);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }


        //if the node is the exact size.
        else if (bestNode->free == 1 && bestNode->size == bytes) {

            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            bestNode->free = 0;

            //return the allocated nodes memory address.
            return bestNode->memory;
        }

    }
    //If no suitable nodes found, return NULL.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");
        //Return NULL as no valid nodes found.
        return NULL;

    }



}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the WorstFit algorithm to allocate memory from the heap.
 */
void* worstFit(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }


    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the worst node that fits.
    int found = 0;
    int worstFit = 1;
    Node *worstNode = NULL;
    Node *nodePointer = HEAD;
    while (worstFit) {
        if (nodePointer->free) {

            //IF the node is of exact size.
            if (nodePointer->size == bytes) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                } else if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            }

                //IF the node is greater than bytes
            else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                }
                if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                printf("Invalid node to use. searching next\n");

            }

        }
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            } else {
                break;
            }

    }

    if(found) {

        //If the node is free and is large enough for the allocated bytes.
        if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(worstNode,bytes);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }



        //if the node is the exact size.
        else if (worstNode->free == 1 && worstNode->size == bytes) {
            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            worstNode->free = 0;

            //return the allocated nodes memory address.
            return worstNode->memory;

        }
```

```c

    }
    //If not found.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");

        return NULL;

    }
}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the initial HEAD node.
*/
void initialise(void *memory ,size_t size, char* algorithm){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);


    //Assign type of algorithm

    if(strcmp(algorithm,"NextFit")==0){
        printf("Using NextFit algorithm.\n");
        allocate = nextFit;
        NEXTFITNODE = HEAD;

    }
    else if(strcmp(algorithm, "BestFit")==0){
        printf("Using BestFit algorithm.\n");
        allocate = bestFit;

    }
    else if(strcmp(algorithm,"WorstFit")==0){
        printf("Using WorstFit algorithm.\n");
        allocate = worstFit;

    }
    //Default first fit
    else{
        printf("Using FirstFit algorithm.\n");
        allocate = firstFit;

    }


    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;
```

```c

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory address.
 */
void deallocate ( void * memory ){

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;


    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful\n");
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;

                //If NextFit algorithm is set, handle the pointer to it.
                if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
                        printf("Moving NEXTFITNODE due to coalace taking place.\n
                           ");
                        NEXTFITNODE = connectedHolesPointer;
                    }
                 }

                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }

    }

}
```

```c
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)), point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)), point->memory);
            //look at next node.
            point = point->nextNode;
        }

    }
}
```

5013ACW02-100237137-file1.c

Filename scrubbed (one or more forbidden characters found). Original name:

`part2_test.c`

```
/*  -----------------------------------------------------------------------------------------
/                                                                                            /
/   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory Manager.     /
/                                                                                            /
/                                                  part2 TEST HARNESS                        /
/                                                                                            /
/   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.                         /
/                                                                                            /
/   Last edit date: 06/12/19                                                                 /
/                                                                                            /
/   Description: Program to simulate a thread-safe memory manager using a linked list        /
/                    implementation.                                                         /
/                                                                                            /
 ----------------------------------------------------------------------------------------- */


#include <stdio.h>
#include <stdlib.h>
#include "part2.h"
#include <string.h>


//global variables.
Node *HEAD = NULL;
Node *NEXTFITNODE = NULL;


//Function Set as a function pointer so that the different algorithms can be applied.
/*
 * Inputs: size_t variable.
 * Outputs: NONE.
 * Description: function pointer to other algorithm allocate functions.
 */
void*(*allocate)(size_t);

/*
```

```
 * Inputs: Node* nodePointer, size_t bytes.
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free
     node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating
       node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a
        struct.
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *firstFit(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }


    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated.\n",bytes);

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("found node of exact size.\n");

            //Set free to 0 as node is taken.
            nodePointer->free = 0;

            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the NextFit algorithm to allocate memory from the heap.
 */
void* nextFit(size_t bytes){

    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial start location to the NEXTFITNODE node.
    Node *nodePointer = NEXTFITNODE;

    //Set the end node to the start node, therefore it will only loop once and not infinitely.
    Node *limitNode = NEXTFITNODE;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
```

```c
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node *returnNode = allocateNodeWithHole(nodePointer,bytes);

            //update NEXTFITNODE.
            NEXTFITNODE = returnNode->nextNode;

            //return the allocated nodes memory address.
            return nodePointer->memory;
        }

            //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            //Set free to 0 as node is taken.
            printf("Found node of exact size.\n");
            nodePointer->free = 0;

            //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
                NEXTFITNODE = nodePointer->nextNode;
            }
            else{
                NEXTFITNODE = HEAD;
            }

            //return the allocated nodes memory address.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {

            //Look at next node.
            if (nodePointer->nextNode == NULL) {

                //loop through whole list
                nodePointer = HEAD;
            }
            else{
                nodePointer = nodePointer->nextNode;
            }
            if(nodePointer->memory == limitNode->memory){

                printf("!!!! No free nodes, returning NULL !!!!\n");

                //update NEXTFITNODE.
                NEXTFITNODE= HEAD;

                //Return NULL as no valid nodes found.
                return  NULL;
            }


        }
    }
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the BestFit algorithm to allocate memory from the heap.
 */
void* bestFit(size_t bytes){
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the best node that fits.
    int found = 0;
    int bestFit = 1;
    Node *bestNode = NULL;
    Node *nodePointer = HEAD;

    while(bestFit){
        if(nodePointer->free){

            //IF the node is of exact size.
            if (nodePointer->size == bytes){
                bestNode = nodePointer;
                found = 1;
                break;
            }

            //IF the node is greater than bytes
            if(nodePointer->size > (bytes + (sizeof(Node)))){
                found = 1;
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                if(nodePointer->size < bestNode->size){
                    bestNode = nodePointer;
                }

            }
            //If there is enough size to create a new node but not assign a size
               > 0 to it.
            if(nodePointer->size == (bytes +(sizeof(Node)))){
                printf("Invalid node to use. searching next\n");

            }
            //If there is a next node.
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                break;

            }
        }
        //Look at next node.
        else{
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                printf("!!!! No free nodes, returning NULL !!!!\n");
                //Return NULL as no valid nodes found.
                return NULL;
            }
        }


    }

    //Allocate the best node found with the desired bytes.
    if(found) {
        //If the node is free and is large enough for the allocated bytes.
        if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(bestNode,bytes);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }


        //if the node is the exact size.
        else if (bestNode->free == 1 && bestNode->size == bytes) {

            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            bestNode->free = 0;

            //return the allocated nodes memory address.
            return bestNode->memory;
        }

    }
    //If no suitable nodes found, return NULL.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");
        //Return NULL as no valid nodes found.
        return NULL;

    }



}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the WorstFit algorithm to allocate memory from the heap.
 */
void* worstFit(size_t bytes) {
    //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
        printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
    }


    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the worst node that fits.
    int found = 0;
    int worstFit = 1;
    Node *worstNode = NULL;
    Node *nodePointer = HEAD;

    while (worstFit) {
        if (nodePointer->free) {

            //IF the node is of exact size.
            if (nodePointer->size == bytes) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                } else if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            }

                //IF the node is greater than bytes
            else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                }
                if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                printf("Invalid node to use. searching next\n");

            }

        }
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            } else {
                break;
            }

    }

    if(found) {

        //If the node is free and is large enough for the allocated bytes.
        if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(worstNode,bytes);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }



        //if the node is the exact size.
        else if (worstNode->free == 1 && worstNode->size == bytes) {
            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            worstNode->free = 0;

            //return the allocated nodes memory address.
```

```c
                return worstNode->memory;

            }

        }
        //If not found.
        else {
            printf("!!!! No free nodes, returning NULL !!!!\n");

            return NULL;

        }
}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the
   initial HEAD node.
*/
void initialise(void *memory ,size_t size, char* algorithm){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);


    //Assign type of algorithm

    if(strcmp(algorithm,"NextFit")==0){
        printf("Using NextFit algorithm.\n");
        allocate = nextFit;
        NEXTFITNODE = HEAD;

    }
    else if(strcmp(algorithm, "BestFit")==0){
        printf("Using BestFit algorithm.\n");
        allocate = bestFit;

    }
    else if(strcmp(algorithm,"WorstFit")==0){
        printf("Using WorstFit algorithm.\n");
        allocate = worstFit;

    }
    //Default first fit
    else{
        printf("Using FirstFit algorithm.\n");
        allocate = firstFit;

    }


    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory
    address.
 */
void deallocate ( void * memory ){

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;


    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a
           node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked
               list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful\n");
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;

                //If NextFit algorithm is set, handle the pointer to it.
                if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
                        printf("Moving NEXTFITNODE due to coalace taking place.\n
                           ");
                        NEXTFITNODE = connectedHolesPointer;
                    }
                 }

                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }
```

```
    }

}
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),point->memory);
            //look at next node.
            point = point->nextNode;
        }

    }
}



int main() {
    /*
     * TEST HARNESS FOR EACH ALGORITHM COMMENTED OUT.
     */

    //Set the initial heap size.
    size_t size = 1024;

    //Allocate memory for the heap.
    void *memory = malloc(size);

    //FirstFit Algorithm test harness
    //The test allocation should go into node x.

    char * algo = "FirstFit";
    initialise(memory,size,algo);

    void *x = allocate(100);
    printf("allocate x output: %d\n===============================\n\nVisual Node output:\n\n",(char*)x);
    output();

    void *y = allocate(50);
    printf("allocate y output: %d\n===============================\n\nVisual Node output:\n\n",(char*)y);
    output();

```

```c
    void *z = allocate(100);
    printf("allocate z output: %d\n=============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    void *t = allocate(100);
    printf("allocate t output: %d\n=============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    deallocate(x);
    deallocate(z);

    printf("De-allocate x & z: \n=============================\n\nVisual Node output:\n\n");
    output();

    void *test = allocate(70);
    printf("test allocation of 70: \n=============================\n\nVisual Node output:\n\n");
    output();



    //NextFit Algorithm test harness
    //The test allocation should go into node last node, as that is the NEXTFITNODE due to the algorithm.
    /*
    char * algo = "NextFit";
    initialise(memory,size,algo);

    void *x = allocate(100);
    printf("allocate x output: %d\n=============================\n\nVisual Node output:\n\n",(char*)x);
    output();

    void *y = allocate(50);
    printf("allocate y output: %d\n=============================\n\nVisual Node output:\n\n",(char*)y);
    output();

    void *z = allocate(100);
    printf("allocate z output: %d\n=============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    void *t = allocate(100);
    printf("allocate t output: %d\n=============================\n\nVisual Node output:\n\n",(char*)z);
    output();

    deallocate(x);
    deallocate(z);

    printf("De-allocate x & z: \n=============================\n\nVisual Node output:\n\n");
    output();

    void *test = allocate(70);
    printf("test allocation of 70: \n=============================\n\nVisual Node output:\n\n");
    output();
    */

    //BestFit Algorithm Test Harness
    //test1 should go in x, test2 should go in z.
    /*
    char * algo = "BestFit";
    initialise(memory,size,algo);

    void *x = allocate(50);
    void *y = allocate(100);
    void *z = allocate(200);
    void *b = allocate(30);
    printf("Allocate 4 nodes: \n================================\n\nVisual Node output:\n\n");
    output();

    deallocate(x);
    deallocate(z);

    printf("De-allocate x & z: \n================================\n\nVisual Node output:\n\n");
    output();

    //Should replace node x;
    void *test1 = allocate(50);
    printf("test allocation of 50: \n================================\n\nVisual Node output:\n\n");
    output();

    deallocate(test1);
    printf("De-allocation of testAllocation: \n================================\n\nVisual Node output:\n\n");
    output();

    //Should use node z, and create new hole:
    void *test2 = allocate(150);
    printf("test2 of 150: \n================================\n\nVisual Node output:\n\n");
    output();
     */


    //WorstFit Algorithm Test Harness
    //test1 should go in the hole after b, and so should test2
    /*
    char * algo = "WorstFit";
    initialise(memory,size,algo);

    void *x = allocate(50);
    void *y = allocate(100);
    void *z = allocate(200);
    void *b = allocate(30);
    printf("Allocate 4 nodes: \n================================\n\nVisual Node output:\n\n");
    output();

    deallocate(x);
    deallocate(z);

    printf("De-allocate x & z: \n================================\n\nVisual Node output:\n\n");
    output();

    //Should replace node x;
    void *test1 = allocate(50);
    printf("test allocation of 50: \n==================================\n\nVisual
       Node output:\n\n");
    output();

    deallocate(test1);
    printf("De-allocation of testAllocation: \n==================================\n\
       nVisual Node output:\n\n");
    output();

    //Should use node z, and create new hole:
    void *test2 = allocate(150);
    printf("test allocation of 150: \n==================================\n\nVisual
       Node output:\n\n");
    output();
     */
    free(memory);
    return EXIT_SUCCESS;

}
```

part3.h

```

#ifndef CW2_PART3_H
#define CW2_PART3_H

typedef struct NodeStruct{

    int free;
    size_t size;
    void *memory;
    struct NodeStruct* prevNode;
    struct NodeStruct* nextNode;

}Node;


void*(*allocate)(size_t);
void deallocate ( void * memory );
void initialise ( void * memory , size_t size, char* algorithm);

void *firstFit(size_t bytes);
void *nextFit(size_t bytes);
void *bestFit(size_t bytes);
void *worstFit(size_t bytes);

#endif //CW2_PART3_H
```

part3.c

```
/*
    --------------------------------------------------------------------------------------------------------
/                                                                                 /
/    Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory
    Manager.                   /
/                                                                                 /
/                                                 part3
                                                                        /
/                                                                                 /
/    Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.
                                                  /
/                                                                                 /
/    Last edit date: 06/12/19
                                                                                            /
/                                                                                 /
/    Description: Program to simulate a thread-safe memory manager using a linked
    list                    /
/                  implementation.
                                                                                       /
/                                                                                 /
  ------------------------------------------------------------------------------------------------------------
     */


#include <stdio.h>
#include <stdlib.h>
#include "part3.h"
#include <pthread.h>
#include <string.h>


//global variables.
Node *HEAD = NULL;
Node *NEXTFITNODE = NULL;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


//Function Set as a function pointer so that the different algorithms can be
    applied.
/*
 * Inputs: size_t variable.
 * Outputs: NONE.
 * Description: function pointer to other algorithm allocate functions.
 */
void*(*allocate)(size_t);

/*
 * Inputs: Node* nodePointer, size_t bytes.
```

```c
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free
     node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating
       node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a
        struct.
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *firstFit(size_t bytes) {

    //Used to stop the function allocating 0 bytes to a node if the calling
       thread requests it, if not handled
    //node would be invalid.
    if((int)bytes <= 0){
        printf("Thread ID: %d, attempted to allocate 0 memory, returning NULL
           .!!!\n",pthread_self());
        return NULL;
    }
    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;
    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated by thread %d.\n",bytes,pthread_self());

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("thread: %d: found node of exact size.\n",pthread_self());

            //Set free to 0 as node is taken.
            nodePointer->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Unlock Mutex
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the NextFit algorithm to allocate memory from the heap.
 */
void* nextFit(size_t bytes){

    //Used to stop the function allocating 0 bytes to a node if the calling
```

```
       thread requests it, if not handled node
    // would be invalid.
    if((int)bytes <= 0){
        printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
           NULL.!!!\n",pthread_self());
        return NULL;
    }
   printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial start location to the NEXTFITNODE node.
    Node *nodePointer = NEXTFITNODE;

    //Set the end node to the start node, therefore it will only loop once and
       not infinitely.
    Node *limitNode = NEXTFITNODE;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node *returnNode = allocateNodeWithHole(nodePointer,bytes);

            //update NEXTFITNODE.
            NEXTFITNODE = returnNode->nextNode;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return nodePointer->memory;
        }

            //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            //Set free to 0 as node is taken.
            printf("Found node of exact size.\n");
            nodePointer->free = 0;

            //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
                NEXTFITNODE = nodePointer->nextNode;
            }
            else{
                NEXTFITNODE = HEAD;
            }

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return nodePointer->memory;

        }
        //if the node is taken.
        else {

            //Look at next node.
            if (nodePointer->nextNode == NULL) {

                //loop through whole list
                nodePointer = HEAD;
            }
            else{
                nodePointer = nodePointer->nextNode;
            }
            if(nodePointer->memory == limitNode->memory){

                printf("!!!! No free nodes, returning NULL !!!!\n");

                //update NEXTFITNODE.
                NEXTFITNODE= HEAD;

                //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return  NULL;
            }


        }
    }
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the BestFit algorithm to allocate memory from the heap.
 */
void* bestFit(size_t bytes){

    //Used to stop the function allocating 0 bytes to a node if the calling
       thread requests it, if not handled
    // node would be invalid.
    if((int)bytes <= 0){
        printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
           NULL.!!!\n",pthread_self());
        return NULL;
    }

    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the best node that fits.
    int found = 0;
    int bestFit = 1;
    Node *bestNode = NULL;
    Node *nodePointer = HEAD;

    while(bestFit){
        if(nodePointer->free){

            //IF the node is of exact size.
            if (nodePointer->size == bytes){
                bestNode = nodePointer;
                found = 1;
                break;
            }

            //IF the node is greater than bytes
            if(nodePointer->size > (bytes + (sizeof(Node)))){
                found = 1;
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                if(nodePointer->size < bestNode->size){
                    bestNode = nodePointer;
                }

            }
            //If there is enough size to create a new node but not assign a size > 0 to it.
            if(nodePointer->size == (bytes +(sizeof(Node)))){
                printf("Invalid node to use. searching next\n");

            }
            //If there is a next node.
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                break;

            }
        }
        //Look at next node.
        else{
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                printf("!!!! No free nodes, returning NULL !!!!\n");
                //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return NULL;
            }
        }


    }

    //Allocate the best node found with the desired bytes.
    if(found) {
        //If the node is free and is large enough for the allocated bytes.
        if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(bestNode,bytes);

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }


        //if the node is the exact size.
        else if (bestNode->free == 1 && bestNode->size == bytes) {

            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            bestNode->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return bestNode->memory;
        }

    }
    //If no suitable nodes found, return NULL.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");

        //Unlock the Mutex.
        pthread_mutex_unlock(&mutex);

        //Return NULL as no valid nodes found.
        return NULL;

    }



}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the WorstFit algorithm to allocate memory from the heap.
 */
void* worstFit(size_t bytes) {

    //Used to stop the function allocating 0 bytes to a node if the calling
       thread requests it, if not handled
    // node would be invalid.
    if((int)bytes <= 0){
         printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
            NULL.!!!\n",pthread_self());
        return NULL;
    }

    printf("\n %d: attempting to gain lock.\n",pthread_self());
    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.

    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the worst node that fits.
    int found = 0;
    int worstFit = 1;
    Node *worstNode = NULL;
    Node *nodePointer = HEAD;

    while (worstFit) {
        if (nodePointer->free) {

            //IF the node is of exact size.
            if (nodePointer->size == bytes) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                } else if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            }

                //IF the node is greater than bytes
            else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                }
                if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                printf("Invalid node to use. searching next\n");

            }

        }
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            } else {
                break;
            }

    }

    if(found) {

        //If the node is free and is large enough for the allocated bytes.
        if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(worstNode,bytes);

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }



        //if the node is the exact size.
        else if (worstNode->free == 1 && worstNode->size == bytes) {
            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            worstNode->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);


            //return the allocated nodes memory address.
            return worstNode->memory;

        }

    }
    //If not found.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");

        //Unlock the Mutex.
        pthread_mutex_unlock(&mutex);

        return NULL;

    }
}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the initial HEAD node.
*/
void initialise(void *memory ,size_t size, char* algorithm){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);


    //Assign type of algorithm

    if(strcmp(algorithm,"NextFit")==0){
        allocate = nextFit;
        NEXTFITNODE = HEAD;

    }
    else if(strcmp(algorithm, "BestFit")==0){
        allocate = bestFit;

    }
    else if(strcmp(algorithm,"WorstFit")==0){
        allocate = worstFit;
```

```c
    }
    //Default first fit
    else{
        allocate = firstFit;

    }


    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory address.
 */
void deallocate ( void * memory ){

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;


    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful of thread ID: %d.\n",pthread_self())
                ;
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;

                //If NextFit algorithm is set, handle the pointer to it.
                if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
                        printf("Moving NEXTFITNODE due to coalace taking place.\n
                            ");
                        NEXTFITNODE = connectedHolesPointer;
                    }
                 }

                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
```

```c
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }

    }
    //Unlock the Mutex.
    pthread_mutex_unlock(&mutex);

}
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                       point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                       point->memory);
            //look at next node.
            point = point->nextNode;
        }

    }
}


/*
 * Inputs: void.
 * Outputs: void.
 * Description: Used by threads that act as a library to randomly allocate random
     amounts of memory.
 */
```

```
void* threadAllocate(){
    srand(time(NULL));
    for (int i = 0; i < 500; i++) {
        size_t allocateAmount = rand() % 200;
        printf("ID: %d, trying to allocate: %d.\n",pthread_self(),allocateAmount)
            ;
        void *x = allocate(allocateAmount);
    }
}


/*
 * Inputs: void.
 * Outputs: void.
 * Description: Used by threads that act as a library to randomly deallocate
    random amounts of memory.
 */
void* threadDeallocate(void* memory){
    srand(time(NULL));
    for (int i=0; i<800; i++){
        int x = rand() % ((memory+1024) + 1 - memory) + memory;
        deallocate((void*)x);
    }
}
```

5013ACW02-100237137-file2.c

Filename scrubbed (one or more forbidden characters found). Original name:

`part3_test.c`

```c
/*
   --------------------------------------------------------------------------------------------------
/
   /
/   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory
   Manager.                /
/
   /
/                                         part3 TEST HARNESS
                                             /
/
   /
/   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.
                                  /
/
   /
/   Last edit date: 06/12/19
                                                                              /
/
   /
/   Description: Program to simulate a thread-safe memory manager using a linked
   list                    /
/                   implementation.
                                                                          /
/
   /
  ----------------------------------------------------------------------------------------------------
    */

#include <stdio.h>
#include <stdlib.h>
#include "part3.h"
#include <pthread.h>
#include <string.h>


//global variables.
Node *HEAD = NULL;
Node *NEXTFITNODE = NULL;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


//Function Set as a function pointer so that the different algorithms can be
   applied.
/*
 * Inputs: size_t variable.
 * Outputs: NONE.
 * Description: function pointer to other algorithm allocate functions.
 */
void*(*allocate)(size_t);
```

```c
/*
 * Inputs: Node* nodePointer, size_t bytes.
 * Outputs: Node* (return node).
 * Description: Function takes a node, allocates a size to it, and creates a free node next to it with the remaining space.
 */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating node\n");

    //Allocate memory for the struct hole node at the current nodes memory plus a struct.
    Node *nextNode = (Node *) (nodePointer->memory + bytes);

    //memory address at the start of the empty hole node.
    void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

    //Assign the hole nodes variables.
    nextNode->memory = memoryStart2;
    nextNode->free = 1;
    nextNode->size = nodePointer->size - bytes - sizeof(Node);
    nextNode->nextNode = nodePointer->nextNode;
    nextNode->prevNode = nodePointer;

    //allocate the taken size.
    nodePointer->size = bytes;

    //Set free bool to 0 as this node is taken.
    nodePointer->free = 0;

    //Point the current node to the next node which is the created hole
    nodePointer->nextNode = nextNode;

    //return the taken nodes memory address.
    return nodePointer;
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the FirstFit algorithm to allocate memory from the heap.
 */
void *firstFit(size_t bytes) {

    //Used to stop the function allocating 0 bytes to a node if the calling thread requests it, if not handled
    //node would be invalid.
    if((int)bytes <= 0){
        printf("Thread ID: %d, attempted to allocate 0 memory, returning NULL.!!!\n",pthread_self());
        return NULL;
    }
    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial start location to the HEAD node.
    Node *nodePointer = HEAD;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(nodePointer,bytes);

            printf("%d bytes allocated by thread %d.\n",bytes,pthread_self());

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //Return the memory address of the allocated node.
            return returnNode->memory;
        }

            //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            printf("thread: %d: found node of exact size.\n",pthread_self());

            //Set free to 0 as node is taken.
            nodePointer->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //Return the memory address of the allocated node.
            return nodePointer->memory;

        }
            //if the node is taken.
        else {
            //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
                printf("!!!! No free nodes, returning NULL !!!!\n");

                //Unlock Mutex
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return NULL;
            }
            //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
        }
    }

}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the NextFit algorithm to allocate memory from the heap.
 */
```

```
void* nextFit(size_t bytes){

    //Used to stop the function allocating 0 bytes to a node if the calling
       thread requests it, if not handled node
    // would be invalid.
    if((int)bytes <= 0){
        printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
           NULL.!!!\n",pthread_self());
        return NULL;
    }
    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial start location to the NEXTFITNODE node.
    Node *nodePointer = NEXTFITNODE;

    //Set the end node to the start node, therefore it will only loop once and
       not infinitely.
    Node *limitNode = NEXTFITNODE;

    int allocateBool = 1;

    //Loop through every node.
    while (allocateBool == 1) {

        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {

            //Call the allocate function to allocate the node.
            Node *returnNode = allocateNodeWithHole(nodePointer,bytes);

            //update NEXTFITNODE.
            NEXTFITNODE = returnNode->nextNode;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return nodePointer->memory;
        }

            //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {

            //Set free to 0 as node is taken.
            printf("Found node of exact size.\n");
            nodePointer->free = 0;

            //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
                NEXTFITNODE = nodePointer->nextNode;
            }
            else{
                NEXTFITNODE = HEAD;
            }

            //Unlock the Mutex.
```

```c
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return nodePointer->memory;

        }
            //if the node is taken.
        else {

            //Look at next node.
            if (nodePointer->nextNode == NULL) {

                //loop through whole list
                nodePointer = HEAD;
            }
            else{
                nodePointer = nodePointer->nextNode;
            }
            if(nodePointer->memory == limitNode->memory){

                printf("!!!! No free nodes, returning NULL !!!!\n");

                //update NEXTFITNODE.
                NEXTFITNODE= HEAD;

                //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return  NULL;
            }


        }
    }
}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the BestFit algorithm to allocate memory from the heap.
 */
void* bestFit(size_t bytes){

    //Used to stop the function allocating 0 bytes to a node if the calling thread requests it, if not handled
    // node would be invalid.
    if((int)bytes <= 0){
        printf("==== Worker ID: %d, attempted to allocate 0 memory, returning NULL.!!!\n",pthread_self());
        return NULL;
    }

    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the best node that fits.
    int found = 0;
    int bestFit = 1;
    Node *bestNode = NULL;
    Node *nodePointer = HEAD;

    while(bestFit){
        if(nodePointer->free){

            //IF the node is of exact size.
            if (nodePointer->size == bytes){
                bestNode = nodePointer;
                found = 1;
                break;
            }

            //IF the node is greater than bytes
            if(nodePointer->size > (bytes + (sizeof(Node)))){
                found = 1;
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                if(nodePointer->size < bestNode->size){
                    bestNode = nodePointer;
                }

            }
            //If there is enough size to create a new node but not assign a size > 0 to it.
            if(nodePointer->size == (bytes +(sizeof(Node)))){
                printf("Invalid node to use. searching next\n");

            }
            //If there is a next node.
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                if(bestNode == NULL){
                    bestNode = nodePointer;
                }
                break;

            }
        }
            //Look at next node.
        else{
            if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
            }
            else{
                printf("!!!! No free nodes , returning NULL !!!!\n");
                //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);

                //Return NULL as no valid nodes found.
                return NULL;
            }
        }


    }

```

```c
    //Allocate the best node found with the desired bytes.
    if(found) {
        //If the node is free and is large enough for the allocated bytes.
        if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(bestNode,bytes);

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }


            //if the node is the exact size.
        else if (bestNode->free == 1 && bestNode->size == bytes) {

            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            bestNode->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return bestNode->memory;
        }

    }
        //If no suitable nodes found, return NULL.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");

        //Unlock the Mutex.
        pthread_mutex_unlock(&mutex);

        //Return NULL as no valid nodes found.
        return NULL;

    }



}


/*
 * Inputs: size_t bytes.
 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the WorstFit algorithm to allocate memory from the heap.
 */
void* worstFit(size_t bytes) {

    //Used to stop the function allocating 0 bytes to a node if the calling thread requests it, if not handled
    // node would be invalid.
    if((int)bytes <= 0){
        printf("==== Worker ID: %d, attempted to allocate 0 memory, returning NULL.!!!\n",pthread_self());
        return NULL;
    }

    printf("\n %d: attempting to gain lock.\n",pthread_self());

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //When the thread gains the lock, attempt to allocate the input bytes.

    printf("\n %d: acquired lock\n",pthread_self());

    //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the worst node that fits.
    int found = 0;
    int worstFit = 1;
    Node *worstNode = NULL;
    Node *nodePointer = HEAD;

    while (worstFit) {
        if (nodePointer->free) {

            //IF the node is of exact size.
            if (nodePointer->size == bytes) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                } else if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            }

                //IF the node is greater than bytes
            else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                found = 1;
                if (worstNode == NULL) {
                    worstNode = nodePointer;
                }
                if (nodePointer->size > worstNode->size) {
                    worstNode = nodePointer;
                }

            } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                printf("Invalid node to use. searching next\n");

            }

        }
        if (nodePointer->nextNode != NULL) {
            nodePointer = nodePointer->nextNode;
        } else {
            break;
        }

    }

    if(found) {

        //If the node is free and is large enough for the allocated bytes.
        if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {

            //Call the allocate function to allocate the node.
            Node* returnNode = allocateNodeWithHole(worstNode,bytes);

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);

            //return the allocated nodes memory address.
            return returnNode->memory;
        }



            //if the node is the exact size.
        else if (worstNode->free == 1 && worstNode->size == bytes) {
            printf("found node of exact size.\n");
            //Set free to 0 as node is taken.
            worstNode->free = 0;

            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);


            //return the allocated nodes memory address.
            return worstNode->memory;

        }

    }
        //If not found.
    else {
        printf("!!!! No free nodes, returning NULL !!!!\n");

        //Unlock the Mutex.
        pthread_mutex_unlock(&mutex);

        return NULL;

    }
}


/*
* Inputs: void* memory, size_t size, char* algorithm.
* Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the initial HEAD node.
*/
void initialise(void *memory ,size_t size, char* algorithm){

    //Allocate (size) amount of memory to the heap (called memory in this case).
    //Returns pointer to the start of the address.

    //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);


    //Assign type of algorithm

    if(strcmp(algorithm,"NextFit")==0){
        allocate = nextFit;
        NEXTFITNODE = HEAD;

    }
    else if(strcmp(algorithm, "BestFit")==0){
        allocate = bestFit;

    }
    else if(strcmp(algorithm,"WorstFit")==0){
        allocate = worstFit;

    }
        //Default first fit
    else{
        allocate = firstFit;

    }


    //memoryStart is the memory address where data can be stored.
    //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);

    //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
    HEAD->free = 1;
    HEAD->nextNode = NULL;
    HEAD->prevNode = NULL;

}

/*
 * Inputs: void* memory.
 * Outputs: void.
 * Description: Deallocates a node (frees it) based on the input of its memory address.
 */
void deallocate ( void * memory ){

    //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);

    //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;

    //int to act as bool for while loop.
    int deallocateBool = 1;


    while(deallocateBool == 1){

        //if the memory address of dealocation equals the memory address of a node.

        if(nodePointer->memory != memory){

            //if the next node is null, current node is the last in the linked list.
            if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
                break;
            }
            else {

                //look at next node.
                nodePointer = nodePointer->nextNode;
            }
        }

```

```
            //else the node is the memory address.
        else{
            //Set the nodes free variable to 1, as node has been deallocated.
            printf("De-allocation successful of thread ID: %d.\n",pthread_self())
               ;
            nodePointer->free=1;
            deallocateBool = 0;
            break;
        }
    }

    //Next part coalaces connected holes.

    //int to act as bool for while loop.
    int connectedHoleSearch = 1;

    //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;


    while (connectedHoleSearch == 1){

        //if the node is free.
        if (connectedHolesPointer == NULL){
            break;
        }
        if (connectedHolesPointer->free == 1){

            //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){

                //set the loop bool to 0.
                connectedHoleSearch = 0;
                break;
            }

            //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {

                //Increase the current nodes size, to that containing both nodes.
                connectedHolesPointer->size = connectedHolesPointer->size +
                                              sizeof(Node)+connectedHolesPointer
                                                 ->nextNode->size;

                //If NextFit algorithm is set, handle the pointer to it.
                if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
                        printf("Moving NEXTFITNODE due to coalace taking place.\n
                           ");
                        NEXTFITNODE = connectedHolesPointer;
                    }
                }

                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                    connectedHoleSearch = 0;
                    break;
                }

                else {
                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;

                    //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
                }

            }
            else{

                //Look at next node.
                connectedHolesPointer = connectedHolesPointer->nextNode;
            }

        }
        else{
            //look at next node.
            connectedHolesPointer = connectedHolesPointer->nextNode;

        }

    }
    //Unlock the Mutex.
    pthread_mutex_unlock(&mutex);

}
/*
 * Inputs: void
 * Outputs: void
 * Description: Outputs all nodes.
 */
void output(){
    Node *point = HEAD;
    int loop = 1;

    while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
            loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                   (int)point->size,point->free,(point->memory-sizeof(Node)),
                      point->memory);
            break;
        }else{
            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
               %d\n\n",
                   (int)point->size,point->free,(point->memory-sizeof(Node)),
                      point->memory);
            //look at next node.
            point = point->nextNode;
        }

    }
}


/*
 * Inputs: void.
 * Outputs: void.
 * Description: Used by threads that act as a library to randomly allocate random
     amounts of memory.
 */
void* threadAllocate(){
    srand(time(NULL));
    for (int i = 0; i < 500; i++) {
        size_t allocateAmount = rand() % 200;
        printf("ID: %d, trying to allocate: %d.\n",pthread_self(),allocateAmount)
           ;
        void *x = allocate(allocateAmount);
    }
}


/*
 * Inputs: void.
 * Outputs: void.
 * Description: Used by threads that act as a library to randomly deallocate
    random amounts of memory.
 */
void* threadDeallocate(void* memory){
    srand(time(NULL));
    for (int i=0; i<800; i++){
        int x = rand() % ((memory+1024) + 1 - memory) + memory;
        deallocate((void*)x);
    }
}


int main() {

    //Set the initial heap size.
    size_t size = 1024;

    //Allocate memory for the heap.
    void *memory = malloc(size);

    //TEST 1
    /*
    //Set the algorithm type.
    char * algo = "FirstFit";

    //Initialise the memory manager.
    initialise(memory,size,algo);

    //Output nodes.
    output();

    //Create and join all threads.
    //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this
       program.
    //They hit the allocate many times, so most nodes should be taken by the end.
    pthread_t thread1,thread2,thread3,thread4,thread5,thread6,thread7,thread8,
       thread9,
            thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
    pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
    pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
    pthread_create(&thread6, NULL, threadAllocate, NULL);
    pthread_create(&thread7, NULL, threadAllocate, NULL);
    pthread_create(&thread8, NULL, threadAllocate, NULL);
    pthread_create(&thread9, NULL, threadDeallocate, memory);
    pthread_create(&thread10, NULL, threadDeallocate, memory);
    pthread_create(&thread11, NULL, threadDeallocate, memory);
    pthread_create(&thread12, NULL, threadDeallocate, memory);
    pthread_create(&thread13, NULL, threadDeallocate, memory);
    pthread_create(&thread14, NULL, threadDeallocate, memory);
    pthread_create(&thread15, NULL, threadDeallocate, memory);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    pthread_join(thread4,NULL);
    pthread_join(thread5,NULL);
    pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
    pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
    pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
    pthread_join(thread12,NULL);
    pthread_join(thread13,NULL);
    pthread_join(thread14,NULL);
    pthread_join(thread15,NULL);


    //Final node output.
    printf("\n\n\n\nFinal node output:\n\n");
    output();
     */
    //TEST 2
    /*
    //Set the algorithm type.
    char * algo = "NextFit";

    //Initialise the memory manager.
    initialise(memory,size,algo);

    //Output nodes.
    output();

    //Create and join all threads.
    //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this program.
    //They hit the allocate many times, so most nodes should be taken by the end.
    pthread_t thread1,thread2,thread3,thread4,thread5,thread6,thread7,thread8,thread9,
            thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
    pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
    pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
    pthread_create(&thread6, NULL, threadAllocate, NULL);
    pthread_create(&thread7, NULL, threadAllocate, NULL);
    pthread_create(&thread8, NULL, threadAllocate, NULL);
    pthread_create(&thread9, NULL, threadDeallocate, memory);
    pthread_create(&thread10, NULL, threadDeallocate, memory);
    pthread_create(&thread11, NULL, threadDeallocate, memory);
    pthread_create(&thread12, NULL, threadDeallocate, memory);
    pthread_create(&thread13, NULL, threadDeallocate, memory);
    pthread_create(&thread14, NULL, threadDeallocate, memory);
    pthread_create(&thread15, NULL, threadDeallocate, memory);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    pthread_join(thread4,NULL);
    pthread_join(thread5,NULL);
    pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
    pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
    pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
    pthread_join(thread12,NULL);
    pthread_join(thread13,NULL);
    pthread_join(thread14,NULL);
    pthread_join(thread15,NULL);


    //Final node output.
    printf("\n\n\n\nFinal node output:\n\n");
    output();
     */
    //TEST 3
    /*
    //Set the algorithm type.
    char * algo = "BestFit";

    //Initialise the memory manager.
    initialise(memory,size,algo);

    //Output nodes.
    output();

    //Create and join all threads.
    //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this
        program.
    //They hit the allocate many times, so most nodes should be taken by the end.
    pthread_t thread1,thread2,thread3,thread4,thread5,thread6,thread7,thread8,
        thread9,
            thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
    pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
    pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
    pthread_create(&thread6, NULL, threadAllocate, NULL);
    pthread_create(&thread7, NULL, threadAllocate, NULL);
    pthread_create(&thread8, NULL, threadAllocate, NULL);
    pthread_create(&thread9, NULL, threadDeallocate, memory);
    pthread_create(&thread10, NULL, threadDeallocate, memory);
    pthread_create(&thread11, NULL, threadDeallocate, memory);
    pthread_create(&thread12, NULL, threadDeallocate, memory);
    pthread_create(&thread13, NULL, threadDeallocate, memory);
    pthread_create(&thread14, NULL, threadDeallocate, memory);
    pthread_create(&thread15, NULL, threadDeallocate, memory);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    pthread_join(thread4,NULL);
    pthread_join(thread5,NULL);
    pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
    pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
    pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
    pthread_join(thread12,NULL);
    pthread_join(thread13,NULL);
    pthread_join(thread14,NULL);
    pthread_join(thread15,NULL);


    //Final node output.
    printf("\n\n\n\nFinal node output:\n\n");
    output();
     */
    //TEST 4
    /*
    //Set the algorithm type.
    char * algo = "WorstFit";

    //Initialise the memory manager.
    initialise(memory,size,algo);

    //Output nodes.
    output();

    //Create and join all threads.
    //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this
        program.
    //They hit the allocate many times, so most nodes should be taken by the end.
    pthread_t thread1,thread2,thread3,thread4,thread5,thread6,thread7,thread8,
        thread9,
            thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
    pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
    pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
    pthread_create(&thread6, NULL, threadAllocate, NULL);
    pthread_create(&thread7, NULL, threadAllocate, NULL);
    pthread_create(&thread8, NULL, threadAllocate, NULL);
    pthread_create(&thread9, NULL, threadDeallocate, memory);
    pthread_create(&thread10, NULL, threadDeallocate, memory);
    pthread_create(&thread11, NULL, threadDeallocate, memory);
    pthread_create(&thread12, NULL, threadDeallocate, memory);
    pthread_create(&thread13, NULL, threadDeallocate, memory);
    pthread_create(&thread14, NULL, threadDeallocate, memory);
    pthread_create(&thread15, NULL, threadDeallocate, memory);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    pthread_join(thread4,NULL);
    pthread_join(thread5,NULL);
    pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
    pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
    pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
    pthread_join(thread12,NULL);
    pthread_join(thread13,NULL);
    pthread_join(thread14,NULL);
    pthread_join(thread15,NULL);


    //Final node output.
    printf("\n\n\n\nFinal node output:\n\n");
    output();
     */

    free(memory);
    return EXIT_SUCCESS;
}
```