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

```
1
#ifndef CW2_PART1_H
3 #define CW2_PART1_H

5 typedef struct NodeStruct{

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

13 }Node;

15
void *allocate(size_t);
17 void deallocate ( void * memory );
void initialise ( void * memory , size_t size);
19

21 #endif //CW2_PART1_H
```

part1.c

```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.
```

```
    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;
```

```
101             //Return the memory address of the allocated node.
                return nodePointer->memory;
103
            }
105         //if the node is taken.
            else {
107             //If the next node is NULL.
                if (nodePointer->nextNode == NULL) {
109                 printf("!!!! No free nodes, returning NULL !!!!\n");

111                 //Return NULL as no valid nodes found.
                    return NULL;
113             }
                //Look at next node, and continue in the loop.
115             nodePointer = nodePointer->nextNode;
            }
117     }

119 }

121
    /*
123 * Inputs: void* memory, size_t size, char* algorithm.
    * Outputs: void.
125 * Description: Initialises the type of allocation algorithm and creates the
       initial HEAD node.
    */
127 void initialise(void *memory ,size_t size){

129     //Allocate (size) amount of memory to the heap (called memory in this case).
        //Returns pointer to the start of the address.
131
        //Allocate the head node the memory it needs.
133     HEAD = (Node*)(memory);

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

139     //Assign all the nodes variables.
        HEAD->size = size - sizeof(Node);
141     HEAD->memory = memoryStart;
        HEAD->free = 1;
143     HEAD->nextNode = NULL;
        HEAD->prevNode = NULL;
145
    }
147
    /*
149  * Inputs: void* memory.
     * Outputs: void.
151  * Description: Deallocates a node (frees it) based on the input of its memory
        address.
     */
153 void deallocate ( void * memory ){

155     //Assign new node pointer to HEAD node.
        Node *nodePointer = HEAD;
157
        //int to act as bool for while loop.
159     int deallocateBool = 1;
```

```
161
    while(deallocateBool == 1){
163
        //if the memory address of dealocation equals the memory address of a
            node.
165
        if(nodePointer->memory != memory){
167
            //if the next node is null, current node is the last in the linked
                list.
169         if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
171             break;
            }
173         else {
175             //look at next node.
                nodePointer = nodePointer->nextNode;
177         }
        }
179
            //else the node is the memory address.
181     else{
            //Set the nodes free variable to 1, as node has been deallocated.
183         printf("De-allocation successful\n");
            nodePointer->free=1;
185         deallocateBool = 0;
            break;
187     }
    }
189
    //Next part coalaces connected holes.
191
    //int to act as bool for while loop.
193 int connectedHoleSearch = 1;

195 //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;
197

199 while (connectedHoleSearch == 1){

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

207         //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){
209
                //set the loop bool to 0.
211             connectedHoleSearch = 0;
                break;
213         }

215         //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {
217
                //Increase the current nodes size, to that containing both nodes.
219             connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;
221
```

```
223
                    //if the next next node is NULL
225                 if (connectedHolesPointer->nextNode->nextNode == NULL){
                        //Assign a new node pointer to current nodes next next node.
227                     Node *nextNode = connectedHolesPointer->nextNode->nextNode;

229                     //Assign the current nodes next node to ^.
                        connectedHolesPointer->nextNode = nextNode;
231                     connectedHoleSearch = 0;
                        break;
233                 }

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

239                     //Assign the current nodes next node to ^.
                        connectedHolesPointer->nextNode = nextNode;
241                 }

243             }
                else{
245
                    //Look at next node.
247                 connectedHolesPointer = connectedHolesPointer->nextNode;
                }
249
            }
251         else{
                //look at next node.
253             connectedHolesPointer = connectedHolesPointer->nextNode;

255         }

257     }

259 }
    /*
261  * Inputs: void
     * Outputs: void
263  * Description: Outputs all nodes.
     */
265 void output(){
        Node *point = HEAD;
267     int loop = 1;

269     while (loop ==1){
            //if next node is null, at the end of the linked list.
271         if (!point->nextNode){
                loop = 0;
273             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);
275             break;
            }else{
277             //Output information about a node.
                printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
                    %d\n\n",
279                     (int)point->size,point->free,(point->memory-sizeof(Node)),
                            point->memory);
                //look at next node.
```

```
281                 point = point->nextNode;
            }
283
        }
285 }
```

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");
```

```
97
                //Set free to 0 as node is taken.
99              nodePointer->free = 0;

101             //Return the memory address of the allocated node.
                return nodePointer->memory;
103
            }
105         //if the node is taken.
            else {
107             //If the next node is NULL.
                if (nodePointer->nextNode == NULL) {
109                 printf("!!!! No free nodes, returning NULL !!!!\n");

111                 //Return NULL as no valid nodes found.
                    return NULL;
113             }
                //Look at next node, and continue in the loop.
115             nodePointer = nodePointer->nextNode;
            }
117     }

119 }


121
    /*
123 * Inputs: void* memory, size_t size, char* algorithm.
    * Outputs: void.
125 * Description: Initialises the type of allocation algorithm and creates the
        initial HEAD node.
    */
127 void initialise(void *memory ,size_t size){

129     //Allocate (size) amount of memory to the heap (called memory in this case).
        //Returns pointer to the start of the address.
131
        //Allocate the head node the memory it needs.
133     HEAD = (Node*)(memory);

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

139     //Assign all the nodes variables.
        HEAD->size = size - sizeof(Node);
141     HEAD->memory = memoryStart;
        HEAD->free = 1;
143     HEAD->nextNode = NULL;
        HEAD->prevNode = NULL;
145
    }
147
    /*
149  * Inputs: void* memory.
     * Outputs: void.
151  * Description: Deallocates a node (frees it) based on the input of its memory
        address.
     */
153 void deallocate ( void * memory ){

155     //Assign new node pointer to HEAD node.
        Node *nodePointer = HEAD;
157
```

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

161

    while(deallocateBool == 1){
163
        //if the memory address of dealocation equals the memory address of a
            node.
165
        if(nodePointer->memory != memory){
167
            //if the next node is null, current node is the last in the linked
                list.
169         if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
171             break;
            }
173         else {

175             //look at next node.
                nodePointer = nodePointer->nextNode;
177         }
        }
179
            //else the node is the memory address.
181     else{
            //Set the nodes free variable to 1, as node has been deallocated.
183         printf("De-allocation successful\n");
            nodePointer->free=1;
185         deallocateBool = 0;
            break;
187     }
    }
189
    //Next part coalaces connected holes.
191
    //int to act as bool for while loop.
193 int connectedHoleSearch = 1;

195 //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;
197


199 while (connectedHoleSearch == 1){

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

207         //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){
209
                //set the loop bool to 0.
211             connectedHoleSearch = 0;
                break;
213         }

215         //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {
217
                //Increase the current nodes size, to that containing both nodes.
```

```
219                 connectedHolesPointer->size = connectedHolesPointer->size +
                            sizeof(Node)+connectedHolesPointer->nextNode->size;
221
223
                    //if the next next node is NULL
225                 if (connectedHolesPointer->nextNode->nextNode == NULL){
                        //Assign a new node pointer to current nodes next next node.
227                     Node *nextNode = connectedHolesPointer->nextNode->nextNode;

229                     //Assign the current nodes next node to ^.
                        connectedHolesPointer->nextNode = nextNode;
231                     connectedHoleSearch = 0;
                        break;
233                 }

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

239                     //Assign the current nodes next node to ^.
                        connectedHolesPointer->nextNode = nextNode;
241                 }

243             }
                else{
245
                    //Look at next node.
247                 connectedHolesPointer = connectedHolesPointer->nextNode;
                }
249
            }
251         else{
                //look at next node.
253             connectedHolesPointer = connectedHolesPointer->nextNode;

255         }

257     }

259 }
    /*
261  * Inputs: void
     * Outputs: void
263  * Description: Outputs all nodes.
     */
265 void output(){
        Node *point = HEAD;
267     int loop = 1;

269     while (loop ==1){
            //if next node is null, at the end of the linked list.
271         if (!point->nextNode){
                loop = 0;
273             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);
275             break;
            }else{
277             //Output information about a node.
                printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
                    %d\n\n",
```

```
279                     (int)point->size,point->free,(point->memory-sizeof(Node)),
                            point->memory);
                //look at next node.
281             point = point->nextNode;
            }
283
        }
285 }

287

289 int main() {

291     //TEST HARNESS FOR EACH ALGORITHM COMMENTED OUT.

293     //Set the initial heap size.
        size_t size = 1024;
295
        //Allocate memory for the heap.
297     void *memory = malloc(size);

299     //Algorithm test harness
        initialise(memory,size);
301
        void *x = allocate(100);
303     printf("allocate x output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)x);
        output();
305
        void *y = allocate(50);
307     printf("allocate y output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)y);
        output();
309
        void *z = allocate(100);
311     printf("allocate z output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)z);
        output();
313
        void *t = allocate(100);
315     printf("allocate t output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)z);
        output();
317
        deallocate(x);
319     deallocate(z);

321     printf("De-allocate x & z: \n================================\n\nVisual Node
           output:\n\n");
        output();
323
        void *test1 = allocate(70);
325     printf("test allocation of 70: \n================================\n\nVisual
           Node output:\n\n");
        output();
327
        printf("Allocate all nodes.\n");
329     void *test2 = allocate(10);
        void *test3 = allocate(100);
331     void *test4 = allocate(574);
        output();
333
        printf("Try to allocate when all nodes are not free.\n");
```

```
335     void *test5 = allocate(50);

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

343     //test invalid values.
        void *test6 = allocate(-5);
345
        free(memory);
347     return EXIT_SUCCESS;

349 }
```

part2.h

```
1
#ifndef CW2_PART2_H
3 #define CW2_PART2_H

5 typedef struct NodeStruct{

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

13 }Node;


15
void*(*allocate)(size_t);
17 void deallocate ( void * memory );
void initialise ( void * memory , size_t size, char* algorithm);
19
void *firstFit(size_t bytes);
21 void *nextFit(size_t bytes);
void *bestFit(size_t bytes);
23 void *worstFit(size_t bytes);

25 #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);
```

```
99
                //Return the memory address of the allocated node.
101             return returnNode->memory;
            }
103
            //if the node is the exact size.
105         else if (nodePointer->free == 1 && nodePointer->size == bytes) {

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

109             //Set free to 0 as node is taken.
                nodePointer->free = 0;
111
                //Return the memory address of the allocated node.
113             return nodePointer->memory;

115         }
            //if the node is taken.
117         else {
                //If the next node is NULL.
119             if (nodePointer->nextNode == NULL) {
                    printf("!!!! No free nodes, returning NULL !!!!\n");
121
                    //Return NULL as no valid nodes found.
123                 return NULL;
                }
125             //Look at next node, and continue in the loop.
                nodePointer = nodePointer->nextNode;
127         }
        }
129
    }
131

133 /*
     * Inputs: size_t bytes.
135  * Outputs: (void*) ->memory. Memory address of allocated memory.
     * Description: Uses the NextFit algorithm to allocate memory from the heap.
137  */
    void* nextFit(size_t bytes){
139
        //Only allow valid amount of bytes.
141     if ((int)bytes <= 0){
            printf("Invalid amount of bytes. Returning NULL\n");
143         return NULL;
        }
145
        //Set the initial start location to the NEXTFITNODE node.
147     Node *nodePointer = NEXTFITNODE;

149     //Set the end node to the start node, therefore it will only loop once and
            not infinitely.
        Node *limitNode = NEXTFITNODE;
151
        int allocateBool = 1;
153
        //Loop through every node.
155     while (allocateBool == 1) {

157         //If the node is free and is large enough for the allocated bytes.
            if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
                {
159
```

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

163         //update NEXTFITNODE.
            NEXTFITNODE = returnNode->nextNode;
165
            //return the allocated nodes memory address.
167         return nodePointer->memory;
        }
169
            //if the node is the exact size.
171     else if (nodePointer->free == 1 && nodePointer->size == bytes) {

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

177         //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
179             NEXTFITNODE = nodePointer->nextNode;
            }
181         else{
                NEXTFITNODE = HEAD;
183         }

185         //return the allocated nodes memory address.
            return nodePointer->memory;
187
        }
189     //if the node is taken.
        else {
191
            //Look at next node.
193         if (nodePointer->nextNode == NULL) {

195             //loop through whole list
                nodePointer = HEAD;
197         }
            else{
199             nodePointer = nodePointer->nextNode;
            }
201         if(nodePointer->memory == limitNode->memory){

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

205             //update NEXTFITNODE.
                NEXTFITNODE= HEAD;
207
                //Return NULL as no valid nodes found.
209             return  NULL;
            }
211

213     }
    }
215 }

217
    /*
219  * Inputs: size_t bytes.
     * Outputs: (void*) ->memory. Memory address of allocated memory.
221  * Description: Uses the BestFit algorithm to allocate memory from the heap.
     */
```

```
223 void* bestFit(size_t bytes){
        //Only allow valid amount of bytes.
225     if ((int)bytes <= 0){
            printf("Invalid amount of bytes. Returning NULL\n");
227         return NULL;
        }
229
        //Set the initial node to the HEAD node.
231     //Loop through all the nodes and find the best node that fits.
        int found = 0;
233     int bestFit = 1;
        Node *bestNode = NULL;
235     Node *nodePointer = HEAD;

237     while(bestFit){
            if(nodePointer->free){
239
                //IF the node is of exact size.
241             if (nodePointer->size == bytes){
                    bestNode = nodePointer;
243                 found = 1;
                    break;
245             }

247             //IF the node is greater than bytes
                if(nodePointer->size > (bytes + (sizeof(Node)))){
249                 found = 1;
                    if(bestNode == NULL){
251                     bestNode = nodePointer;
                    }
253                 if(nodePointer->size < bestNode->size){
                        bestNode = nodePointer;
255                 }

257             }
                //If there is enough size to create a new node but not assign a size
                   > 0 to it.
259             if(nodePointer->size == (bytes +(sizeof(Node)))){
                    printf("Invalid node to use. searching next\n");
261
                }
263             //If there is a next node.
                if (nodePointer->nextNode != NULL) {
265                 nodePointer = nodePointer->nextNode;
                }
267             else{
                    if(bestNode == NULL){
269                     bestNode = nodePointer;
                    }
271                 break;

273             }
            }
275         //Look at next node.
            else{
277             if (nodePointer->nextNode != NULL) {
                    nodePointer = nodePointer->nextNode;
279             }
                else{
281                 printf("!!!! No free nodes, returning NULL !!!!\n");

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

```
285             }
            }
287

289     }

291     //Allocate the best node found with the desired bytes.
        if(found) {
293         //If the node is free and is large enough for the allocated bytes.
            if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {
295
                //Call the allocate function to allocate the node.
297             Node* returnNode = allocateNodeWithHole(bestNode,bytes);

299             //return the allocated nodes memory address.
                return returnNode->memory;
301         }


303
            //if the node is the exact size.
305         else if (bestNode->free == 1 && bestNode->size == bytes) {

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

311             //return the allocated nodes memory address.
                return bestNode->memory;
313         }

315     }
        //If no suitable nodes found, return NULL.
317     else {
            printf("!!!! No free nodes, returning NULL !!!!\n");
319         //Return NULL as no valid nodes found.
            return NULL;
321
        }
323


325
    }
327


329 /*
     * Inputs: size_t bytes.
331  * Outputs: (void*) ->memory. Memory address of allocated memory.
     * Description: Uses the WorstFit algorithm to allocate memory from the heap.
333  */
    void* worstFit(size_t bytes) {
335     //Only allow valid amount of bytes.
        if ((int)bytes <= 0){
337         printf("Invalid amount of bytes. Returning NULL\n");
            return NULL;
339     }


341
        //Set the initial node to the HEAD node.
343     //Loop through all the nodes and find the worst node that fits.
        int found = 0;
345     int worstFit = 1;
        Node *worstNode = NULL;
347     Node *nodePointer = HEAD;
```

```
349     while (worstFit) {
            if (nodePointer->free) {
351
                //IF the node is of exact size.
353             if (nodePointer->size == bytes) {
                    found = 1;
355                 if (worstNode == NULL) {
                        worstNode = nodePointer;
357                 } else if (nodePointer->size > worstNode->size) {
                        worstNode = nodePointer;
359                 }

361             }

                    //IF the node is greater than bytes
363             else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                    found = 1;
365                 if (worstNode == NULL) {
                        worstNode = nodePointer;
367                 }
                    if (nodePointer->size > worstNode->size) {
369                     worstNode = nodePointer;
                    }

371             } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                    printf("Invalid node to use. searching next\n");
373
                }
375
            }
377             if (nodePointer->nextNode != NULL) {
                    nodePointer = nodePointer->nextNode;
379             } else {
                    break;
381             }

383     }

385     if(found) {

387         //If the node is free and is large enough for the allocated bytes.
            if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
389
                //Call the allocate function to allocate the node.
391             Node* returnNode = allocateNodeWithHole(worstNode,bytes);

393             //return the allocated nodes memory address.
                return returnNode->memory;
395         }


397
            //if the node is the exact size.
399         else if (worstNode->free == 1 && worstNode->size == bytes) {
                printf("found node of exact size.\n");
401             //Set free to 0 as node is taken.
                worstNode->free = 0;

403             //return the allocated nodes memory address.
                return worstNode->memory;
405
            }
```

```
411
        }
413     //If not found.
        else {
415         printf("!!!! No free nodes, returning NULL !!!!\n");

417         return NULL;

419     }
    }
421

423 /*
    * Inputs: void* memory, size_t size, char* algorithm.
425 * Outputs: void.
    * Description: Initialises the type of allocation algorithm and creates the
        initial HEAD node.
427 */
    void initialise(void *memory ,size_t size, char* algorithm){
429
        //Allocate (size) amount of memory to the heap (called memory in this case).
431     //Returns pointer to the start of the address.

433     //Allocate the head node the memory it needs.
        HEAD = (Node*)(memory);
435

437     //Assign type of algorithm

439     if(strcmp(algorithm,"NextFit")==0){
            printf("Using NextFit algorithm.\n");
441         allocate = nextFit;
            NEXTFITNODE = HEAD;
443
        }
445     else if(strcmp(algorithm, "BestFit")==0){
            printf("Using BestFit algorithm.\n");
447         allocate = bestFit;

449     }
        else if(strcmp(algorithm,"WorstFit")==0){
451         printf("Using WorstFit algorithm.\n");
            allocate = worstFit;
453
        }
455     //Default first fit
        else{
457         printf("Using FirstFit algorithm.\n");
            allocate = firstFit;
459
        }
461

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

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

```
473
}
475
/*
477  * Inputs: void* memory.
 * Outputs: void.
479  * Description: Deallocates a node (frees it) based on the input of its memory
     address.
 */
481 void deallocate ( void * memory ){

483     //Assign new node pointer to HEAD node.
    Node *nodePointer = HEAD;
485
    //int to act as bool for while loop.
487     int deallocateBool = 1;


489
    while(deallocateBool == 1){
491
        //if the memory address of dealocation equals the memory address of a
            node.
493
        if(nodePointer->memory != memory){
495
            //if the next node is null, current node is the last in the linked
                list.
497             if (nodePointer->nextNode == NULL){
                deallocateBool = 0;
499                 break;
            }
501             else {

503                 //look at next node.
                nodePointer = nodePointer->nextNode;
505             }
        }
507
            //else the node is the memory address.
509         else{
            //Set the nodes free variable to 1, as node has been deallocated.
511             printf("De-allocation successful\n");
            nodePointer->free=1;
513             deallocateBool = 0;
            break;
515         }
    }
517
    //Next part coalaces connected holes.
519
    //int to act as bool for while loop.
521     int connectedHoleSearch = 1;

523     //Assign a new node pointer to the HEAD node.
    Node *connectedHolesPointer = HEAD;
525


527     while (connectedHoleSearch == 1){

529         //if the node is free.
        if (connectedHolesPointer == NULL){
531             break;
        }
```

```
533         if (connectedHolesPointer->free == 1){

535             //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){
537
                //set the loop bool to 0.
539             connectedHoleSearch = 0;
                break;
541         }

543         //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {
545
                //Increase the current nodes size, to that containing both nodes.
547             connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;
549
                //If NextFit algorithm is set, handle the pointer to it.
551             if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
553                     printf("Moving NEXTFITNODE due to coalace taking place.\n
                           ");
                        NEXTFITNODE = connectedHolesPointer;
555                 }
                 }
557
                //if the next next node is NULL
559             if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
561                 Node *nextNode = connectedHolesPointer->nextNode->nextNode;

563                 //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
565                 connectedHoleSearch = 0;
                    break;
567             }

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

573                 //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
575             }

577         }
            else{
579
                //Look at next node.
581             connectedHolesPointer = connectedHolesPointer->nextNode;
            }
583
        }
585     else{
            //look at next node.
587         connectedHolesPointer = connectedHolesPointer->nextNode;

589     }

591 }

593 }
```

```
/*
 * 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);

/*
```

```
37   * Inputs: Node* nodePointer, size_t bytes.
     * Outputs: Node* (return node).
39   * Description: Function takes a node, allocates a size to it, and creates a free
         node next to it with the remaining space.
     */
41 Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
       printf("Found node of suitable size, creating, allocating space and creating
           node\n");
43
       //Allocate memory for the struct hole node at the current nodes memory plus a
            struct.
45     Node *nextNode = (Node *) (nodePointer->memory + bytes);

47     //memory address at the start of the empty hole node.
       void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));
49
       //Assign the hole nodes variables.
51     nextNode->memory = memoryStart2;
       nextNode->free = 1;
53     nextNode->size = nodePointer->size - bytes - sizeof(Node);
       nextNode->nextNode = nodePointer->nextNode;
55     nextNode->prevNode = nodePointer;

57     //allocate the taken size.
       nodePointer->size = bytes;
59
       //Set free bool to 0 as this node is taken.
61     nodePointer->free = 0;

63     //Point the current node to the next node which is the created hole
       nodePointer->nextNode = nextNode;
65
       //return the taken nodes memory address.
67     return nodePointer;
   }
69


71 /*
     * Inputs: size_t bytes.
73   * Outputs: (void*) ->memory. Memory address of allocated memory.
     * Description: Uses the FirstFit algorithm to allocate memory from the heap.
75   */
   void *firstFit(size_t bytes) {
77     //Only allow valid amount of bytes.
       if ((int)bytes <= 0){
79         printf("Invalid amount of bytes. Returning NULL\n");
           return NULL;
81     }


83
       //Set the initial start location to the HEAD node.
85     Node *nodePointer = HEAD;

87     int allocateBool = 1;

89     //Loop through every node.
       while (allocateBool == 1) {
91
           //If the node is free and is large enough for the allocated bytes.
93         if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
               {

95             //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.
```

```
        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");
```

```
283                 //Return NULL as no valid nodes found.
                    return NULL;
285             }
        }
287

289     }

291     //Allocate the best node found with the desired bytes.
        if(found) {
293         //If the node is free and is large enough for the allocated bytes.
            if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {
295
                //Call the allocate function to allocate the node.
297             Node* returnNode = allocateNodeWithHole(bestNode,bytes);

299             //return the allocated nodes memory address.
                return returnNode->memory;
301         }


303
            //if the node is the exact size.
305         else if (bestNode->free == 1 && bestNode->size == bytes) {

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

311             //return the allocated nodes memory address.
                return bestNode->memory;
313         }

315     }
        //If no suitable nodes found, return NULL.
317     else {
            printf("!!!! No free nodes, returning NULL !!!!\n");
319         //Return NULL as no valid nodes found.
            return NULL;
321
        }
323


325
    }
327

329 /*
     * Inputs: size_t bytes.
331  * Outputs: (void*) ->memory. Memory address of allocated memory.
     * Description: Uses the WorstFit algorithm to allocate memory from the heap.
333  */
    void* worstFit(size_t bytes) {
335     //Only allow valid amount of bytes.
        if ((int)bytes <= 0){
337         printf("Invalid amount of bytes. Returning NULL\n");
            return NULL;
339     }


341
        //Set the initial node to the HEAD node.
343     //Loop through all the nodes and find the worst node that fits.
        int found = 0;
```

```
345     int worstFit = 1;
        Node *worstNode = NULL;
347     Node *nodePointer = HEAD;

349     while (worstFit) {
            if (nodePointer->free) {
351
                //IF the node is of exact size.
353             if (nodePointer->size == bytes) {
                    found = 1;
355                 if (worstNode == NULL) {
                        worstNode = nodePointer;
357                 } else if (nodePointer->size > worstNode->size) {
                        worstNode = nodePointer;
359                 }

361             }

363                 //IF the node is greater than bytes
                else if (nodePointer->size > (bytes + (sizeof(Node)))) {
365                 found = 1;
                    if (worstNode == NULL) {
367                     worstNode = nodePointer;
                    }
369                 if (nodePointer->size > worstNode->size) {
                        worstNode = nodePointer;
371                 }

373             } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                    printf("Invalid node to use. searching next\n");
375
                }
377
            }
379             if (nodePointer->nextNode != NULL) {
                    nodePointer = nodePointer->nextNode;
381             } else {
                    break;
383             }

385     }

387     if(found) {

389         //If the node is free and is large enough for the allocated bytes.
            if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
391
                //Call the allocate function to allocate the node.
393             Node* returnNode = allocateNodeWithHole(worstNode,bytes);

395             //return the allocated nodes memory address.
                return returnNode->memory;
397         }


399


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

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

```
            return worstNode->memory;
409
        }
411
    }
    //If not found.
    else {
415     printf("!!!! No free nodes, returning NULL !!!!\n");

417     return NULL;

419 }
}
421

423 /*
* Inputs: void* memory, size_t size, char* algorithm.
425 * Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the
    initial HEAD node.
427 */
void initialise(void *memory ,size_t size, char* algorithm){
429
    //Allocate (size) amount of memory to the heap (called memory in this case).
431 //Returns pointer to the start of the address.

433 //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);
435

437 //Assign type of algorithm

439 if(strcmp(algorithm,"NextFit")==0){
        printf("Using NextFit algorithm.\n");
441     allocate = nextFit;
        NEXTFITNODE = HEAD;
443
    }
445 else if(strcmp(algorithm, "BestFit")==0){
        printf("Using BestFit algorithm.\n");
447     allocate = bestFit;

449 }
    else if(strcmp(algorithm,"WorstFit")==0){
451     printf("Using WorstFit algorithm.\n");
        allocate = worstFit;
453
    }
455 //Default first fit
    else{
457     printf("Using FirstFit algorithm.\n");
        allocate = firstFit;
459
    }
461

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

467 //Assign all the nodes variables.
    HEAD->size = size - sizeof(Node);
469 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){
531         break;
        }
533     if (connectedHolesPointer->free == 1){

535         //if the node is null.
            if (connectedHolesPointer->nextNode == NULL){
537
                //set the loop bool to 0.
539             connectedHoleSearch = 0;
                break;
541         }

543         //if the next node is free.
            if (connectedHolesPointer->nextNode->free == 1) {
545
                //Increase the current nodes size, to that containing both nodes.
547             connectedHolesPointer->size = connectedHolesPointer->size +
                        sizeof(Node)+connectedHolesPointer->nextNode->size;
549
                //If NextFit algorithm is set, handle the pointer to it.
551             if (NEXTFITNODE != NULL){
                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                       memory){
553                     printf("Moving NEXTFITNODE due to coalace taking place.\n
                           ");
                        NEXTFITNODE = connectedHolesPointer;
555                 }
                 }
557
                //if the next next node is NULL
559             if (connectedHolesPointer->nextNode->nextNode == NULL){
                    //Assign a new node pointer to current nodes next next node.
561                 Node *nextNode = connectedHolesPointer->nextNode->nextNode;

563                 //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
565                 connectedHoleSearch = 0;
                    break;
567             }

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

573                 //Assign the current nodes next node to ^.
                    connectedHolesPointer->nextNode = nextNode;
575             }

577         }
            else{
579
                //Look at next node.
581             connectedHolesPointer = connectedHolesPointer->nextNode;
            }
583
        }
585     else{
            //look at next node.
587         connectedHolesPointer = connectedHolesPointer->nextNode;

589     }
```

```
591     }

593 }
    /*
595  * Inputs: void
     * Outputs: void
597  * Description: Outputs all nodes.
     */
599 void output(){
        Node *point = HEAD;
601     int loop = 1;

603     while (loop ==1){
            //if next node is null, at the end of the linked list.
605         if (!point->nextNode){
                loop = 0;
607             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);
609             break;
            }else{
611             //Output information about a node.
                printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
                    %d\n\n",
613                     (int)point->size,point->free,(point->memory-sizeof(Node)),
                            point->memory);
                //look at next node.
615             point = point->nextNode;
            }
617
        }
619 }


621


623 int main() {
        /*
625      * TEST HARNESS FOR EACH ALGORITHM COMMENTED OUT.
         */
627
        //Set the initial heap size.
629     size_t size = 1024;

631     //Allocate memory for the heap.
        void *memory = malloc(size);
633
        //FirstFit Algorithm test harness
635     //The test allocation should go into node x.

637     char * algo = "FirstFit";
        initialise(memory,size,algo);
639
        void *x = allocate(100);
641     printf("allocate x output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)x);
        output();
643
        void *y = allocate(50);
645     printf("allocate y output: %d\n================================\n\nVisual Node
            output:\n\n",(char*)y);
        output();
647
```

```
    void *z = allocate(100);
649 printf("allocate z output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)z);
    output();
651
    void *t = allocate(100);
653 printf("allocate t output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)z);
    output();
655
    deallocate(x);
657 deallocate(z);

659 printf("De-allocate x & z: \n================================\n\nVisual Node
       output:\n\n");
    output();
661
    void *test = allocate(70);
663 printf("test allocation of 70: \n================================\n\nVisual
       Node output:\n\n");
    output();
665


667
    //NextFit Algorithm test harness
669 //The test allocation should go into node last node, as that is the
       NEXTFITNODE due to the algorithm.
    /*
671 char * algo = "NextFit";
    initialise(memory,size,algo);
673
    void *x = allocate(100);
675 printf("allocate x output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)x);
    output();
677
    void *y = allocate(50);
679 printf("allocate y output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)y);
    output();
681
    void *z = allocate(100);
683 printf("allocate z output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)z);
    output();
685
    void *t = allocate(100);
687 printf("allocate t output: %d\n================================\n\nVisual Node
        output:\n\n",(char*)z);
    output();
689
    deallocate(x);
691 deallocate(z);

693 printf("De-allocate x & z: \n================================\n\nVisual Node
       output:\n\n");
    output();
695
    void *test = allocate(70);
697 printf("test allocation of 70: \n================================\n\nVisual
       Node output:\n\n");
    output();
699 */
```

```
701
        //BestFit Algorithm Test Harness
703     //test1 should go in x, test2 should go in z.
        /*
705     char * algo = "BestFit";
        initialise(memory,size,algo);
707
        void *x = allocate(50);
709     void *y = allocate(100);
        void *z = allocate(200);
711     void *b = allocate(30);
        printf("Allocate 4 nodes: \n================================\n\nVisual Node
            output:\n\n");
713     output();

715     deallocate(x);
        deallocate(z);
717
        printf("De-allocate x & z: \n================================\n\nVisual Node
            output:\n\n");
719     output();

721     //Should replace node x;
        void *test1 = allocate(50);
723     printf("test allocation of 50: \n================================\n\nVisual
            Node output:\n\n");
        output();
725
        deallocate(test1);
727     printf("De-allocation of testAllocation: \n================================\n\
            nVisual Node output:\n\n");
        output();
729
        //Should use node z, and create new hole:
731     void *test2 = allocate(150);
        printf("test2 of 150: \n================================\n\nVisual Node output
            :\n\n");
733     output();
         */
735

737     //WorstFit Algorithm Test Harness
        //test1 should go in the hole after b, and so should test2
739     /*
        char * algo = "WorstFit";
741     initialise(memory,size,algo);

743     void *x = allocate(50);
        void *y = allocate(100);
745     void *z = allocate(200);
        void *b = allocate(30);
747     printf("Allocate 4 nodes: \n================================\n\nVisual Node
            output:\n\n");
        output();
749
        deallocate(x);
751     deallocate(z);

753     printf("De-allocate x & z: \n================================\n\nVisual Node
            output:\n\n");
        output();
755
```

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

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

765     //Should use node z, and create new hole:
        void *test2 = allocate(150);
767     printf("test allocation of 150: \n================================\n\nVisual
            Node output:\n\n");
        output();
769      */
        free(memory);
771     return EXIT_SUCCESS;

773 }
```

part3.h

```
1
#ifndef CW2_PART3_H
3 #define CW2_PART3_H

5 typedef struct NodeStruct{

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

13 }Node;


15
void*(*allocate)(size_t);
17 void deallocate ( void * memory );
void initialise ( void * memory , size_t size, char* algorithm);
19
void *firstFit(size_t bytes);
21 void *nextFit(size_t bytes);
void *bestFit(size_t bytes);
23 void *worstFit(size_t bytes);

25 #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.
```

```
 * 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;
```

```
99      //Loop through every node.
        while (allocateBool == 1) {
101
            //If the node is free and is large enough for the allocated bytes.
103         if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
                {
105             //Call the allocate function to allocate the node.
                Node* returnNode = allocateNodeWithHole(nodePointer,bytes);
107
                printf("%d bytes allocated by thread %d.\n",bytes,pthread_self());
109
                //Unlock the Mutex.
111             pthread_mutex_unlock(&mutex);

113             //Return the memory address of the allocated node.
                return returnNode->memory;
115         }

117         //if the node is the exact size.
            else if (nodePointer->free == 1 && nodePointer->size == bytes) {
119
                printf("thread: %d: found node of exact size.\n",pthread_self());
121
                //Set free to 0 as node is taken.
123             nodePointer->free = 0;

125             //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);
127
                //Return the memory address of the allocated node.
129             return nodePointer->memory;

131         }
            //if the node is taken.
133         else {
                //If the next node is NULL.
135             if (nodePointer->nextNode == NULL) {
                    printf("!!!! No free nodes, returning NULL !!!!\n");
137
                    //Unlock Mutex
139                 pthread_mutex_unlock(&mutex);

141                 //Return NULL as no valid nodes found.
                    return NULL;
143             }
                //Look at next node, and continue in the loop.
145             nodePointer = nodePointer->nextNode;
            }
147     }

149 }

151
    /*
153  * Inputs: size_t bytes.
     * Outputs: (void*) ->memory. Memory address of allocated memory.
155  * Description: Uses the NextFit algorithm to allocate memory from the heap.
     */
157 void* nextFit(size_t bytes){

159     //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.
```

```
219             return nodePointer->memory;

221         }
            //if the node is taken.
223         else {

225             //Look at next node.
            if (nodePointer->nextNode == NULL) {
227
                //loop through whole list
229             nodePointer = HEAD;
            }
231         else{
                nodePointer = nodePointer->nextNode;
233         }
            if(nodePointer->memory == limitNode->memory){
235
                printf("!!!! No free nodes, returning NULL !!!!\n");
237
                //update NEXTFITNODE.
239             NEXTFITNODE= HEAD;

241             //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);
243
                //Return NULL as no valid nodes found.
245             return  NULL;
            }
247


249         }
    }
251 }


253
    /*
255  * Inputs: size_t bytes.
     * Outputs: (void*) ->memory. Memory address of allocated memory.
257  * Description: Uses the BestFit algorithm to allocate memory from the heap.
     */
259 void* bestFit(size_t bytes){

261     //Used to stop the function allocating 0 bytes to a node if the calling
           thread requests it, if not handled
        // node would be invalid.
263     if((int)bytes <= 0){
            printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
               NULL.!!!\n",pthread_self());
265         return NULL;
        }
267
        printf("\n %d: attempting to gain lock.\n",pthread_self());
269
        //Thread attempts to lock the mutex.
271     pthread_mutex_lock(&mutex);

273     //When the thread gains the lock, attempt to allocate the input bytes.
        printf("\n %d: acquired lock\n",pthread_self());
275
        //Set the initial node to the HEAD node.
277     //Loop through all the nodes and find the best node that fits.
        int found = 0;
279     int bestFit = 1;
```

```
    Node *bestNode = NULL;
281 Node *nodePointer = HEAD;

283 while(bestFit){
        if(nodePointer->free){
285
            //IF the node is of exact size.
287         if (nodePointer->size == bytes){
                bestNode = nodePointer;
289             found = 1;
                break;
291         }

293         //IF the node is greater than bytes
            if(nodePointer->size > (bytes + (sizeof(Node)))){
295             found = 1;
                if(bestNode == NULL){
297                 bestNode = nodePointer;
                }
299             if(nodePointer->size < bestNode->size){
                    bestNode = nodePointer;
301             }

303         }
            //If there is enough size to create a new node but not assign a size
                > 0 to it.
305         if(nodePointer->size == (bytes +(sizeof(Node)))){
                printf("Invalid node to use. searching next\n");
307
            }
309         //If there is a next node.
            if (nodePointer->nextNode != NULL) {
311             nodePointer = nodePointer->nextNode;
            }
313         else{
                if(bestNode == NULL){
315                 bestNode = nodePointer;
                }
317             break;

319         }
        }
321     //Look at next node.
        else{
323         if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
325         }
            else{
327             printf("!!!! No free nodes, returning NULL !!!!\n");
                //Unlock the Mutex.
329             pthread_mutex_unlock(&mutex);

331             //Return NULL as no valid nodes found.
                return NULL;
333         }
        }
335

337 }

339 //Allocate the best node found with the desired bytes.
    if(found) {
341     //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());
```

```
403     //Thread attempts to lock the mutex.
        pthread_mutex_lock(&mutex);
405
        //When the thread gains the lock, attempt to allocate the input bytes.
407
        printf("\n %d: acquired lock\n",pthread_self());
409
        //Set the initial node to the HEAD node.
411     //Loop through all the nodes and find the worst node that fits.
        int found = 0;
413     int worstFit = 1;
        Node *worstNode = NULL;
415     Node *nodePointer = HEAD;

417     while (worstFit) {
            if (nodePointer->free) {
419
                //IF the node is of exact size.
421             if (nodePointer->size == bytes) {
                    found = 1;
423                 if (worstNode == NULL) {
                        worstNode = nodePointer;
425                 } else if (nodePointer->size > worstNode->size) {
                        worstNode = nodePointer;
427                 }

429             }

431                 //IF the node is greater than bytes
                else if (nodePointer->size > (bytes + (sizeof(Node)))) {
433                 found = 1;
                    if (worstNode == NULL) {
435                     worstNode = nodePointer;
                    }
437                 if (nodePointer->size > worstNode->size) {
                        worstNode = nodePointer;
439                 }

441             } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
                    printf("Invalid node to use. searching next\n");
443
                }
445
            }
447             if (nodePointer->nextNode != NULL) {
                    nodePointer = nodePointer->nextNode;
449             } else {
                    break;
451             }

453     }

455     if(found) {

457         //If the node is free and is large enough for the allocated bytes.
            if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
459
                //Call the allocate function to allocate the node.
461             Node* returnNode = allocateNodeWithHole(worstNode,bytes);

463             //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);
465
```

```
            //return the allocated nodes memory address.
467         return returnNode->memory;
        }
469


471
        //if the node is the exact size.
473     else if (worstNode->free == 1 && worstNode->size == bytes) {
            printf("found node of exact size.\n");
475         //Set free to 0 as node is taken.
            worstNode->free = 0;
477
            //Unlock the Mutex.
479         pthread_mutex_unlock(&mutex);


481
            //return the allocated nodes memory address.
483         return worstNode->memory;

485     }

487 }
    //If not found.
489 else {
        printf("!!!! No free nodes, returning NULL !!!!\n");
491
        //Unlock the Mutex.
493     pthread_mutex_unlock(&mutex);

495     return NULL;

497 }
}
499


501 /*
* Inputs: void* memory, size_t size, char* algorithm.
503 * Outputs: void.
* Description: Initialises the type of allocation algorithm and creates the
    initial HEAD node.
505 */
void initialise(void *memory ,size_t size, char* algorithm){
507
    //Allocate (size) amount of memory to the heap (called memory in this case).
509 //Returns pointer to the start of the address.

511 //Allocate the head node the memory it needs.
    HEAD = (Node*)(memory);
513


515 //Assign type of algorithm

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

521 }
    else if(strcmp(algorithm, "BestFit")==0){
523     allocate = bestFit;

525 }
    else if(strcmp(algorithm,"WorstFit")==0){
527     allocate = worstFit;
```

```
529     }
        //Default first fit
531     else{
            allocate = firstFit;
533
        }
535

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

541     //Assign all the nodes variables.
        HEAD->size = size - sizeof(Node);
543     HEAD->memory = memoryStart;
        HEAD->free = 1;
545     HEAD->nextNode = NULL;
        HEAD->prevNode = NULL;
547
}
549
/*
551  * Inputs: void* memory.
     * Outputs: void.
553  * Description: Deallocates a node (frees it) based on the input of its memory
         address.
     */
555 void deallocate ( void * memory ){

557     //Thread attempts to lock the mutex.
        pthread_mutex_lock(&mutex);
559
        //Assign new node pointer to HEAD node.
561     Node *nodePointer = HEAD;

563     //int to act as bool for while loop.
        int deallocateBool = 1;
565

567     while(deallocateBool == 1){

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

571         if(nodePointer->memory != memory){

573             //if the next node is null, current node is the last in the linked
                    list.
                if (nodePointer->nextNode == NULL){
575                 deallocateBool = 0;
                    break;
577             }
                else {
579
                    //look at next node.
581                 nodePointer = nodePointer->nextNode;
                }
583         }

585             //else the node is the memory address.
            else{
587             //Set the nodes free variable to 1, as node has been deallocated.
```

```
            printf("De-allocation successful of thread ID: %d.\n",pthread_self())
                ;
589         nodePointer->free=1;
            deallocateBool = 0;
591         break;
        }
593     }

595     //Next part coalaces connected holes.

597     //int to act as bool for while loop.
        int connectedHoleSearch = 1;
599
        //Assign a new node pointer to the HEAD node.
601     Node *connectedHolesPointer = HEAD;


603
        while (connectedHoleSearch == 1){
605
            //if the node is free.
607         if (connectedHolesPointer == NULL){
                break;
609         }
            if (connectedHolesPointer->free == 1){
611
                //if the node is null.
613             if (connectedHolesPointer->nextNode == NULL){

615                 //set the loop bool to 0.
                    connectedHoleSearch = 0;
617                 break;
                }
619
                //if the next node is free.
621             if (connectedHolesPointer->nextNode->free == 1) {

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

627                 //If NextFit algorithm is set, handle the pointer to it.
                    if (NEXTFITNODE != NULL){
629                     if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                            memory){
                            printf("Moving NEXTFITNODE due to coalace taking place.\n
                                ");
631                         NEXTFITNODE = connectedHolesPointer;
                        }
633                  }

635                 //if the next next node is NULL
                    if (connectedHolesPointer->nextNode->nextNode == NULL){
637                     //Assign a new node pointer to current nodes next next node.
                        Node *nextNode = connectedHolesPointer->nextNode->nextNode;
639
                        //Assign the current nodes next node to ^.
641                     connectedHolesPointer->nextNode = nextNode;
                        connectedHoleSearch = 0;
643                     break;
                    }
645
                    else {
647                     //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);
    }
}
```

5013ACW02-100237137-file2.c

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

`part3_test.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);
```

```
38 /*
    * Inputs: Node* nodePointer, size_t bytes.
40  * 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.
42  */
   Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
44      printf("Found node of suitable size, creating, allocating space and creating
           node\n");

46      //Allocate memory for the struct hole node at the current nodes memory plus a
            struct.
        Node *nextNode = (Node *) (nodePointer->memory + bytes);
48
        //memory address at the start of the empty hole node.
50      void *memoryStart2 = (((char *) nodePointer->memory) + bytes + sizeof(Node));

52      //Assign the hole nodes variables.
        nextNode->memory = memoryStart2;
54      nextNode->free = 1;
        nextNode->size = nodePointer->size - bytes - sizeof(Node);
56      nextNode->nextNode = nodePointer->nextNode;
        nextNode->prevNode = nodePointer;
58
        //allocate the taken size.
60      nodePointer->size = bytes;

62      //Set free bool to 0 as this node is taken.
        nodePointer->free = 0;
64
        //Point the current node to the next node which is the created hole
66      nodePointer->nextNode = nextNode;

68      //return the taken nodes memory address.
        return nodePointer;
70  }


72
   /*
74  * Inputs: size_t bytes.
    * Outputs: (void*) ->memory. Memory address of allocated memory.
76  * Description: Uses the FirstFit algorithm to allocate memory from the heap.
    */
78 void *firstFit(size_t bytes) {

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

88      //Thread attempts to lock the mutex.
        pthread_mutex_lock(&mutex);
90
        //When the thread gains the lock, attempt to allocate the input bytes.
92      printf("\n %d: acquired lock\n",pthread_self());

94      //Set the initial start location to the HEAD node.
```

```
    Node *nodePointer = HEAD;
96
    int allocateBool = 1;
98
    //Loop through every node.
100 while (allocateBool == 1) {
102     //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
            {
104
            //Call the allocate function to allocate the node.
106         Node* returnNode = allocateNodeWithHole(nodePointer,bytes);
108         printf("%d bytes allocated by thread %d.\n",bytes,pthread_self());
110         //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);
112
            //Return the memory address of the allocated node.
114         return returnNode->memory;
        }
116
            //if the node is the exact size.
118     else if (nodePointer->free == 1 && nodePointer->size == bytes) {
120         printf("thread: %d: found node of exact size.\n",pthread_self());
122         //Set free to 0 as node is taken.
            nodePointer->free = 0;
124
            //Unlock the Mutex.
126         pthread_mutex_unlock(&mutex);
128         //Return the memory address of the allocated node.
            return nodePointer->memory;
130
        }
132         //if the node is taken.
        else {
134         //If the next node is NULL.
            if (nodePointer->nextNode == NULL) {
136             printf("!!!! No free nodes, returning NULL !!!!\n");
138             //Unlock Mutex
                pthread_mutex_unlock(&mutex);
140
                //Return NULL as no valid nodes found.
142             return NULL;
            }
144         //Look at next node, and continue in the loop.
            nodePointer = nodePointer->nextNode;
146     }
    }
148
}
150

152 /*
 * Inputs: size_t bytes.
154 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the NextFit algorithm to allocate memory from the heap.
156 */
```

```
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.
```

```
216             pthread_mutex_unlock(&mutex);

218             //return the allocated nodes memory address.
                return nodePointer->memory;
220
        }
222             //if the node is taken.
        else {
224
            //Look at next node.
226         if (nodePointer->nextNode == NULL) {

228             //loop through whole list
                nodePointer = HEAD;
230         }
            else{
232             nodePointer = nodePointer->nextNode;
            }
234         if(nodePointer->memory == limitNode->memory){

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

238             //update NEXTFITNODE.
                NEXTFITNODE= HEAD;
240
                //Unlock the Mutex.
242             pthread_mutex_unlock(&mutex);

244             //Return NULL as no valid nodes found.
                return  NULL;
246         }


248
        }
250 }
}
252


254 /*
 * Inputs: size_t bytes.
256 * Outputs: (void*) ->memory. Memory address of allocated memory.
 * Description: Uses the BestFit algorithm to allocate memory from the heap.
258 */
void* bestFit(size_t bytes){
260
    //Used to stop the function allocating 0 bytes to a node if the calling
        thread requests it, if not handled
262 // node would be invalid.
    if((int)bytes <= 0){
264     printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
            NULL.!!!\n",pthread_self());
        return NULL;
266 }

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

270 //Thread attempts to lock the mutex.
    pthread_mutex_lock(&mutex);
272
    //When the thread gains the lock, attempt to allocate the input bytes.
274 printf("\n %d: acquired lock\n",pthread_self());

276 //Set the initial node to the HEAD node.
```

```
    //Loop through all the nodes and find the best node that fits.
278 int found = 0;
    int bestFit = 1;
280 Node *bestNode = NULL;
    Node *nodePointer = HEAD;
282
    while(bestFit){
284     if(nodePointer->free){

286         //IF the node is of exact size.
            if (nodePointer->size == bytes){
288             bestNode = nodePointer;
                found = 1;
290             break;
            }
292
            //IF the node is greater than bytes
294         if(nodePointer->size > (bytes + (sizeof(Node)))){
                found = 1;
296             if(bestNode == NULL){
                    bestNode = nodePointer;
298             }
                if(nodePointer->size < bestNode->size){
300                 bestNode = nodePointer;
                }
302
            }
304         //If there is enough size to create a new node but not assign a size
               > 0 to it.
            if(nodePointer->size == (bytes +(sizeof(Node)))){
306             printf("Invalid node to use. searching next\n");

308         }
            //If there is a next node.
310         if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
312         }
            else{
314             if(bestNode == NULL){
                    bestNode = nodePointer;
316             }
                break;
318
            }
320     }
            //Look at next node.
322     else{
            if (nodePointer->nextNode != NULL) {
324             nodePointer = nodePointer->nextNode;
            }
326         else{
                printf("!!!! No free nodes, returning NULL !!!!\n");
328             //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);
330
                //Return NULL as no valid nodes found.
332             return NULL;
            }
334     }


336
    }
338
```

```
    //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;
    }
```

```
400
        printf("\n %d: attempting to gain lock.\n",pthread_self());
402
        //Thread attempts to lock the mutex.
404     pthread_mutex_lock(&mutex);

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

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

410     //Set the initial node to the HEAD node.
        //Loop through all the nodes and find the worst node that fits.
412     int found = 0;
        int worstFit = 1;
414     Node *worstNode = NULL;
        Node *nodePointer = HEAD;
416
        while (worstFit) {
418         if (nodePointer->free) {

420             //IF the node is of exact size.
                if (nodePointer->size == bytes) {
422                 found = 1;
                    if (worstNode == NULL) {
424                     worstNode = nodePointer;
                    } else if (nodePointer->size > worstNode->size) {
426                     worstNode = nodePointer;
                    }
428
                }
430
                    //IF the node is greater than bytes
432             else if (nodePointer->size > (bytes + (sizeof(Node)))) {
                    found = 1;
434                 if (worstNode == NULL) {
                        worstNode = nodePointer;
436                 }
                    if (nodePointer->size > worstNode->size) {
438                     worstNode = nodePointer;
                    }
440
                } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
442                 printf("Invalid node to use. searching next\n");

444             }

446         }
            if (nodePointer->nextNode != NULL) {
448             nodePointer = nodePointer->nextNode;
            } else {
450             break;
            }
452
        }
454
        if(found) {
456
            //If the node is free and is large enough for the allocated bytes.
458         if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {

460             //Call the allocate function to allocate the node.
                Node* returnNode = allocateNodeWithHole(worstNode,bytes);
462
```

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

466             //return the allocated nodes memory address.
                return returnNode->memory;
468         }


470

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

478             //Unlock the Mutex.
                pthread_mutex_unlock(&mutex);
480

482             //return the allocated nodes memory address.
                return worstNode->memory;
484

            }
486
        }
488         //If not found.
        else {
490         printf("!!!! No free nodes, returning NULL !!!!\n");

492         //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);
494
            return NULL;
496
        }
498 }


500
    /*
502 * Inputs: void* memory, size_t size, char* algorithm.
    * Outputs: void.
504 * Description: Initialises the type of allocation algorithm and creates the
       initial HEAD node.
    */
506 void initialise(void *memory ,size_t size, char* algorithm){

508     //Allocate (size) amount of memory to the heap (called memory in this case).
        //Returns pointer to the start of the address.
510
        //Allocate the head node the memory it needs.
512     HEAD = (Node*)(memory);


514
        //Assign type of algorithm
516
        if(strcmp(algorithm,"NextFit")==0){
518         allocate = nextFit;
            NEXTFITNODE = HEAD;
520
        }
522     else if(strcmp(algorithm, "BestFit")==0){
            allocate = bestFit;
524
```

```
    }
526     else if(strcmp(algorithm,"WorstFit")==0){
            allocate = worstFit;
528
    }
530         //Default first fit
    else{
532         allocate = firstFit;

534 }


536
    //memoryStart is the memory address where data can be stored.
538 //This is done so the struct data come before it.
    void *memoryStart = ((char*)HEAD) + sizeof(Node);
540
    //Assign all the nodes variables.
542 HEAD->size = size - sizeof(Node);
    HEAD->memory = memoryStart;
544 HEAD->free = 1;
    HEAD->nextNode = NULL;
546 HEAD->prevNode = NULL;

548 }

550 /*
     * Inputs: void* memory.
552  * Outputs: void.
     * Description: Deallocates a node (frees it) based on the input of its memory
        address.
554  */
    void deallocate ( void * memory ){
556
        //Thread attempts to lock the mutex.
558     pthread_mutex_lock(&mutex);

560     //Assign new node pointer to HEAD node.
        Node *nodePointer = HEAD;
562
        //int to act as bool for while loop.
564     int deallocateBool = 1;


566
        while(deallocateBool == 1){
568
            //if the memory address of dealocation equals the memory address of a
                node.
570
            if(nodePointer->memory != memory){
572
                //if the next node is null, current node is the last in the linked
                    list.
574             if (nodePointer->nextNode == NULL){
                    deallocateBool = 0;
576                 break;
                }
578             else {

580                 //look at next node.
                    nodePointer = nodePointer->nextNode;
582             }
            }
584
```

```
            //else the node is the memory address.
586         else{
                //Set the nodes free variable to 1, as node has been deallocated.
588             printf("De-allocation successful of thread ID: %d.\n",pthread_self())
                    ;
                nodePointer->free=1;
590             deallocateBool = 0;
                break;
592         }
        }
594
        //Next part coalaces connected holes.
596
        //int to act as bool for while loop.
598     int connectedHoleSearch = 1;

600     //Assign a new node pointer to the HEAD node.
        Node *connectedHolesPointer = HEAD;
602


604     while (connectedHoleSearch == 1){

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

612             //if the node is null.
                if (connectedHolesPointer->nextNode == NULL){
614
                    //set the loop bool to 0.
616                 connectedHoleSearch = 0;
                    break;
618             }

620             //if the next node is free.
                if (connectedHolesPointer->nextNode->free == 1) {
622
                    //Increase the current nodes size, to that containing both nodes.
624                 connectedHolesPointer->size = connectedHolesPointer->size +
                                                  sizeof(Node)+connectedHolesPointer
                                                     ->nextNode->size;
626
                    //If NextFit algorithm is set, handle the pointer to it.
628                 if (NEXTFITNODE != NULL){
                        if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                           memory){
630                         printf("Moving NEXTFITNODE due to coalace taking place.\n
                               ");
                            NEXTFITNODE = connectedHolesPointer;
632                     }
                    }
634
                    //if the next next node is NULL
636                 if (connectedHolesPointer->nextNode->nextNode == NULL){
                        //Assign a new node pointer to current nodes next next node.
638                     Node *nextNode = connectedHolesPointer->nextNode->nextNode;

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

```
644                 }
646                 else {
                        //Assign a new node pointer to current nodes next next node.
648                     Node *nextNode = connectedHolesPointer->nextNode->nextNode;
650                     //Assign the current nodes next node to ^.
                        connectedHolesPointer->nextNode = nextNode;
652                 }
654             }
                else{
656
                    //Look at next node.
658                 connectedHolesPointer = connectedHolesPointer->nextNode;
                }
660
            }
662         else{
                //look at next node.
664             connectedHolesPointer = connectedHolesPointer->nextNode;
666         }
668     }
        //Unlock the Mutex.
670     pthread_mutex_unlock(&mutex);
672 }
    /*
674  * Inputs: void
     * Outputs: void
676  * Description: Outputs all nodes.
     */
678 void output(){
        Node *point = HEAD;
680     int loop = 1;
682     while (loop ==1){
            //if next node is null, at the end of the linked list.
684         if (!point->nextNode){
                loop = 0;
686             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);
688             break;
            }else{
690             //Output information about a node.
                printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
                    %d\n\n",
692                     (int)point->size,point->free,(point->memory-sizeof(Node)),
                            point->memory);
                //look at next node.
694             point = point->nextNode;
            }
696
        }
698 }


700
    /*
702  * 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\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);
```

```
822     pthread_create(&thread14, NULL, threadDeallocate, memory);
        pthread_create(&thread15, NULL, threadDeallocate, memory);
824
        pthread_join(thread1,NULL);
826     pthread_join(thread2,NULL);
        pthread_join(thread3,NULL);
828     pthread_join(thread4,NULL);
        pthread_join(thread5,NULL);
830     pthread_join(thread6,NULL);
        pthread_join(thread7,NULL);
832     pthread_join(thread8,NULL);
        pthread_join(thread9,NULL);
834     pthread_join(thread10,NULL);
        pthread_join(thread11,NULL);
836     pthread_join(thread12,NULL);
        pthread_join(thread13,NULL);
838     pthread_join(thread14,NULL);
        pthread_join(thread15,NULL);
840
842     //Final node output.
        printf("\n\n\nFinal node output:\n\n");
844     output();
         */
846     //TEST 3
        /*
848     //Set the algorithm type.
        char * algo = "BestFit";
850
        //Initialise the memory manager.
852     initialise(memory,size,algo);

854     //Output nodes.
        output();
856
        //Create and join all threads.
858     //This creates them and makes them run their respective library functions.
        //They act as libraries trying to allocate memory from themselves using this
            program.
860     //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,
862             thread10,thread11,thread12,thread13,thread14,thread15;
        pthread_create(&thread1, NULL, threadAllocate, NULL);
864     pthread_create(&thread2, NULL, threadAllocate, NULL);
        pthread_create(&thread3, NULL, threadAllocate, NULL);
866     pthread_create(&thread4, NULL, threadAllocate, NULL);
        pthread_create(&thread5, NULL, threadAllocate, NULL);
868     pthread_create(&thread6, NULL, threadAllocate, NULL);
        pthread_create(&thread7, NULL, threadAllocate, NULL);
870     pthread_create(&thread8, NULL, threadAllocate, NULL);
        pthread_create(&thread9, NULL, threadDeallocate, memory);
872     pthread_create(&thread10, NULL, threadDeallocate, memory);
        pthread_create(&thread11, NULL, threadDeallocate, memory);
874     pthread_create(&thread12, NULL, threadDeallocate, memory);
        pthread_create(&thread13, NULL, threadDeallocate, memory);
876     pthread_create(&thread14, NULL, threadDeallocate, memory);
        pthread_create(&thread15, NULL, threadDeallocate, memory);
878
        pthread_join(thread1,NULL);
880     pthread_join(thread2,NULL);
        pthread_join(thread3,NULL);
882     pthread_join(thread4,NULL);
```

```
    pthread_join(thread5,NULL);
884 pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
886 pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
888 pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
890 pthread_join(thread12,NULL);
    pthread_join(thread13,NULL);
892 pthread_join(thread14,NULL);
    pthread_join(thread15,NULL);
894

896 //Final node output.
    printf("\n\n\n\nFinal node output:\n\n");
898 output();
     */
900 //TEST 4
    /*
902 //Set the algorithm type.
    char * algo = "WorstFit";
904
    //Initialise the memory manager.
906 initialise(memory,size,algo);

908 //Output nodes.
    output();
910
    //Create and join all threads.
912 //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this
        program.
914 //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,
916          thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
918 pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
920 pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
922 pthread_create(&thread6, NULL, threadAllocate, NULL);
    pthread_create(&thread7, NULL, threadAllocate, NULL);
924 pthread_create(&thread8, NULL, threadAllocate, NULL);
    pthread_create(&thread9, NULL, threadDeallocate, memory);
926 pthread_create(&thread10, NULL, threadDeallocate, memory);
    pthread_create(&thread11, NULL, threadDeallocate, memory);
928 pthread_create(&thread12, NULL, threadDeallocate, memory);
    pthread_create(&thread13, NULL, threadDeallocate, memory);
930 pthread_create(&thread14, NULL, threadDeallocate, memory);
    pthread_create(&thread15, NULL, threadDeallocate, memory);
932

    pthread_join(thread1,NULL);
934 pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
936 pthread_join(thread4,NULL);
    pthread_join(thread5,NULL);
938 pthread_join(thread6,NULL);
    pthread_join(thread7,NULL);
940 pthread_join(thread8,NULL);
    pthread_join(thread9,NULL);
942 pthread_join(thread10,NULL);
    pthread_join(thread11,NULL);
```

```
944         pthread_join(thread12,NULL);
            pthread_join(thread13,NULL);
946         pthread_join(thread14,NULL);
            pthread_join(thread15,NULL);
948
950         //Final node output.
            printf("\n\n\n\nFinal node output:\n\n");
952         output();
             */
954
            free(memory);
956         return EXIT_SUCCESS;
    }
```