

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	2
part1.c	3
5013ACW02-100237137-file.c	9
part2.h	16
part2.c	17
5013ACW02-100237137-file1.c	28
part3.h	42
part3.c	43
5013ACW02-100237137-file2.c	56

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

21 #endif //CW2_PART1_H
```

part1.c

```

1  /*
   -----

   /

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

   /
5  /                               part1                               /
   /

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

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

11 /   Description: Program to simulate a thread-safe memory manager using a linked
   list           /
   /               implementation.
   /
13 /
   /
   -----
   */

15

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

21

23 //global variables.
   Node *HEAD = NULL;
25

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

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

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

```

```

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

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

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

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

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

53     //return the taken nodes memory address.
54     return nodePointer;
55 }

56
57
58
59
60
61 /*
62  * Inputs: size_t bytes.
63  * Outputs: (void*) ->memory. Memory address of allocated memory.
64  * Description: Uses the FirstFit algorithm to allocate memory from the heap.
65  */
66 void *allocate(size_t bytes) {
67     //Only allow valid amount of bytes.
68     if ((int)bytes <= 0){
69         printf("Invalid amount of bytes. Returning NULL\n");
70         return NULL;
71     }

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

74
75     int allocateBool = 1;

76
77     //Loop through every node.
78     while (allocateBool == 1) {

81         //If the node is free and is large enough for the allocated bytes.
82         if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
83             {

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

86
87                 printf("%d bytes allocated.\n", bytes);

88
89                 //Return the memory address of the allocated node.
90                 return returnNode->memory;
91             }

92
93         //if the node is the exact size.
94         else if (nodePointer->free == 1 && nodePointer->size == bytes) {

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

96
97             //Set free to 0 as node is taken.
98             nodePointer->free = 0;

```

```

101         //Return the memory address of the allocated node.
102         return nodePointer->memory;
103
104     }
105     //if the node is taken.
106     else {
107         //If the next node is NULL.
108         if (nodePointer->nextNode == NULL) {
109             printf("!!!! No free nodes, returning NULL !!!!!\n");
110
111             //Return NULL as no valid nodes found.
112             return NULL;
113         }
114         //Look at next node, and continue in the loop.
115         nodePointer = nodePointer->nextNode;
116     }
117 }
118
119 }
120
121 /*
122  * Inputs: void* memory, size_t size, char* algorithm.
123  * Outputs: void.
124  * Description: Initialises the type of allocation algorithm and creates the
125               initial HEAD node.
126  */
127 void initialise(void *memory ,size_t size){
128
129     //Allocate (size) amount of memory to the heap (called memory in this case).
130     //Returns pointer to the start of the address.
131
132     //Allocate the head node the memory it needs.
133     HEAD = (Node*)(memory);
134
135     //memoryStart is the memory address where data can be stored.
136     //This is done so the struct data come before it.
137     void *memoryStart = ((char*)HEAD) + sizeof(Node);
138
139     //Assign all the nodes variables.
140     HEAD->size = size - sizeof(Node);
141     HEAD->memory = memoryStart;
142     HEAD->free = 1;
143     HEAD->nextNode = NULL;
144     HEAD->prevNode = NULL;
145 }
146
147 /*
148  * Inputs: void* memory.
149  * Outputs: void.
150  * Description: Deallocates a node (frees it) based on the input of its memory
151               address.
152  */
153 void deallocate ( void * memory ){
154
155     //Assign new node pointer to HEAD node.
156     Node *nodePointer = HEAD;
157
158     //int to act as bool for while loop.
159     int deallocateBool = 1;

```

```

161 while(deallocateBool == 1){
163     //if the memory address of deallocation equals the memory address of a
        node.
165
166     if(nodePointer->memory != memory){
167
168         //if the next node is null, current node is the last in the linked
            list.
169         if (nodePointer->nextNode == NULL){
170             deallocateBool = 0;
171             break;
172         }
173         else {
174
175             //look at next node.
176             nodePointer = nodePointer->nextNode;
177         }
178     }
179
180     //else the node is the memory address.
181     else{
182         //Set the nodes free variable to 1, as node has been deallocated.
183         printf("De-allocation successful\n");
184         nodePointer->free=1;
185         deallocateBool = 0;
186         break;
187     }
188 }
189
190 //Next part coalaces connected holes.
191
192 //int to act as bool for while loop.
193 int connectedHoleSearch = 1;
194
195 //Assign a new node pointer to the HEAD node.
196 Node *connectedHolesPointer = HEAD;
197
198
199 while (connectedHoleSearch == 1){
200
201     //if the node is free.
202     if (connectedHolesPointer == NULL){
203         break;
204     }
205     if (connectedHolesPointer->free == 1){
206
207         //if the node is null.
208         if (connectedHolesPointer->nextNode == NULL){
209
210             //set the loop bool to 0.
211             connectedHoleSearch = 0;
212             break;
213         }
214
215         //if the next node is free.
216         if (connectedHolesPointer->nextNode->free == 1) {
217
218             //Increase the current nodes size, to that containing both nodes.
219             connectedHolesPointer->size = connectedHolesPointer->size +
                sizeof(Node)+connectedHolesPointer->nextNode->size;
220
221

```

```

223         //if the next next node is NULL
225         if (connectedHolesPointer->nextNode->nextNode == NULL){
227             //Assign a new node pointer to current nodes next next node.
228             Node *nextNode = connectedHolesPointer->nextNode->nextNode;
229
230             //Assign the current nodes next node to ^.
231             connectedHolesPointer->nextNode = nextNode;
232             connectedHoleSearch = 0;
233             break;
234         }
235
236         else {
237             //Assign a new node pointer to current nodes next next node.
238             Node *nextNode = connectedHolesPointer->nextNode->nextNode;
239
240             //Assign the current nodes next node to ^.
241             connectedHolesPointer->nextNode = nextNode;
242         }
243     }
244     else{
245
246         //Look at next node.
247         connectedHolesPointer = connectedHolesPointer->nextNode;
248     }
249 }
250 else{
251     //look at next node.
252     connectedHolesPointer = connectedHolesPointer->nextNode;
253 }
254 }
255
256 }
257
258 }
259
260 /*
261  * Inputs: void
262  * Outputs: void
263  * Description: Outputs all nodes.
264  */
265 void output(){
266     Node *point = HEAD;
267     int loop = 1;
268
269     while (loop ==1){
270         //if next node is null, at the end of the linked list.
271         if (!point->nextNode){
272             loop = 0;
273             printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
274                 (int)point->size, point->free, (point->memory - sizeof(Node)),
275                 point->memory);
276             break;
277         }else{
278             //Output information about a node.
279             printf("Node \n size: %d\n free: %d\n node start: %d\n memory start: %d\n\n",
280                 (int)point->size, point->free, (point->memory - sizeof(Node)),
281                 point->memory);
282             //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

```

1  /*
   -----

   /

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

   /
5  /                               part1                               /
   /

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

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

   /
11 /   Description: Program to simulate a thread-safe memory manager using a linked
   list      /
   /                               implementation.
   /
13 /
   /
   -----
   */

15

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

21

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

25

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

33

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

```

```

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

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

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

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

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

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

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

59

61  /*
62   * Inputs: size_t bytes.
63   * Outputs: (void*) ->memory. Memory address of allocated memory.
64   * Description: Uses the FirstFit algorithm to allocate memory from the heap.
65   */
void *allocate(size_t bytes) {
67     //Only allow valid amount of bytes.
    if ((int)bytes <= 0){
69         printf("Invalid amount of bytes. Returning NULL\n");
        return NULL;
71     }

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

75     int allocateBool = 1;

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

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

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

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

87             //Return the memory address of the allocated node.
            return returnNode->memory;
91         }

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

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

```

```

97         //Set free to 0 as node is taken.
98         nodePointer->free = 0;
99
100        //Return the memory address of the allocated node.
101        return nodePointer->memory;
102
103    }
104    //if the node is taken.
105    else {
106        //If the next node is NULL.
107        if (nodePointer->nextNode == NULL) {
108            printf("!!!! No free nodes, returning NULL !!!!\n");
109
110            //Return NULL as no valid nodes found.
111            return NULL;
112        }
113        //Look at next node, and continue in the loop.
114        nodePointer = nodePointer->nextNode;
115    }
116 }
117
118 }
119
120
121 /*
122  * Inputs: void* memory, size_t size, char* algorithm.
123  * Outputs: void.
124  * Description: Initialises the type of allocation algorithm and creates the
125                initial HEAD node.
126  */
127 void initialise(void *memory ,size_t size){
128
129     //Allocate (size) amount of memory to the heap (called memory in this case).
130     //Returns pointer to the start of the address.
131
132     //Allocate the head node the memory it needs.
133     HEAD = (Node*)(memory);
134
135     //memoryStart is the memory address where data can be stored.
136     //This is done so the struct data come before it.
137     void *memoryStart = ((char*)HEAD) + sizeof(Node);
138
139     //Assign all the nodes variables.
140     HEAD->size = size - sizeof(Node);
141     HEAD->memory = memoryStart;
142     HEAD->free = 1;
143     HEAD->nextNode = NULL;
144     HEAD->prevNode = NULL;
145
146 }
147
148 /*
149  * Inputs: void* memory.
150  * Outputs: void.
151  * Description: Deallocates a node (frees it) based on the input of its memory
152                address.
153  */
154 void deallocate ( void * memory ){
155
156     //Assign new node pointer to HEAD node.
157     Node *nodePointer = HEAD;

```

```
//int to act as bool for while loop.
```

```
int deallocateBool = 1;
```

```
while(deallocateBool == 1){
```

```
//if the memory address of deallocation equals the memory address of a node.
```

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

```
//if the next node is null, current node is the last in the linked list.
```

```
if (nodePointer->nextNode == NULL){
```

```
    deallocateBool = 0;
```

```
    break;
```

```
}
```

```
else {
```

```
//look at next node.
```

```
nodePointer = nodePointer->nextNode;
```

```
}
```

```
}
```

```
//else the node is the memory address.
```

```
else{
```

```
//Set the nodes free variable to 1, as node has been deallocated.
```

```
printf("De-allocation successful\n");
```

```
nodePointer->free=1;
```

```
deallocateBool = 0;
```

```
break;
```

```
}
```

```
}
```

```
//Next part coalaces connected holes.
```

```
//int to act as bool for while loop.
```

```
int connectedHoleSearch = 1;
```

```
//Assign a new node pointer to the HEAD node.
```

```
Node *connectedHolesPointer = HEAD;
```

```
while (connectedHoleSearch == 1){
```

```
//if the node is free.
```

```
if (connectedHolesPointer == NULL){
```

```
    break;
```

```
}
```

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

```
//if the node is null.
```

```
if (connectedHolesPointer->nextNode == NULL){
```

```
//set the loop bool to 0.
```

```
connectedHoleSearch = 0;
```

```
break;
```

```
}
```

```
//if the next node is free.
```

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

```
//Increase the current nodes size, to that containing both nodes.
```

```

219         connectedHolesPointer->size = connectedHolesPointer->size +
220             sizeof(Node)+connectedHolesPointer->nextNode->size;
221
222
223         //if the next next node is NULL
224         if (connectedHolesPointer->nextNode->nextNode == NULL){
225             //Assign a new node pointer to current nodes next next node.
226             Node *nextNode = connectedHolesPointer->nextNode->nextNode;
227
228             //Assign the current nodes next node to ^.
229             connectedHolesPointer->nextNode = nextNode;
230             connectedHoleSearch = 0;
231             break;
232         }
233
234         else {
235             //Assign a new node pointer to current nodes next next node.
236             Node *nextNode = connectedHolesPointer->nextNode->nextNode;
237
238             //Assign the current nodes next node to ^.
239             connectedHolesPointer->nextNode = nextNode;
240         }
241
242     }
243     else{
244
245         //Look at next node.
246         connectedHolesPointer = connectedHolesPointer->nextNode;
247     }
248
249 }
250 else{
251     //look at next node.
252     connectedHolesPointer = connectedHolesPointer->nextNode;
253
254 }
255
256 }
257
258 }
259
260 /*
261  * Inputs: void
262  * Outputs: void
263  * Description: Outputs all nodes.
264  */
265 void output(){
266     Node *point = HEAD;
267     int loop = 1;
268
269     while (loop ==1){
270         //if next node is null, at the end of the linked list.
271         if (!point->nextNode){
272             loop = 0;
273             printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
274                 %d\n\n",
275                 (int)point->size,point->free,(point->memory-sizeof(Node)),
276                 point->memory);
277             break;
278         }else{
279             //Output information about a node.
280             printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
281                 %d\n\n",

```

```

279         (int)point->size,point->free,(point->memory-typeof(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.
    void *memory = malloc(size);

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

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

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

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

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

317     deallocate(x);
    deallocate(z);

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

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

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

```

1  /*
   -----

   /

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

   /
5  /                               part2                               /
   /

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

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

11 /   Description: Program to simulate a thread-safe memory manager using a linked
   list           /
   /               implementation.
   /
13 /
   /
   -----
   */

15

17 #include <stdio.h>
   #include <stdlib.h>
19 #include "part2.h"
   #include <string.h>
21

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

27 //Function Set as a function pointer so that the different algorithms can be
   applied.
29 /*
   * Inputs: size_t variable.
31 * Outputs: NONE.
   * Description: function pointer to other algorithm allocate functions.
33 */
   void*(*allocate)(size_t);

35

   /*
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;
}

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);
97
                    printf("%d bytes allocated.\n", bytes);

```

```

99         //Return the memory address of the allocated node.
101         return returnNode->memory;
103     }

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

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

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

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

116     }

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

123             //Return NULL as no valid nodes found.
124             return NULL;
125         }
126         //Look at next node, and continue in the loop.
127         nodePointer = nodePointer->nextNode;
128     }
129 }

131

133 /*
134  * Inputs: size_t bytes.
135  * Outputs: (void*) ->memory. Memory address of allocated memory.
136  * Description: Uses the NextFit algorithm to allocate memory from the heap.
137  */
138 void* nextFit(size_t bytes){
139
140     //Only allow valid amount of bytes.
141     if ((int)bytes <= 0){
142         printf("Invalid amount of bytes. Returning NULL\n");
143         return NULL;
144     }

145
146     //Set the initial start location to the NEXTFITNODE node.
147     Node *nodePointer = NEXTFITNODE;

148
149     //Set the end node to the start node, therefore it will only loop once and
150     //not infinitely.
151     Node *limitNode = NEXTFITNODE;

152
153     int allocateBool = 1;

154
155     //Loop through every node.
156     while (allocateBool == 1) {

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

```

```

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

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

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

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

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

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

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

187 }
//if the node is taken.
else {

191     //Look at next node.
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.
return NULL;
209     }

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.
    //Loop through all the nodes and find the best node that fits.
231     int found = 0;
    int bestFit = 1;
    Node *bestNode = NULL;
233     Node *nodePointer = HEAD;
235

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

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

257             //If there is a next node.
            if (nodePointer->nextNode != NULL) {
259                 nodePointer = nodePointer->nextNode;
            }
            else{
261                 if(bestNode == NULL){
263                     bestNode = nodePointer;
                }
                break;
265
            }
        }
        //Look at next node.
        else{
267             if (nodePointer->nextNode != NULL) {
                nodePointer = nodePointer->nextNode;
269             }
            else{
271                 printf("!!!! No free nodes, returning NULL !!!!\n");
            }

273             //Return NULL as no valid nodes found.
            return NULL;
275
        }
    }
}

```

```

285         }
286     }
287
288 }
289
290 //Allocate the best node found with the desired bytes.
291 if(found) {
292     //If the node is free and is large enough for the allocated bytes.
293     if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {
294
295         //Call the allocate function to allocate the node.
296         Node* returnNode = allocateNodeWithHole(bestNode, bytes);
297
298         //return the allocated nodes memory address.
299         return returnNode->memory;
300     }
301
302     //if the node is the exact size.
303     else if (bestNode->free == 1 && bestNode->size == bytes) {
304
305         printf("found node of exact size.\n");
306         //Set free to 0 as node is taken.
307         bestNode->free = 0;
308
309         //return the allocated nodes memory address.
310         return bestNode->memory;
311     }
312 }
313
314 //If no suitable nodes found, return NULL.
315 else {
316     printf("!!!! No free nodes, returning NULL !!!!\n");
317     //Return NULL as no valid nodes found.
318     return NULL;
319 }
320
321 }
322
323
324 }
325
326
327
328
329 /*
330  * Inputs: size_t bytes.
331  * Outputs: (void*) ->memory. Memory address of allocated memory.
332  * Description: Uses the WorstFit algorithm to allocate memory from the heap.
333  */
334 void* worstFit(size_t bytes) {
335     //Only allow valid amount of bytes.
336     if ((int)bytes <= 0){
337         printf("Invalid amount of bytes. Returning NULL\n");
338         return NULL;
339     }
340
341     //Set the initial node to the HEAD node.
342     //Loop through all the nodes and find the worst node that fits.
343     int found = 0;
344     int worstFit = 1;
345     Node *worstNode = NULL;
346     Node *nodePointer = HEAD;

```

```

349 while (worstFit) {
350     if (nodePointer->free) {
351
352         //IF the node is of exact size.
353         if (nodePointer->size == bytes) {
354             found = 1;
355             if (worstNode == NULL) {
356                 worstNode = nodePointer;
357             } else if (nodePointer->size > worstNode->size) {
358                 worstNode = nodePointer;
359             }
360
361         }
362
363         //IF the node is greater than bytes
364         else if (nodePointer->size > (bytes + (sizeof(Node)))) {
365             found = 1;
366             if (worstNode == NULL) {
367                 worstNode = nodePointer;
368             }
369             if (nodePointer->size > worstNode->size) {
370                 worstNode = nodePointer;
371             }
372
373         } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
374             printf("Invalid node to use. searching next\n");
375
376         }
377     }
378
379     if (nodePointer->nextNode != NULL) {
380         nodePointer = nodePointer->nextNode;
381     } else {
382         break;
383     }
384 }
385
386 if(found) {
387
388     //If the node is free and is large enough for the allocated bytes.
389     if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
390
391         //Call the allocate function to allocate the node.
392         Node* returnNode = allocateNodeWithHole(worstNode, bytes);
393
394         //return the allocated nodes memory address.
395         return returnNode->memory;
396     }
397
398
399
400     //if the node is the exact size.
401     else if (worstNode->free == 1 && worstNode->size == bytes) {
402         printf("found node of exact size.\n");
403         //Set free to 0 as node is taken.
404         worstNode->free = 0;
405
406         //return the allocated nodes memory address.
407         return worstNode->memory;
408     }
409 }

```

```

411     }
413     //If not found.
414     else {
415         printf("!!!! No free nodes, returning NULL !!!!!\n");
417         return NULL;
419     }
421 }
423
424 /*
425  * Inputs: void* memory, size_t size, char* algorithm.
426  * Outputs: void.
427  * Description: Initialises the type of allocation algorithm and creates the
428  *              initial HEAD node.
429  */
430 void initialise(void *memory, size_t size, char* algorithm){
431     //Allocate (size) amount of memory to the heap (called memory in this case).
432     //Returns pointer to the start of the address.
434
435     //Allocate the head node the memory it needs.
436     HEAD = (Node*)(memory);
438
439     //Assign type of algorithm
441
442     if(strcmp(algorithm, "NextFit")==0){
443         printf("Using NextFit algorithm.\n");
444         allocate = nextFit;
445         NEXTFITNODE = HEAD;
447     }
448     else if(strcmp(algorithm, "BestFit")==0){
449         printf("Using BestFit algorithm.\n");
450         allocate = bestFit;
452     }
453     else if(strcmp(algorithm, "WorstFit")==0){
454         printf("Using WorstFit algorithm.\n");
455         allocate = worstFit;
457     }
458     //Default first fit
459     else{
460         printf("Using FirstFit algorithm.\n");
461         allocate = firstFit;
463     }
465
466     //memoryStart is the memory address where data can be stored.
467     //This is done so the struct data come before it.
468     void *memoryStart = ((char*)HEAD) + sizeof(Node);
470
471     //Assign all the nodes variables.
472     HEAD->size = size - sizeof(Node);
473     HEAD->memory = memoryStart;
474     HEAD->free = 1;
475     HEAD->nextNode = NULL;
476     HEAD->prevNode = NULL;

```



```

473     }
475
476     /*
477      * Inputs: void* memory.
478      * Outputs: void.
479      * Description: Deallocates a node (frees it) based on the input of its memory
480      address.
481      */
482     void deallocate ( void * memory ){
483
484         //Assign new node pointer to HEAD node.
485         Node *nodePointer = HEAD;
486
487         //int to act as bool for while loop.
488         int deallocateBool = 1;
489
490         while(deallocateBool == 1){
491
492             //if the memory address of deallocation equals the memory address of a
493             node.
494
495             if(nodePointer->memory != memory){
496
497                 //if the next node is null, current node is the last in the linked
498                 list.
499                 if (nodePointer->nextNode == NULL){
500                     deallocateBool = 0;
501                     break;
502                 }
503                 else {
504
505                     //look at next node.
506                     nodePointer = nodePointer->nextNode;
507                 }
508             }
509
510             //else the node is the memory address.
511             else{
512                 //Set the nodes free variable to 1, as node has been deallocated.
513                 printf("De-allocation successful\n");
514                 nodePointer->free=1;
515                 deallocateBool = 0;
516                 break;
517             }
518         }
519
520         //Next part coalaces connected holes.
521
522         //int to act as bool for while loop.
523         int connectedHoleSearch = 1;
524
525         //Assign a new node pointer to the HEAD node.
526         Node *connectedHolesPointer = HEAD;
527
528         while (connectedHoleSearch == 1){
529
530             //if the node is free.
531             if (connectedHolesPointer == NULL){
532                 break;
533             }

```

```

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

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

537             //set the loop bool to 0.
538             connectedHoleSearch = 0;
539             break;

541         }

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

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

549             //If NextFit algorithm is set, handle the pointer to it.
550             if (NEXTFITNODE != NULL){
551                 if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
552                     memory){
553                     printf("Moving NEXTFITNODE due to coalesce taking place.\n
554                         ");
555                     NEXTFITNODE = connectedHolesPointer;
556                 }
557             }

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

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

567             }

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

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

575             }

577         }
578         else{

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

583         }
584     }
585     else{
586         //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 }
```

5013ACW02-100237137-file1.c

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

part2_test.c

```

1  /*
   -----

   /

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

   /
5  /                                part2 TEST HARNESS
   /                                /
   /

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

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

   /
11 /  Description: Program to simulate a thread-safe memory manager using a linked
   list      /
   /                                implementation.
   /                                /
13 /
   /
   -----
   */

15

17 #include <stdio.h>
   #include <stdlib.h>
19 #include "part2.h"
   #include <string.h>
21

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

27
   //Function Set as a function pointer so that the different algorithms can be
   applied.
29 /*
   * Inputs: size_t variable.
31 * Outputs: NONE.
   * Description: function pointer to other algorithm allocate functions.
33 */
   void*(*allocate)(size_t);
35
   /*

```

```

37  * Inputs: Node* nodePointer, size_t bytes.
38  * 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.
40  */
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);

97     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;
    }
    //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.

```

```

159     if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
160     {
161         //Call the allocate function to allocate the node.
162         Node *returnNode = allocateNodeWithHole(nodePointer, bytes);
163
164         //update NEXTFITNODE.
165         NEXTFITNODE = returnNode->nextNode;
166
167         //return the allocated nodes memory address.
168         return nodePointer->memory;
169     }
170
171     //if the node is the exact size.
172     else if (nodePointer->free == 1 && nodePointer->size == bytes) {
173
174         //Set free to 0 as node is taken.
175         printf("Found node of exact size.\n");
176         nodePointer->free = 0;
177
178         //update NEXTFITNODE,
179         if (nodePointer->nextNode != NULL) {
180             NEXTFITNODE = nodePointer->nextNode;
181         }
182         else{
183             NEXTFITNODE = HEAD;
184         }
185
186         //return the allocated nodes memory address.
187         return nodePointer->memory;
188     }
189     //if the node is taken.
190     else {
191
192         //Look at next node.
193         if (nodePointer->nextNode == NULL) {
194
195             //loop through whole list
196             nodePointer = HEAD;
197         }
198         else{
199             nodePointer = nodePointer->nextNode;
200         }
201         if (nodePointer->memory == limitNode->memory){
202
203             printf("!!!! No free nodes, returning NULL !!!!\n");
204
205             //update NEXTFITNODE.
206             NEXTFITNODE= HEAD;
207
208             //Return NULL as no valid nodes found.
209             return NULL;
210         }
211     }
212 }
213 }
214 }
215 }
216
217 /*
218 * Inputs: size_t bytes.

```

```

221  * Outputs: (void*) ->memory. Memory address of allocated memory.
222  * Description: Uses the BestFit algorithm to allocate memory from the heap.
223  */
224 void* bestFit(size_t bytes){
225     //Only allow valid amount of bytes.
226     if ((int)bytes <= 0){
227         printf("Invalid amount of bytes. Returning NULL\n");
228         return NULL;
229     }
230
231     //Set the initial node to the HEAD node.
232     //Loop through all the nodes and find the best node that fits.
233     int found = 0;
234     int bestFit = 1;
235     Node *bestNode = NULL;
236     Node *nodePointer = HEAD;
237
238     while(bestFit){
239         if(nodePointer->free){
240
241             //IF the node is of exact size.
242             if (nodePointer->size == bytes){
243                 bestNode = nodePointer;
244                 found = 1;
245                 break;
246             }
247
248             //IF the node is greater than bytes
249             if(nodePointer->size > (bytes + (sizeof(Node)))){
250                 found = 1;
251                 if(bestNode == NULL){
252                     bestNode = nodePointer;
253                 }
254                 if(nodePointer->size < bestNode->size){
255                     bestNode = nodePointer;
256                 }
257             }
258
259             //If there is enough size to create a new node but not assign a size
260             // > 0 to it.
261             if(nodePointer->size == (bytes +(sizeof(Node)))){
262                 printf("Invalid node to use. searching next\n");
263             }
264
265             //If there is a next node.
266             if (nodePointer->nextNode != NULL) {
267                 nodePointer = nodePointer->nextNode;
268             }
269             else{
270                 if(bestNode == NULL){
271                     bestNode = nodePointer;
272                 }
273                 break;
274             }
275
276             //Look at next node.
277             else{
278                 if (nodePointer->nextNode != NULL) {
279                     nodePointer = nodePointer->nextNode;
280                 }
281                 else{
282                     printf("!!!! No free nodes, returning NULL !!!!\n");

```



```

283         //Return NULL as no valid nodes found.
284         return NULL;
285     }
286 }
287
288 }
289
290 //Allocate the best node found with the desired bytes.
291 if(found) {
292     //If the node is free and is large enough for the allocated bytes.
293     if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {
294
295         //Call the allocate function to allocate the node.
296         Node* returnNode = allocateNodeWithHole(bestNode, bytes);
297
298         //return the allocated nodes memory address.
299         return returnNode->memory;
300     }
301
302     //if the node is the exact size.
303     else if (bestNode->free == 1 && bestNode->size == bytes) {
304
305         printf("found node of exact size.\n");
306         //Set free to 0 as node is taken.
307         bestNode->free = 0;
308
309         //return the allocated nodes memory address.
310         return bestNode->memory;
311     }
312 }
313
314 //If no suitable nodes found, return NULL.
315 else {
316     printf("!!!! No free nodes, returning NULL !!!!\n");
317     //Return NULL as no valid nodes found.
318     return NULL;
319 }
320
321 }
322
323 }
324
325 }
326
327
328
329 /*
330  * Inputs: size_t bytes.
331  * Outputs: (void*) ->memory. Memory address of allocated memory.
332  * Description: Uses the WorstFit algorithm to allocate memory from the heap.
333  */
334 void* worstFit(size_t bytes) {
335     //Only allow valid amount of bytes.
336     if ((int)bytes <= 0){
337         printf("Invalid amount of bytes. Returning NULL\n");
338         return NULL;
339     }
340
341     //Set the initial node to the HEAD node.
342     //Loop through all the nodes and find the worst node that fits.
343     int found = 0;

```

```

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

349     while (worstFit) {
350         if (nodePointer->free) {
351
352             //IF the node is of exact size.
353             if (nodePointer->size == bytes) {
354                 found = 1;
355                 if (worstNode == NULL) {
356                     worstNode = nodePointer;
357                 } else if (nodePointer->size > worstNode->size) {
358                     worstNode = nodePointer;
359                 }
360             }
361
362             //IF the node is greater than bytes
363             else if (nodePointer->size > (bytes + (sizeof(Node)))) {
364                 found = 1;
365                 if (worstNode == NULL) {
366                     worstNode = nodePointer;
367                 }
368                 if (nodePointer->size > worstNode->size) {
369                     worstNode = nodePointer;
370                 }
371             }
372
373             } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
374                 printf("Invalid node to use. searching next\n");
375             }
376         }
377     }
378
379     if (nodePointer->nextNode != NULL) {
380         nodePointer = nodePointer->nextNode;
381     } else {
382         break;
383     }
384 }

386 if(found) {
387
388     //If the node is free and is large enough for the allocated bytes.
389     if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
390
391         //Call the allocate function to allocate the node.
392         Node* returnNode = allocateNodeWithHole(worstNode, bytes);
393
394         //return the allocated nodes memory address.
395         return returnNode->memory;
396     }
397
398
399
400     //if the node is the exact size.
401     else if (worstNode->free == 1 && worstNode->size == bytes) {
402         printf("found node of exact size.\n");
403         //Set free to 0 as node is taken.
404         worstNode->free = 0;
405
406         //return the allocated nodes memory address.

```

```

        return worstNode->memory;
409
    }
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
}
451 else if(strcmp(algorithm,"WorstFit")==0){
    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.
478  * Outputs: void.
479  * Description: Deallocates a node (frees it) based on the input of its memory
      address.
480  */
481 void deallocate ( void * memory ){

482     //Assign new node pointer to HEAD node.
483     Node *nodePointer = HEAD;
484
485     //int to act as bool for while loop.
486     int deallocateBool = 1;
487
488
489     while(deallocateBool == 1){
490
491         //if the memory address of deallocation equals the memory address of a
            node.
492
493         if(nodePointer->memory != memory){
494
495             //if the next node is null, current node is the last in the linked
                list.
496             if (nodePointer->nextNode == NULL){
497                 deallocateBool = 0;
498                 break;
499             }
500             else {
501
502                 //look at next node.
503                 nodePointer = nodePointer->nextNode;
504             }
505         }
506
507         //else the node is the memory address.
508     else{
509         //Set the nodes free variable to 1, as node has been deallocated.
510         printf("De-allocation successful\n");
511         nodePointer->free=1;
512         deallocateBool = 0;
513         break;
514     }
515 }
516
517 //Next part coalaces connected holes.
518
519 //int to act as bool for while loop.
520 int connectedHoleSearch = 1;
521
522 //Assign a new node pointer to the HEAD node.
523 Node *connectedHolesPointer = HEAD;
524
525
526 while (connectedHoleSearch == 1){
527
528     //if the node is free.

```

```

531     if (connectedHolesPointer == NULL){
532         break;
533     }
534     if (connectedHolesPointer->free == 1){
535
536         //if the node is null.
537         if (connectedHolesPointer->nextNode == NULL){
538
539             //set the loop bool to 0.
540             connectedHoleSearch = 0;
541             break;
542         }
543
544         //if the next node is free.
545         if (connectedHolesPointer->nextNode->free == 1) {
546
547             //Increase the current nodes size, to that containing both nodes.
548             connectedHolesPointer->size = connectedHolesPointer->size +
549                 sizeof(Node)+connectedHolesPointer->nextNode->size;
550
551             //If NextFit algorithm is set, handle the pointer to it.
552             if (NEXTFITNODE != NULL){
553                 if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
554                     memory){
555                     printf("Moving NEXTFITNODE due to coalesce taking place.\n
556                         ");
557                     NEXTFITNODE = connectedHolesPointer;
558                 }
559             }
560
561             //if the next next node is NULL
562             if (connectedHolesPointer->nextNode->nextNode == NULL){
563                 //Assign a new node pointer to current nodes next next node.
564                 Node *nextNode = connectedHolesPointer->nextNode->nextNode;
565
566                 //Assign the current nodes next node to ^.
567                 connectedHolesPointer->nextNode = nextNode;
568                 connectedHoleSearch = 0;
569                 break;
570             }
571
572             else {
573                 //Assign a new node pointer to current nodes next next node.
574                 Node *nextNode = connectedHolesPointer->nextNode->nextNode;
575
576                 //Assign the current nodes next node to ^.
577                 connectedHolesPointer->nextNode = nextNode;
578             }
579
580         }
581         else{
582
583             //Look at next node.
584             connectedHolesPointer = connectedHolesPointer->nextNode;
585         }
586     }
587     else{
588         //look at next node.
589         connectedHolesPointer = connectedHolesPointer->nextNode;
590     }

```

```

591     }

593 }
/*
595  * Inputs: void
596  * Outputs: void
597  * Description: Outputs all nodes.
598 */
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",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                    point->memory);
613             //look at next node.
            point = point->nextNode;
615         }
617     }
619 }

621

623 int main() {
    /*
625     * TEST HARNESS FOR EACH ALGORITHM COMMENTED OUT.
626     */
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
702 //test1 should go in x, test2 should go in z.
703 /*
704 char * algo = "BestFit";
705 initialise(memory,size,algo);
706
707 void *x = allocate(50);
708 void *y = allocate(100);
709 void *z = allocate(200);
710 void *b = allocate(30);
711 printf("Allocate 4 nodes: \n===== \n\nVisual Node
       output:\n\n");
712 output();
713
714 deallocate(x);
715 deallocate(z);
716
717 printf("De-allocate x & z: \n===== \n\nVisual Node
       output:\n\n");
718 output();
719
720 //Should replace node x;
721 void *test1 = allocate(50);
722 printf("test allocation of 50: \n===== \n\nVisual
       Node output:\n\n");
723 output();
724
725 deallocate(test1);
726 printf("De-allocation of testAllocation: \n===== \n\n
       Visual Node output:\n\n");
727 output();
728
729 //Should use node z, and create new hole:
730 void *test2 = allocate(150);
731 printf("test2 of 150: \n===== \n\nVisual Node output
       :\n\n");
732 output();
733 */
734
735
736 //WorstFit Algorithm Test Harness
737 //test1 should go in the hole after b, and so should test2
738 /*
739 char * algo = "WorstFit";
740 initialise(memory,size,algo);
741
742 void *x = allocate(50);
743 void *y = allocate(100);
744 void *z = allocate(200);
745 void *b = allocate(30);
746 printf("Allocate 4 nodes: \n===== \n\nVisual Node
       output:\n\n");
747 output();
748
749 deallocate(x);
750 deallocate(z);
751
752 printf("De-allocate x & z: \n===== \n\nVisual Node
       output:\n\n");
753 output();
754
755

```



```
757 //Should replace node x;
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\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

```

1  /*
   -----

   /

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

   /
5  /                               part3                               /
   /

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

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

   /
11 /   Description: Program to simulate a thread-safe memory manager using a linked
   list           /
   /               implementation.
   /
13 /
   /
   -----
   */

15

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

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

29 //Function Set as a function pointer so that the different algorithms can be
   applied.
31 /*
   * Inputs: size_t variable.
33 * Outputs: NONE.
   * Description: function pointer to other algorithm allocate functions.
35 */
   void*(*allocate)(size_t);

37
   /*
39 * Inputs: Node* nodePointer, size_t bytes.

```

```

41  * Outputs: Node* (return node).
42  * Description: Function takes a node, allocates a size to it, and creates a free
      node next to it with the remaining space.
43  */
Node* allocateNodeWithHole(Node* nodePointer, size_t bytes){
    printf("Found node of suitable size, creating, allocating space and creating
      node\n");

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

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

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

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

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

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

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

71

73  /*
74  * Inputs: size_t bytes.
75  * Outputs: (void*) ->memory. Memory address of allocated memory.
76  * Description: Uses the FirstFit algorithm to allocate memory from the heap.
77  */
void *firstFit(size_t bytes) {
79    //Used to stop the function allocating 0 bytes to a node if the calling
      thread requests it, if not handled
81    //node would be invalid.
    if((int)bytes <= 0){
83        printf("Thread ID: %d, attempted to allocate 0 memory, returning NULL
          .!!!\n",pthread_self());
        return NULL;
85    }
    printf("\n %d: attempting to gain lock.\n",pthread_self());

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

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

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

97    int allocateBool = 1;

```

```

99  //Loop through every node.
    while (allocateBool == 1) {
101
102      //If the node is free and is large enough for the allocated bytes.
103      if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
104      {
105
106          //Call the allocate function to allocate the node.
107          Node* returnNode = allocateNodeWithHole(nodePointer, bytes);
108
109          printf("%d bytes allocated by thread %d.\n", bytes, pthread_self());
110
111          //Unlock the Mutex.
112          pthread_mutex_unlock(&mutex);
113
114          //Return the memory address of the allocated node.
115          return returnNode->memory;
116      }
117
118      //if the node is the exact size.
119      else if (nodePointer->free == 1 && nodePointer->size == bytes) {
120
121          printf("thread: %d: found node of exact size.\n", pthread_self());
122
123          //Set free to 0 as node is taken.
124          nodePointer->free = 0;
125
126          //Unlock the Mutex.
127          pthread_mutex_unlock(&mutex);
128
129          //Return the memory address of the allocated node.
130          return nodePointer->memory;
131      }
132      //if the node is taken.
133      else {
134          //If the next node is NULL.
135          if (nodePointer->nextNode == NULL) {
136              printf("!!!! No free nodes, returning NULL !!!!\n");
137
138              //Unlock Mutex
139              pthread_mutex_unlock(&mutex);
140
141              //Return NULL as no valid nodes found.
142              return NULL;
143          }
144          //Look at next node, and continue in the loop.
145          nodePointer = nodePointer->nextNode;
146      }
147  }
148
149 }
150
151 /*
152  * Inputs: size_t bytes.
153  * Outputs: (void*) ->memory. Memory address of allocated memory.
154  * Description: Uses the NextFit algorithm to allocate memory from the heap.
155  */
156 void* nextFit(size_t bytes){
157
158     //Used to stop the function allocating 0 bytes to a node if the calling

```

```

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

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

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

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

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

179    int allocateBool = 1;

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

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

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

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

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

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

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

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

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

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

215            //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
            nodePointer = HEAD;

229         }
        else{
            nodePointer = nodePointer->nextNode;

231         }
        if (nodePointer->memory == limitNode->memory){

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

235             //update NEXTFITNODE.
            NEXTFITNODE = HEAD;

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

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

241         }

243     }

245 }

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.
    pthread_mutex_lock(&mutex);

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

273     //Set the initial node to the HEAD node.
    //Loop through all the nodes and find the best node that fits.
275     int found = 0;
    int bestFit = 1;
277
279

```

```

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

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

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

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

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

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

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

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

//Allocate the best node found with the desired bytes.
if(found) {
    //If the node is free and is large enough for the allocated bytes.

```



```

343     if (bestNode->free == 1 && bestNode->size > (bytes + sizeof(Node))) {
344         //Call the allocate function to allocate the node.
345         Node* returnNode = allocateNodeWithHole(bestNode, bytes);
346
347         //Unlock the Mutex.
348         pthread_mutex_unlock(&mutex);
349
350         //return the allocated nodes memory address.
351         return returnNode->memory;
352     }
353
354     //if the node is the exact size.
355     else if (bestNode->free == 1 && bestNode->size == bytes) {
356
357         printf("found node of exact size.\n");
358         //Set free to 0 as node is taken.
359         bestNode->free = 0;
360
361         //Unlock the Mutex.
362         pthread_mutex_unlock(&mutex);
363
364         //return the allocated nodes memory address.
365         return bestNode->memory;
366     }
367
368 }
369 //If no suitable nodes found, return NULL.
370 else {
371     printf("!!!! No free nodes, returning NULL !!!!!\n");
372
373     //Unlock the Mutex.
374     pthread_mutex_unlock(&mutex);
375
376     //Return NULL as no valid nodes found.
377     return NULL;
378 }
379
380 }
381
382 }
383
384 }
385
386
387 /*
388  * Inputs: size_t bytes.
389  * Outputs: (void*) ->memory. Memory address of allocated memory.
390  * Description: Uses the WorstFit algorithm to allocate memory from the heap.
391  */
392 void* worstFit(size_t bytes) {
393
394     //Used to stop the function allocating 0 bytes to a node if the calling
395     // thread requests it, if not handled
396     // node would be invalid.
397     if((int)bytes <= 0){
398         printf("==== Worker ID: %d, attempted to allocate 0 memory, returning
399         NULL.!!!!\n", pthread_self());
400         return NULL;
401     }
402
403     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.
//Loop through all the nodes and find the worst node that fits.
411 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 }

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

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

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

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

}

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

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

    return NULL;
}
}

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

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

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

    //Assign type of algorithm

    if(strcmp(algorithm,"NextFit")==0){
        allocate = nextFit;
        NEXTFITNODE = HEAD;
    }
    else if(strcmp(algorithm, "BestFit")==0){
        allocate = bestFit;
    }
    else if(strcmp(algorithm,"WorstFit")==0){
        allocate = worstFit;
    }
}

```

```

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 deallocation 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.
                nodePointer = nodePointer->nextNode;
581
            }
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 }

//Next part coalaces connected holes.

//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){

            //set the loop bool to 0.
615             connectedHoleSearch = 0;
            break;
617         }

619
        //if the next node is free.
621         if (connectedHolesPointer->nextNode->free == 1) {

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

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

633
            //if the next next node is NULL
635             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.

```

```

649         Node *nextNode = connectedHolesPointer->nextNode->nextNode;

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

653 }

655 else{

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

659     }

661 }
    else{
        //look at next node.
        connectedHolesPointer = connectedHolesPointer->nextNode;

665     }

667 }

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

671 }

673 /*
    * Inputs: void
    * Outputs: void
    * Description: Outputs all nodes.
    */
void output(){
679     Node *point = HEAD;
    int loop = 1;

681     while (loop ==1){
        //if next node is null, at the end of the linked list.
        if (!point->nextNode){
683             loop = 0;
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
685                 %d\n\n",
                (int)point->size,point->free,(point->memory-sizeof(Node)),
                    point->memory);

            break;
        }else{
689             //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
691                 %d\n\n",
                (int)point->size,point->free,(point->memory-sizeof(Node)),
                    point->memory);

            //look at next node.
            point = point->nextNode;

693         }

695     }

697 }

699 }

701 /*
    * Inputs: void.
    * Outputs: void.
    * Description: Used by threads that act as a library to randomly allocate random
        amounts of memory.
705 */

```

```
void* threadAllocate(){
707     srand(time(NULL));
    for (int i = 0; i < 500; i++) {
709         size_t allocateAmount = rand() % 200;
        printf("ID: %d, trying to allocate: %d.\n",pthread_self(),allocateAmount)
            ;
711         void *x = allocate(allocateAmount);
    }
713 }

715
/*
717  * Inputs: void.
718  * Outputs: void.
719  * Description: Used by threads that act as a library to randomly deallocate
    random amounts of memory.
*/
721 void* threadDeallocate(void* memory){
    srand(time(NULL));
723     for (int i=0; i<800; i++){
        int x = rand() % ((memory+1024) + 1 - memory) + memory;
725         deallocate((void*)x);
    }
727 }
```

5013ACW02-100237137-file2.c

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

part3_test.c

```

2  /*
   -----
   /
   /
   /   Title: Architectures & Operating Systems: Coursework 2 - Thread-safe Memory
   /   Manager.
4  /
   /
   /
   /                               part3 TEST HARNESS
6  /
   /
   /   Authors: Buzz Embley-Riches 100237137 & James Burrell 100263300.
8  /
   /
   /   Last edit date: 06/12/19
10 /
   /
   /   Description: Program to simulate a thread-safe memory manager using a linked
   /   list
12 /   implementation.
   /
   /
14 -----
   */

16 #include <stdio.h>
18 #include <stdlib.h>
   #include "part3.h"
20 #include <pthread.h>
   #include <string.h>
22

24 //global variables.
   Node *HEAD = NULL;
26 Node *NEXTFITNODE = NULL;
   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
28

30 //Function Set as a function pointer so that the different algorithms can be
   applied.
   /*
32  * Inputs: size_t variable.
   * Outputs: NONE.
34  * Description: function pointer to other algorithm allocate functions.
   */
36 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.
        pthread_mutex_unlock(&mutex);
126
        //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.
            return NULL;
142
        }
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){
158
    //Used to stop the function allocating 0 bytes to a node if the calling
    // thread requests it, if not handled node
160    // would be invalid.
    if((int)bytes <= 0){
162        printf("=== Worker ID: %d, attempted to allocate 0 memory, returning
        NULL.!!!\n",pthread_self());
        return NULL;
164    }
    printf("\n %d: attempting to gain lock.\n",pthread_self());
166
    //Thread attempts to lock the mutex.
168    pthread_mutex_lock(&mutex);

170    //When the thread gains the lock, attempt to allocate the input bytes.
    printf("\n %d: acquired lock\n",pthread_self());
172
    //Set the initial start location to the NEXTFITNODE node.
174    Node *nodePointer = NEXTFITNODE;

176    //Set the end node to the start node, therefore it will only loop once and
    // not infinitely.
    Node *limitNode = NEXTFITNODE;
178

    int allocateBool = 1;
180

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

184        //If the node is free and is large enough for the allocated bytes.
        if (nodePointer->free == 1 && nodePointer->size > (bytes + sizeof(Node)))
        {
186
188            //Call the allocate function to allocate the node.
            Node *returnNode = allocateNodeWithHole(nodePointer,bytes);

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

192            //Unlock the Mutex.
            pthread_mutex_unlock(&mutex);
194

196            //return the allocated nodes memory address.
            return nodePointer->memory;
198        }

200        //if the node is the exact size.
        else if (nodePointer->free == 1 && nodePointer->size == bytes) {
202
204            //Set free to 0 as node is taken.
            printf("Found node of exact size.\n");
            nodePointer->free = 0;

206            //update NEXTFITNODE,
            if(nodePointer->nextNode != NULL) {
208                NEXTFITNODE = nodePointer->nextNode;
210            }
            else{
212                NEXTFITNODE = HEAD;
214            }

            //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.
        if (nodePointer->nextNode == NULL) {

226             //loop through whole list
            nodePointer = HEAD;

228         }
        else{

230             nodePointer = nodePointer->nextNode;

232         }
        if (nodePointer->memory == limitNode->memory){

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

236             //update NEXTFITNODE.
            NEXTFITNODE= HEAD;

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

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

242         }

244     }

246 }

248 }

250 }

252

254 /*
    * 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){

260     //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){

262         printf("=== Worker ID: %d, attempted to allocate 0 memory, returning
            NULL.!!!!\n",pthread_self());
        return NULL;

264     }

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

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

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

272     //Set the initial node to the HEAD node.

```

```

//Loop through all the nodes and find the best node that fits.
278 int found = 0;
279 int bestFit = 1;
280 Node *bestNode = NULL;
281 Node *nodePointer = HEAD;
282
283 while(bestFit){
284     if(nodePointer->free){
285
286         //IF the node is of exact size.
287         if (nodePointer->size == bytes){
288             bestNode = nodePointer;
289             found = 1;
290             break;
291         }
292
293         //IF the node is greater than bytes
294         if(nodePointer->size > (bytes + (sizeof(Node)))){
295             found = 1;
296             if(bestNode == NULL){
297                 bestNode = nodePointer;
298             }
299             if(nodePointer->size < bestNode->size){
300                 bestNode = nodePointer;
301             }
302         }
303
304         //If there is enough size to create a new node but not assign a size
305         > 0 to it.
306         if(nodePointer->size == (bytes +(sizeof(Node)))){
307             printf("Invalid node to use. searching next\n");
308         }
309
310         //If there is a next node.
311         if (nodePointer->nextNode != NULL) {
312             nodePointer = nodePointer->nextNode;
313         }
314         else{
315             if(bestNode == NULL){
316                 bestNode = nodePointer;
317             }
318             break;
319         }
320     }
321
322     //Look at next node.
323     else{
324         if (nodePointer->nextNode != NULL) {
325             nodePointer = nodePointer->nextNode;
326         }
327         else{
328             printf("!!!! No free nodes, returning NULL !!!!!\n");
329             //Unlock the Mutex.
330             pthread_mutex_unlock(&mutex);
331
332             //Return NULL as no valid nodes found.
333             return NULL;
334         }
335     }
336 }
337
338

```

```

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

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

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

350         //return the allocated nodes memory address.
        return returnNode->memory;
352     }

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

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

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

364         //return the allocated nodes memory address.
        return bestNode->memory;
366     }

368 }

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

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

376     //Return NULL as no valid nodes found.
    return NULL;
378 }

380 }

382

384 }

386
/*
388 * Inputs: size_t bytes.
    * Outputs: (void*) ->memory. Memory address of allocated memory.
390 * Description: Uses the WorstFit algorithm to allocate memory from the heap.
    */
392 void* worstFit(size_t bytes) {

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

```

```

400     printf("\n %d: attempting to gain lock.\n",pthread_self());
402
403     //Thread attempts to lock the mutex.
404     pthread_mutex_lock(&mutex);
406
407     //When the thread gains the lock, attempt to allocate the input bytes.
408
409     printf("\n %d: acquired lock\n",pthread_self());
410
411     //Set the initial node to the HEAD node.
412     //Loop through all the nodes and find the worst node that fits.
413     int found = 0;
414     int worstFit = 1;
415     Node *worstNode = NULL;
416     Node *nodePointer = HEAD;
418
419     while (worstFit) {
420         if (nodePointer->free) {
421
422             //IF the node is of exact size.
423             if (nodePointer->size == bytes) {
424                 found = 1;
425                 if (worstNode == NULL) {
426                     worstNode = nodePointer;
427                 } else if (nodePointer->size > worstNode->size) {
428                     worstNode = nodePointer;
429                 }
430
431             }
432
433             //IF the node is greater than bytes
434             else if (nodePointer->size > (bytes + (sizeof(Node)))) {
435                 found = 1;
436                 if (worstNode == NULL) {
437                     worstNode = nodePointer;
438                 }
439                 if (nodePointer->size > worstNode->size) {
440                     worstNode = nodePointer;
441                 }
442
443             } else if (nodePointer->size == (bytes + (sizeof(Node)))) {
444                 printf("Invalid node to use. searching next\n");
445
446             }
447
448             if (nodePointer->nextNode != NULL) {
449                 nodePointer = nodePointer->nextNode;
450             } else {
451                 break;
452             }
453         }
454
455         if(found) {
456
457             //If the node is free and is large enough for the allocated bytes.
458             if (worstNode->free == 1 && worstNode->size > (bytes + sizeof(Node))) {
459
460                 //Call the allocate function to allocate the node.
461                 Node* returnNode = allocateNodeWithHole(worstNode,bytes);

```

```

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

496         return NULL;
    }

498 }

500
501 /*
502  * Inputs: void* memory, size_t size, char* algorithm.
503  * Outputs: void.
504  * Description: Initialises the type of allocation algorithm and creates the
505  *              initial HEAD node.
506  */
507 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

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

514

515     //Assign type of algorithm

516     if(strcmp(algorithm,"NextFit")==0){
518         allocate = nextFit;
        NEXTFITNODE = HEAD;
520     }
    else if(strcmp(algorithm, "BestFit")==0){
522         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);
544 HEAD->memory = memoryStart;
546 HEAD->free = 1;
HEAD->nextNode = NULL;
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 deallocation 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 {

            //look at next node.
            nodePointer = nodePointer->nextNode;
580
        }
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.
                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.
                connectedHolesPointer->size = connectedHolesPointer->size +
624                    sizeof(Node)+connectedHolesPointer
                        ->nextNode->size;

626

                //If NextFit algorithm is set, handle the pointer to it.
                if (NEXTFITNODE != NULL){
628                    if(NEXTFITNODE->memory == connectedHolesPointer->nextNode->
                        memory){
                        printf("Moving NEXTFITNODE due to coalace taking place.\n
630                            ");
                        NEXTFITNODE = connectedHolesPointer;
632                    }
                }

634

                //if the next next node is NULL
                if (connectedHolesPointer->nextNode->nextNode == NULL){
636                    //Assign a new node pointer to current nodes next next node.
                    Node *nextNode = connectedHolesPointer->nextNode->nextNode;
638

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

```

```

644     }

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

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

652     }

654 }

656 else{

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

        }

660 }

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

666     }

668 }

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

672 }
/*
674  * Inputs: void
676  * Outputs: void
678  * Description: Outputs all nodes.
680  */
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:
686                %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                    point->memory);

            break;
        }else{
690            //Output information about a node.
            printf("Node \n size: %d\n free: %d\n node start: %d\n memory start:
692                %d\n\n",
                    (int)point->size,point->free,(point->memory-sizeof(Node)),
                    point->memory);

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

696     }

698 }

700
702 /*
    * Inputs: void.

```

```

    * Outputs: void.
704  * Description: Used by threads that act as a library to randomly allocate random
        amounts of memory.
    */
706 void* threadAllocate(){
    srand(time(NULL));
708     for (int i = 0; i < 500; i++) {
        size_t allocateAmount = rand() % 200;
710         printf("ID: %d, trying to allocate: %d.\n",pthread_self(),allocateAmount)
            ;
        void *x = allocate(allocateAmount);
712     }
}
714

716 /*
    * Inputs: void.
718  * Outputs: void.
    * Description: Used by threads that act as a library to randomly deallocate
        random amounts of memory.
    */
720 void* threadDeallocate(void* memory){
722     srand(time(NULL));
    for (int i=0; i<800; i++){
724         int x = rand() % ((memory+1024) + 1 - memory) + memory;
        deallocate((void*)x);
726     }
}
728

730 int main() {

732     //Set the initial heap size.
    size_t size = 1024;
734

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

738     //TEST 1
    /*
740     //Set the algorithm type.
    char * algo = "FirstFit";
742

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

746     //Output nodes.
    output();
748

    //Create and join all threads.
750     //This creates them and makes them run their respective library functions.
    //They act as libraries trying to allocate memory from themselves using this
        program.
752     //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,
754         thread10,thread11,thread12,thread13,thread14,thread15;
    pthread_create(&thread1, NULL, threadAllocate, NULL);
756     pthread_create(&thread2, NULL, threadAllocate, NULL);
    pthread_create(&thread3, NULL, threadAllocate, NULL);
758     pthread_create(&thread4, NULL, threadAllocate, NULL);
    pthread_create(&thread5, NULL, threadAllocate, NULL);
760     pthread_create(&thread6, NULL, threadAllocate, NULL);

```

```

pthread_create(&thread7, NULL, threadAllocate, NULL);
762 pthread_create(&thread8, NULL, threadAllocate, NULL);
pthread_create(&thread9, NULL, threadDeallocate, memory);
764 pthread_create(&thread10, NULL, threadDeallocate, memory);
pthread_create(&thread11, NULL, threadDeallocate, memory);
766 pthread_create(&thread12, NULL, threadDeallocate, memory);
pthread_create(&thread13, NULL, threadDeallocate, memory);
768 pthread_create(&thread14, NULL, threadDeallocate, memory);
pthread_create(&thread15, NULL, threadDeallocate, memory);
770

pthread_join(thread1, NULL);
772 pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
774 pthread_join(thread4, NULL);
pthread_join(thread5, NULL);
776 pthread_join(thread6, NULL);
pthread_join(thread7, NULL);
778 pthread_join(thread8, NULL);
pthread_join(thread9, NULL);
780 pthread_join(thread10, NULL);
pthread_join(thread11, NULL);
782 pthread_join(thread12, NULL);
pthread_join(thread13, NULL);
784 pthread_join(thread14, NULL);
pthread_join(thread15, NULL);
786

//Final node output.
printf("\n\n\n\nFinal node output:\n\n");
788 output();
/*
790 //TEST 2
/*
792 //Set the algorithm type.
char * algo = "NextFit";
794

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

//Output nodes.
800 output();
802

//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.
804 //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,
806 thread10, thread11, thread12, thread13, thread14, thread15;
pthread_create(&thread1, NULL, threadAllocate, NULL);
808 pthread_create(&thread2, NULL, threadAllocate, NULL);
pthread_create(&thread3, NULL, threadAllocate, NULL);
810 pthread_create(&thread4, NULL, threadAllocate, NULL);
pthread_create(&thread5, NULL, threadAllocate, NULL);
812 pthread_create(&thread6, NULL, threadAllocate, NULL);
pthread_create(&thread7, NULL, threadAllocate, NULL);
814 pthread_create(&thread8, NULL, threadAllocate, NULL);
pthread_create(&thread9, NULL, threadDeallocate, memory);
816 pthread_create(&thread10, NULL, threadDeallocate, memory);
pthread_create(&thread11, NULL, threadDeallocate, memory);
818 pthread_create(&thread12, NULL, threadDeallocate, memory);
pthread_create(&thread13, NULL, threadDeallocate, memory);
820

```

```

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\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);

```

```

884 pthread_join(thread5, NULL);
      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\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);
945     pthread_join(thread13, NULL);
946     pthread_join(thread14, NULL);
947     pthread_join(thread15, NULL);
948
949
950     //Final node output.
951     printf("\n\n\n\nFinal node output:\n\n");
952     output();
953     */
954
955     free(memory);
956     return EXIT_SUCCESS;
}
```