

# SFWRENG 3SH3

## Lab3 Report

Ruiyan Guo 400256752

## Program Explanation

### (1). Assisting function - insert

```
10 void insert(int* arr, int pos_ini, int counter)
11 {
12     for(int i = pos_ini; i < (pos_ini + counter); i++)
13     {
14         //std::cout << "Filling position " << i << "\n";
15         arr[i] = 8;
16     }
17 }
```

This function is used for inserting items into a specific range of positions. It will be called inside the Allocation function for first fit, best fit and worst fit. The function takes three inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'pos\_ini' is the start position for the insertion. (3) 'counter' is the total number of elements needed to be inserted. To distinguish the difference between a filled place and holes, we fill all the filled space with integer 8 and all the holes have a value of 0.

### (2). Allocation function - First fit

```
18 //***** Begin First*****//
19 void allocate_first(int* arr, std::vector<int> &list_allocation)
20 {
21     std::cout << "Entering allocate_first\n";
22     int lower = 1024;
23     int upper = 25600;
24     int each_time = 0;
25
26     int counter = 0;
27     int check = 0;
28
29     std::vector<int> pos_initial;
30 }
```

```

31 while (check == 0)
32 {
33     check = 1;
34     each_time = (rand() % (upper - lower + 1)) + lower; //each_time = random number between 4KB and 100KB
35
36     for(int i = 0; i < 262144; i++)
37     {
38         if (arr[i] == 0)
39         {
40             pos_initial.push_back(i); //Mark the index where first hole show up
41             counter++; // Keep counting the size of the hole
42             if(each_time <= counter)
43             {
44                 list_allocation.push_back(pos_initial[0]);
45                 list_allocation.push_back(pos_initial[0]+each_time-1);
46                 insert(arr, pos_initial[0], each_time);
47                 pos_initial.clear();
48                 pos_initial.resize(0);
49                 counter = 0;
50                 check = 0;
51                 break; //break for
52             }
53         }
54         else
55         {
56             pos_initial.clear();
57             pos_initial.resize(0);
58             counter = 0;
59         }
60     } //end for
61 } //end while
62 std::cout << "Exiting allocate_first\n";
63 }
64 //***** End First*****//

```

This function implements the functionality of the First fit algorithm. The function takes 2 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'list\_allocation' is the vector to store the list of allocations currently in the memory, it holds two properties: the start position of the filled space and the end position.

Variable 'each\_time' holds a randomly generated number between 4KB and 100KB. The outer while loop makes sure that the function continues searching for the opportunity to implement the allocation, the variable 'check' controls the termination of this function - when the allocation process encounters the first failure. The inner for loop searches the memory to find the first fit hole, 'pos\_initial' records the start position of the first fit hole, the 'counter' holds the size of the first fit hole. The if condition implements the insertion immediately once the size of the hole could fit the process. The else branch is when the memory search encounters a filled space, in this case, the 'counter' and 'pos\_initial' need to start over.

### (3). Allocation function - Best fit

```

65 //***** Begin Best*****//
66 void allocate_best(int* arr, std::vector<int> &list_allocation)
67 {
68     std::cout << "Entering allocate_best\n";
69     int lower = 1024;
70     int upper = 25600;
71     int each_time = 0;
72
73     int check = 0;
74     int counter = 0;
75     std::vector<int> pos_initial;
76     std::vector<int> temp;//To save the scan result
77
78     while (check == 0)
79     {
80         check = 1;
81         each_time = (rand() % (upper - lower + 1)) + lower;
82         //Start scanning, store all the holes in temp
83         for(int i = 0; i < 262144; i++)
84         {
85             if(arr[i] == 0)
86             {
87                 pos_initial.push_back(i);
88                 counter++;
89             }
90             else
91             {
92                 if(counter != 0)
93                 {
94                     temp.push_back(pos_initial[0]);
95                     temp.push_back(pos_initial[0]+counter-1);
96                     temp.push_back(counter);
97                     pos_initial.clear();
98                     pos_initial.resize(0);
99                     counter = 0;
100                     continue;
101                 }
102             }
103             if((arr[i] == 0)&&(i == 262143))
104             {
105                 temp.push_back(pos_initial[0]);
106                 temp.push_back(pos_initial[0]+counter-1);
107                 temp.push_back(counter);
108             }
109             }//end for
110
111         pos_initial.clear();
112         pos_initial.resize(0);
113         counter = 0;
114         //Start search for the closest fit
115         int temp_size = temp.size();
116         int bestfit = 262144;
117         int bestidx = 0;
118         for(int i = 2; i <= (temp_size - 1); i+=3)
119         {
120             if((each_time <= temp[i]) && (temp[i] <= bestfit))
121             {
122                 bestfit = temp[i];
123                 bestidx = i;
124                 check = 0;
125             }
126         }
127         //Start inserting
128         if(check == 0)
129         {
130             list_allocation.push_back(temp[bestidx-2]);//store start
131             list_allocation.push_back((temp[bestidx-2]+each_time-1));//store end
132             insert(arr, temp[bestidx-2], each_time);
133         }
134         temp.clear();
135         temp.resize(0);
136     }//end while
137     std::cout << "Exiting allocate_best\n";
138 }
139 //***** End Best*****//

```

This function implements the functionality of the Best fit algorithm. The function takes 2 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'list\_allocation' is

the vector to store the list of allocations currently in the memory, it holds two properties: the start position of the filled space and the end position.

The outer while loop makes sure that the function continues searching for the opportunity to implement the allocation, the variable 'check' controls the termination of this function - when the allocation process encounters the first failure. Inside the while loop, it could be divided into 3 sections. (1) Scanning the entire memory and store all the hole locations inside a vector 'temp', each hole location has 3 properties: start position, end position and size, these 3 properties are all stored in the vector 'temp'. (2) From the information stored inside vector 'temp', determine which hole size is closest to the size of the process, and then invoke the corresponding start/end position property from the vector. (3) Start allocating the process into the memory according to the start/end position obtained in step 2.

#### (4). Allocation function - Worst fit

```
139 //***** Begin Worst*****//
140 void allocate_worst(int* arr, std::vector<int> &list_allocation)
141 {
142     std::cout << "Entering allocate_worst\n";
143     int lower = 1024;
144     int upper = 25600;
145     int each_time = 0;
146
147     int check = 0;
148     int counter = 0;
149     std::vector<int> pos_initial;
150     std::vector<int> temp;//To save the scan result
151
152     while (check == 0)
153     {
154         check = 1;
155         each_time = (rand() % (upper - lower + 1)) + lower;
156         //Start scanning, store all the holes in temp
157         for(int i = 0; i < 262144; i++)
158         {
159             if(arr[i] == 0)
160             {
161                 pos_initial.push_back(i);
162                 counter++;
163             }
164             else
165             {
166                 if(counter != 0)
167                 {
168                     temp.push_back(pos_initial[0]);
169                     temp.push_back(pos_initial[0]+counter-1);
170                     temp.push_back(counter);
171                     pos_initial.clear();
172                     pos_initial.resize(0);
173                     counter = 0;
174                     continue;
175                 }
176             }
177             if((arr[i] == 0)&&(i == 262143))
178             {
179                 temp.push_back(pos_initial[0]);
180                 temp.push_back(pos_initial[0]+counter-1);
181                 temp.push_back(counter);
182             }
183         }
184     }
185 }
```

```

184     pos_initial.clear();
185     pos_initial.resize(0);
186     counter = 0;
187     //Start search for the largest fit
188     int temp_size = temp.size();
189     int worstfit = 0;
190     int worstidx = 0;
191     for(int i = 2; i <= (temp_size - 1); i+=3)
192     {
193         if((each_time <= temp[i])&&(worstfit <= temp[i]))
194         {
195             worstfit = temp[i];
196             worstidx = i;
197             check = 0;
198         }
199     }
200     //Start inserting
201     if(check == 0)
202     {
203         list_allocation.push_back(temp[worstidx-2]); //store start
204         list_allocation.push_back((temp[worstidx-2]+each_time-1)); //store end
205         insert(arr, temp[worstidx-2], each_time);
206     }
207     temp.clear();
208     temp.resize(0);
209 } //end while
210 std::cout << "Exiting allocate_worst\n";
211 }
212 //***** End Worst*****//

```

This function implements the functionality of the Worst fit algorithm. The function takes 2 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'list\_allocation' is the vector to store the list of allocations currently in the memory, it holds two properties: the start position of the filled space and the end position.

The outer while loop makes sure that the function continues searching for the opportunity to implement the allocation, the variable 'check' controls the termination of this function - when the allocation process encounters the first failure. Inside the while loop, it could be divided into 3 sections. (1) Scanning the entire memory and store all the hole locations inside a vector 'temp', each hole location has 3 properties: start position, end position and size, these 3 properties are all stored in the vector 'temp'. (2) From the information stored inside vector 'temp', determine which hole size is the biggest to fit the process, and then invoke the corresponding start/end position property from the vector. (3) Start allocating the processes into the memory according to the start/end position obtained in step 2.

## (5). Function - Release

```
213 void release(int* arr, std::vector<int> &list_allocation)
214 {
215     std::cout << "Entering release\n";
216     float totalprocess = (list_allocation.size())/2; //Total processes allocated
217     float processtoremove = round(totalprocess/10); //number of processes to remove
218
219     int lower = 1;
220     int upper = totalprocess;
221     int number = 0;
222     int start = 0;
223     int end = 0;
224
225     std::vector<int> avoid_dup;
226     avoid_dup.push_back(0);
227     std::vector<int>::iterator temp;
228
229     std::cout << "Total processes = " << totalprocess << "\n";
230     std::cout << "Total process to remove = " << processtoremove << "\n";
231     for(int i = 0; i < processtoremove; i++)
232     {
233         while(1)
234         {
235             number = (rand() % (upper - lower + 1)) + lower; //which process to remove (it's a random number)
236             temp = find (avoid_dup.begin(), avoid_dup.end(), number);
237             if(temp == avoid_dup.end())
238                 {avoid_dup.push_back(number); break;}
239         }
240         std::cout << "Removing process " << number << "\n";
241         start = number*2-2;
242         end = number*2-1;
243         std::cout << "Removing from poistion " << list_allocation[start] << " to position " << list_allocation[end] << "\n";
244         for(int j = list_allocation[start]; j <= list_allocation[end]; j++)
245         {
246             arr[j] = 0;
247         }
248
249         for(int j = list_allocation[start]; j <= list_allocation[end]; j++)
250         {
251             arr[j] = 0;
252         }
253     }
254     std::cout << "Exiting release\n";
255 }
```

This function implements the functionality of the Release algorithm. The function takes 2 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'list\_allocation' is the vector containing the list of allocations currently in the memory, it holds two properties: the start position of the filled space and the end position.

The function uses the information stored in 'list\_allocation' to determine the total number of processes allocated, and then calculates the number of processes that need to be removed. It will generate random numbers within the range of the number of total processes, to decide which processes to remove randomly. Notice that, the while loop inside the function will make sure that the random generated number each time will not be the same. Then, the function will remove the processes by inserting 0 to the corresponding location.

## (6). Function - Compaction

```
252 void compaction(int* arr, std::vector<int> &list_allocation)
253 {
254     int arr_size = 262144;
255     int count_fill = 0;
256     //Count the total numbers of filled locations
257     for(int i = 0; i <= (list_allocation.size() - 2); i+=2)
258     {
259         count_fill = (count_fill + (list_allocation[i+1] - list_allocation[i] + 1));
260     }
261     //Compaction
262     for(int i = 0; i < 262144; i++)
263     {
264         if(i < count_fill)
265             {arr[i] = 8;}
266         else
267             {arr[i] = 0;}
268     }
269 }
```

This function implements the functionality of the Release algorithm. The function takes 2 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'list\_allocation' is the vector containing the list of allocations currently in the memory, it holds two properties: the start position of the filled space and the end position.

The function counts all the filled locations and stores the total number into 'count\_fill', and re-assign the corresponding location with integer 8, and compacted hole area with integer 0.

## (7). Function - Status

```
271 void status(int* arr, std::vector<int> &holes, std::vector<int> &filled)
272 {
273     std::vector<int> hole_initial;
274     int hole_counter = 0;
275
276     std::vector<int> fill_initial;
277     int fill_counter = 0;
278     int total_fill = 0;
```



```

279 //Search and store hole & filled location
280 for(int i = 0; i < 262144; i++)
281 {
282     if(arr[i] == 0)
283     {
284         hole_initial.push_back(i);
285         hole_counter++;
286         if(fill_counter != 0)
287         {
288             filled.push_back(fill_initial[0]);
289             filled.push_back(fill_initial[0]+fill_counter-1);
290             fill_counter = 0;
291             fill_initial.clear();
292             fill_initial.resize(0);
293         }
294     }
295     else
296     {
297         fill_initial.push_back(i);
298         fill_counter++;
299
300         if(hole_counter != 0)
301         {
302             holes.push_back(hole_initial[0]);
303             holes.push_back(hole_initial[0]+hole_counter-1);
304             hole_counter = 0;
305             hole_initial.clear();
306             hole_initial.resize(0);
307         }
308     }
309 } //end for

```

```

311 //Print holes status
312 int hole_arr_size = holes.size();
313 for(int i = 0; i <= (hole_arr_size-2); i+=2)
314 {
315     std::cout << "Hole from position " << holes[i] << " to position " << holes[i+1] << "\n";
316 }
317
318 //Print filled status
319 int fill_arr_size = filled.size();
320 for(int i = 0; i <= (fill_arr_size-2); i+=2)
321 {
322     total_fill = total_fill + (filled[i+1] - filled[i] + 1);
323     std::cout << "Filled space from position " << filled[i] << " to position " << filled[i+1] << "\n";
324 }
325
326 int convert_byte = total_fill*4;
327 std::cout << "There are " << total_fill << " / " << "262144" << " space filled\n";
328 std::cout << "Represent in Bytes, there are " << convert_byte << "Bytes / " << "1048576Bytes(1MB)" << " space filled\n";
329 }

```

This function prints the memory status. The function takes 3 inputs. (1) 'arr' is the 1MB memory location we allocated. (2) 'holes' is the vector to store the location properties of holes. (3) 'filled' is the vector to store the location properties of filled locations. Both holes/filled hold 3 properties: start position, end position and size.

The function searches the entire memory and writes the corresponding location properties to holes/filled. Then all the position information as well as the total memory allocation compared to 1MB are printed.

## (8). Function - Main

The main initializes the 1MB memory space, and then calls the functions described above. The function call orders are the same as described in the lab manual. The main function will automatically print out all the three First fit, Best fit and Worst fit.

## Running Results

```

[RYdeMacBook-Pro:3SH3 ry$ g++ lab3.cpp -o lab3
[RYdeMacBook-Pro:3SH3 ry$ ./lab3
----- Printing First Fit -----
***** Initial filling *****
Entering allocate_first
Exiting allocate_first
Entering release
Total processes = 20
Total process to remove = 2
Removing process 10
Removing from poistion 108661 to position 119226
Removing process 18
Removing from poistion 216688 to position 233696
Exiting release
Hole from position 108661 to position 119226
Hole from position 216688 to position 233696
Filled space from position 0 to position 108660
Filled space from position 119227 to position 216687
Filled space from position 233697 to position 248862
There are 221288 / 262144 space filled
Represent in Bytes, there are 885152Bytes / 1048576Bytes(1MB) space filled
***** Non-compaction *****
Entering allocate_first
Exiting allocate_first
Hole from position 114173 to position 119226
Hole from position 216688 to position 233696
Filled space from position 0 to position 114172
Filled space from position 119227 to position 216687
Filled space from position 233697 to position 248862
There are 226800 / 262144 space filled
Represent in Bytes, there are 907200Bytes / 1048576Bytes(1MB) space filled
***** Compaction *****
Entering allocate_first
Exiting allocate_first
Filled space from position 0 to position 248862
There are 248863 / 262144 space filled
Represent in Bytes, there are 995452Bytes / 1048576Bytes(1MB) space filled
----- Printing Best Fit -----
----- Printing Best Fit -----
***** Initial filling *****
Entering allocate_best
Exiting allocate_best
Entering release
Total processes = 20
Total process to remove = 2
Removing process 15
Removing from poistion 183371 to position 185885
Removing process 18
Removing from poistion 212973 to position 220025
Exiting release
Hole from position 183371 to position 185885
Hole from position 212973 to position 220025
Filled space from position 0 to position 183370
Filled space from position 185886 to position 212972
Filled space from position 220026 to position 245889
There are 236322 / 262144 space filled
Represent in Bytes, there are 945288Bytes / 1048576Bytes(1MB) space filled
***** Non-compaction *****
Entering allocate_best
Exiting allocate_best
Hole from position 183371 to position 185885
Hole from position 217439 to position 220025
Filled space from position 0 to position 183370
Filled space from position 185886 to position 217438
Filled space from position 220026 to position 261916
There are 256815 / 262144 space filled
Represent in Bytes, there are 1027260Bytes / 1048576Bytes(1MB) space filled
***** Compaction *****
Entering allocate_best
Exiting allocate_best
Filled space from position 0 to position 245889
There are 245890 / 262144 space filled
Represent in Bytes, there are 983560Bytes / 1048576Bytes(1MB) space filled
----- Printing Worst Fit -----
----- Printing Worst Fit -----
```

```

----- Printing Worst Fit -----
***** Initial filling *****
Entering allocate_worst
Exiting allocate_worst
Entering release
Total processes = 22
Total process to remove = 2
Removing process 13
Removing from poistion 161344 to position 165209
Removing process 6
Removing from poistion 64295 to position 89314
Exiting release
Hole from position 64295 to position 89314
Hole from position 161344 to position 165209
Filled space from position 0 to position 64294
Filled space from position 89315 to position 161343
Filled space from position 165210 to position 255221
There are 226336 / 262144 space filled
Represent in Bytes, there are 905344Bytes / 1048576Bytes(1MB) space filled
***** Non-compaction *****
Entering allocate_worst
Exiting allocate_worst
Hole from position 84042 to position 89314
Hole from position 161344 to position 165209
Filled space from position 0 to position 84041
Filled space from position 89315 to position 161343
Filled space from position 165210 to position 255221
There are 246083 / 262144 space filled
Represent in Bytes, there are 984332Bytes / 1048576Bytes(1MB) space filled
***** Compaction *****
Entering allocate_worst
Exiting allocate_worst
Filled space from position 0 to position 259881
There are 259882 / 262144 space filled
Represent in Bytes, there are 1039528Bytes / 1048576Bytes(1MB) space filled
(base) RYdeMacBook-Pro:3SH3 ry$

```

## Discussion:

According to the running results shown above, we can generally say that the compaction will improve the memory utilization, since all the three methods all show that more memory can be utilized after compaction. But it is very hard to determine which method is the best, since the memory utilizations are all very close.

## Codes

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <vector>
#include <tgmath.h>
#include <iostream>
#include <fstream>
#include <algorithm>

void insert(int* arr, int pos_ini, int counter)
{
    for(int i = pos_ini; i < (pos_ini + counter); i++)
    {
        //std::cout << "Filling position " << i << "\n";
        arr[i] = 8;
    }
}

//***** Begin First*****//
void allocate_first(int* arr, std::vector<int> &list_allocation)
{
    std::cout << "Entering allocate_first\n";
    int lower = 1024;
    int upper = 25600;
    int each_time = 0;

```

```

int counter = 0;
int check = 0;

std::vector<int> pos_initial;

while (check == 0)
{
    check = 1;
    each_time = (rand() % (upper - lower + 1)) + lower; //each_time = random number
between 4KB and 100KB

    for(int i = 0; i < 262144; i++)
    {
        if (arr[i] == 0)
        {
            pos_initial.push_back(i); //Mark the index where first hole show up
            counter++; // Keep counting the size of the hole
            if(each_time <= counter)
            {
                list_allocation.push_back(pos_initial[0]);
                list_allocation.push_back(pos_initial[0]+each_time-1);
                insert(arr, pos_initial[0], each_time);
                pos_initial.clear();
                pos_initial.resize(0);
                counter = 0;
                check = 0;
                break; //break for
            }
        }
        else
        {
            pos_initial.clear();
            pos_initial.resize(0);
            counter = 0;
        }
    } //end for
} //end while
std::cout << "Exiting allocate_first\n";
}
//***** End First*****//
//***** Begin Best*****//
void allocate_best(int* arr, std::vector<int> &list_allocation)
{
    std::cout << "Entering allocate_best\n";
    int lower = 1024;
    int upper = 25600;
    int each_time = 0;

    int check = 0;
    int counter = 0;
    std::vector<int> pos_initial;
    std::vector<int> temp; //To save the scan result

    while (check == 0)
    {
        check = 1;
        each_time = (rand() % (upper - lower + 1)) + lower;
        //Start scanning, store all the holes in temp

```

```

for(int i = 0; i < 262144; i++)
{
    if(arr[i] == 0)
    {
        pos_initial.push_back(i);
        counter++;
    }
    else
    {
        if(counter != 0)
        {
            temp.push_back(pos_initial[0]);
            temp.push_back(pos_initial[0]+counter-1);
            temp.push_back(counter);
            pos_initial.clear();
            pos_initial.resize(0);
            counter = 0;
            continue;
        }
    }
    if((arr[i] == 0)&&(i == 262143))
    {
        temp.push_back(pos_initial[0]);
        temp.push_back(pos_initial[0]+counter-1);
        temp.push_back(counter);
    }
}
//end for
pos_initial.clear();
pos_initial.resize(0);
counter = 0;
//Start search for the closest fit
int temp_size = temp.size();
int bestfit = 262144;
int bestidx = 0;
for(int i = 2; i <= (temp_size - 1); i+=3)
{
    if((each_time <= temp[i]) && (temp[i] <= bestfit))
    {
        bestfit = temp[i];
        bestidx = i;
        check = 0;
    }
}
//Start inserting
if(check == 0)
{
    list_allocation.push_back(temp[bestidx-2]); //store start
    list_allocation.push_back((temp[bestidx-2]+each_time-1)); //store end
    insert(arr, temp[bestidx-2], each_time);
}
temp.clear();
temp.resize(0);
} //end while
std::cout << "Exiting allocate_best\n";
}
//***** End Best*****//
//***** Begin Worst*****//
void allocate_worst(int* arr, std::vector<int> &list_allocation)
{

```

```

std::cout << "Entering allocate_worst\n";
int lower = 1024;
int upper = 25600;
int each_time = 0;

int check = 0;
int counter = 0;
std::vector<int> pos_initial;
std::vector<int> temp;//To save the scan result

while (check == 0)
{
    check = 1;
    each_time = (rand() % (upper - lower + 1)) + lower;
    //Start scanning, store all the holes in temp
    for(int i = 0; i < 262144; i++)
    {
        if(arr[i] == 0)
        {
            pos_initial.push_back(i);
            counter++;
        }
        else
        {
            if(counter != 0)
            {
                temp.push_back(pos_initial[0]);
                temp.push_back(pos_initial[0]+counter-1);
                temp.push_back(counter);
                pos_initial.clear();
                pos_initial.resize(0);
                counter = 0;
                continue;
            }
        }
        if((arr[i] == 0)&&(i == 262143))
        {
            temp.push_back(pos_initial[0]);
            temp.push_back(pos_initial[0]+counter-1);
            temp.push_back(counter);
        }
    }
    //end for
    pos_initial.clear();
    pos_initial.resize(0);
    counter = 0;
    //Start search for the largest fit
    int temp_size = temp.size();
    int worstfit = 0;
    int worstidx = 0;
    for(int i = 2; i <= (temp_size - 1); i+=3)
    {
        if((each_time <= temp[i])&&(worstfit <= temp[i]))
        {
            worstfit = temp[i];
            worstidx = i;
            check = 0;
        }
    }
    //Start inserting

```

```

        if(check == 0)
        {
            list_allocation.push_back(temp[worstidx-2]); //store start
            list_allocation.push_back((temp[worstidx-2]+each_time-1)); //store end
            insert(arr, temp[worstidx-2], each_time);
        }
        temp.clear();
        temp.resize(0);
    } //end while
    std::cout << "Exiting allocate_worst\n";
}
//***** End Worst*****//
void release(int* arr, std::vector<int> &list_allocation)
{
    std::cout << "Entering release\n";
    float totalprocess = (list_allocation.size())/2; //Total processes allocated
    float processtoremove = round(totalprocess/10); //number of processes to remove

    int lower = 1;
    int upper = totalprocess;
    int number = 0;
    int start = 0;
    int end = 0;

    std::vector<int> avoid_dup;
    avoid_dup.push_back(0);
    std::vector<int>::iterator temp;

    std::cout << "Total processes = " << totalprocess << "\n";
    std::cout << "Total process to remove = " << processtoremove << "\n";
    for(int i = 0; i < processtoremove; i++)
    {
        while(1)
        {
            number = (rand() % (upper - lower + 1)) + lower; //which process to remove
            (it's a random number)
            temp = find (avoid_dup.begin(), avoid_dup.end(), number);
            if(temp == avoid_dup.end())
                {avoid_dup.push_back(number); break;}
        }
        std::cout << "Removing process " << number << "\n";
        start = number*2-2;
        end = number*2-1;
        std::cout << "Removing from poistion " << list_allocation[start] << " to
position "<< list_allocation[end] << "\n";
        for(int j = list_allocation[start]; j <= list_allocation[end]; j++)
        {
            arr[j] = 0;
        }
    }
    std::cout << "Exiting release\n";
}

void compaction(int* arr, std::vector<int> &list_allocation)
{
    int arr_size = 262144;
    int count_fill = 0;
    //Count the total numbers of filled locations
    for(int i = 0; i <= (list_allocation.size() - 2); i+=2)

```

```

        {
            count_fill = (count_fill + (list_allocation[i+1] - list_allocation[i] + 1));
        }
    //Compaction
    for(int i = 0; i < 262144; i++)
    {
        if(i < count_fill)
            {arr[i] = 8;}
        else
            {arr[i] = 0;}
    }
}

void status(int* arr, std::vector<int> &holes, std::vector<int> &filled)
{
    std::vector<int> hole_initial;
    int hole_counter = 0;

    std::vector<int> fill_initial;
    int fill_counter = 0;
    int total_fill = 0;
    //Search and store hole & filled location
    for(int i = 0; i < 262144; i++)
    {
        if(arr[i] == 0)
        {
            hole_initial.push_back(i);
            hole_counter++;
            if(fill_counter != 0)
            {
                filled.push_back(fill_initial[0]);
                filled.push_back(fill_initial[0]+fill_counter-1);
                fill_counter = 0;
                fill_initial.clear();
                fill_initial.resize(0);
            }
        }
        else
        {
            fill_initial.push_back(i);
            fill_counter++;

            if(hole_counter != 0)
            {
                holes.push_back(hole_initial[0]);
                holes.push_back(hole_initial[0]+hole_counter-1);
                hole_counter = 0;
                hole_initial.clear();
                hole_initial.resize(0);
            }
        }
    }
    //end for

    //Print holes status
    int hole_arr_size = holes.size();
    for(int i = 0; i <= (hole_arr_size-2); i+=2)
    {
        std::cout << "Hole from position " << holes[i] << " to position " <<
holes[i+1] << "\n";
    }
}

```



```

    }

    //Print filled status
    int fill_arr_size = filled.size();
    for(int i = 0; i <= (fill_arr_size-2); i+=2)
    {
        total_fill = total_fill + (filled[i+1] - filled[i] + 1);
        std::cout << "Filled space from position " << filled[i] << " to position "
<< filled[i+1] << "\n";
    }

    int convert_byte = total_fill*4;
    std::cout << "There are " << total_fill << " / " << "262144" << " space
filled\n";
    std::cout << "Represent in Bytes, there are " << convert_byte << "Bytes / "
<< "1048576Bytes(1MB)" << " space filled\n";
}

int main()
{
    int n = 262144; // 1048576/4

    // First Fit
    std::cout << "----- Printing First Fit ----- \n";
    int* mem01 = (int*) calloc(n, sizeof(int)); //initialize 1MB memory
    std::cout << "***** Initial filling *****\n";
    std::vector<int> list_allocation;
    std::vector<int> holes;
    std::vector<int> filled;
    allocate_first(mem01, list_allocation);
    release(mem01, list_allocation);
    status(mem01, holes, filled);
    int* mem01_dup = (int*) calloc(n, sizeof(int));
    std::vector<int> list_allocation_dup;
    for(int i = 0; i < n; i++)
        {mem01_dup[i] = mem01[i];}
    for(int i = 0; i < list_allocation.size(); i++)
        {
            list_allocation_dup.push_back(list_allocation[i]);
        }
    //For non-compaction
    std::cout << "***** Non-compaction *****\n";
    list_allocation.clear();
    list_allocation.resize(0);
    holes.clear();
    holes.resize(0);
    filled.clear();
    filled.resize(0);
    allocate_first(mem01, list_allocation);
    status(mem01, holes, filled);
    //For compaction
    std::cout << "***** Compaction *****\n";
    compaction(mem01_dup, list_allocation_dup);
    list_allocation.clear();
    list_allocation.resize(0);
    holes.clear();
    holes.resize(0);
    filled.clear();

```

```

filled.resize(0);
allocate_first(memo1_dup, list_allocation_dup);
status(memo1_dup, holes, filled);
list_allocation.clear();
list_allocation.resize(0);
list_allocation_dup.clear();
list_allocation_dup.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
free(memo1);
free(memo1_dup);

std::cout << "----- Printing Best Fit -----\\n";
int* memo2 = (int*) calloc(n, sizeof(int)); //initialize 1MB memory
// Best Fit
std::cout << "***** Initial filling *****\\n";
allocate_best(memo2, list_allocation);
release(memo2, list_allocation);
status(memo2, holes, filled);
int* memo2_dup = (int*) calloc(n, sizeof(int));
for(int i = 0; i < n; i++)
    {memo2_dup[i] = memo2[i];}
for(int i = 0; i < list_allocation.size(); i++)
    {
        list_allocation_dup.push_back(list_allocation[i]);
    }
//For non-compaction
std::cout << "***** Non-compaction *****\\n";
list_allocation.clear();
list_allocation.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
allocate_best(memo2, list_allocation);
status(memo2, holes, filled);
//For compaction
std::cout << "***** Compaction *****\\n";
compaction(memo2_dup, list_allocation_dup);
list_allocation.clear();
list_allocation.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
allocate_best(memo2_dup, list_allocation_dup);
status(memo2_dup, holes, filled);
list_allocation.clear();
list_allocation.resize(0);
list_allocation_dup.clear();
list_allocation_dup.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
free(memo2);
free(memo2_dup);

```

```

std::cout << "----- Printing Worst Fit -----\\n";
int* memo3 = (int*) calloc(n, sizeof(int)); //initialize 1MB memory
// Worst Fit
std::cout << "***** Initial filling *****\\n";
allocate_worst(memo3, list_allocation);
release(memo3, list_allocation);
status(memo3, holes, filled);
int* memo3_dup = (int*) calloc(n, sizeof(int));
for(int i = 0; i < n; i++)
    {memo3_dup[i] = memo3[i];}
for(int i = 0; i < list_allocation.size(); i++)
    {
        list_allocation_dup.push_back(list_allocation[i]);
    }
//For non-compaction
std::cout << "***** Non-compaction *****\\n";
list_allocation.clear();
list_allocation.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
allocate_worst(memo3, list_allocation);
status(memo3, holes, filled);
//For compaction
std::cout << "***** Compaction *****\\n";
compaction(memo3_dup, list_allocation_dup);
list_allocation.clear();
list_allocation.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
allocate_worst(memo3_dup, list_allocation_dup);
status(memo3_dup, holes, filled);
list_allocation.clear();
list_allocation.resize(0);
list_allocation_dup.clear();
list_allocation_dup.resize(0);
holes.clear();
holes.resize(0);
filled.clear();
filled.resize(0);
free(memo3);
free(memo3_dup);

return 0;
}

```