CLEVELAND STATE UNIVERSITY

***Class*** **=** *CIS340 (Systems Programming)*
***Assignment Title*** **=** *Term Project*
***Professor*** **=** *Dr.Shirazi*
***Semester*** **=** *Fall 2017*
***Total Logged Hours*** **=** *39.3 hrs*
***Labor Start Date*** **=** *10/12/17*
***Labor Finish Date*** **=** *11/25/17*
***Due Date*** **=** *12/06/17*
***Student Name/ID#*** **=** *Daniel Izadnegahdar / 2420596*
***Student Name/ID#*** **=** *Kendra Dennis / 2637975*

# 1. Report Summary:

*This report discusses 2 projects (a Scheduler and a File Manager). The scheduler is discussed in more detail under section 2.0, and mainly exercises linked lists using pointers. The file manager is discussed in more detail under section 3.0, and mainly exercises command line programming in Linux. The report is broken down into both sections with details on execution instructions, code functionality, and software development experiences.*

---

**Scheduler**
**See provided *scheduler* folder for files.**

# 2. Scheduler:

## 2.1. Summary of code:

*The 1st program is designed to simulate a round-robin scheduler used in inter-process communication systems for operating systems. In this scheduler, every process runs for 1 second, and if the process timer is still more than 1 second, it will remain in memory, otherwise it will be removed. A singly linked list data structure is used for storing these process nodes. Since it is singly, the traversal will be in one direction, through the use of a "next" attribute of the process struct node. Pointers will also be used to traverse through the list. The program will first read a text file to construct a singly linked list of process nodes, and then scan through the list every second, to deduct 1 second from each node, and remove a process if necessary.*

## 2.2. Detail description:

### 2.2.1. Header files (libraries & prototypes):

#### 2.2.1.1. Description:

*The program begins with the header files, which includes a series of libraries that are required for the program to run. These libraries are used to keep the code succinct and easy to read. Each library has a comment next to it, to describe why it was included in the code. For example, the* `<string.h>` *library was included so the* `strlen` *function can be used to get the length of a string. Along with these libraries are also function prototypes. These prototypes follow the signature provided in the assignment description, and are included so the program can run properly.*

#### 2.2.1.2. Code:

```
1 //Libraries
2   #include <string.h> //Used for strlen
3   #include <stdio.h> //Used for standard io
4   #include <stdlib.h> //Used for atoi and exit(0)
5   #include <malloc.h> //Used for malloc
6   #include <stdbool.h> //Used for bool
7   #include <unistd.h> //Used for sleeping
8
9 //Function prototypes
10   struct node* createList();
11   struct node* scanList(struct node* endPointerEntry);
12   struct node* removeCurrent(struct node* currentPointerEntry);
13   void add(int pidEntry, int timerEntry, struct node* endPointerEntry);
14   void printList(struct node* endPointerEntry);
15   bool isEmpty();
```

### *2.2.2. Global variables:*

#### *2.2.2.1. Description:*

*The global variables are visible throughout the entire program. This mainly includes the struct node, the processCount, and iterators. The struct node has 3 attributes: a pid, a timer, and a pointer to its "next" struct. The "next" pointer is used to traverse the singly linked list (If this was a doubly linked list, an additional "prev" attribute would be required to traverse backwards). The pid is a unique ID# for each process and the timer is the duration that this process will run. Those attributes are extracted from the input file.*

*Originally, the current, previous, and end pointers were included as global pointers for simplicity. However, these pointers were removed since the project assignment describes specifically to pass pointers between functions, as opposed to keeping them global.*

#### *2.2.2.2. Code:*

```
17 //Global variables
18   //Master node
19     struct node
20       {//Start of startNode
21         int pid;
22         int timer;
23         struct node *next;
24       };//End of startNode
25
26   //Other Variables
27     int i, j, k;
28     int processCount = 0;
```

### 2.2.3. Functions:

#### 2.2.3.1. Main function:

##### 2.2.3.1.1. Description:

*The main function is relatively short, but the important line is:* `scanList(createList())` *. The rest of the code just prints notifications to indicate when the program starts and ends. The* `createList()` *function returns an end pointer that the scanList uses to scan through the nodes.*

##### 2.2.3.1.2. Code:

```
30 //Functions
31   //Main function(s)
32     int main()
33       {//Start of main
34          //Notify START
35            printf("\n**********START**********\n");
36            printf("**********SCHEDULER**********\n");
37          //IMPORTANT! Create and scan the list
38            scanList(createList());
39          //Notify END
40            printf("**********END**********\n");
41            return 0;
42       }//End of main
```

#### 2.2.3.2. createList function:

##### 2.2.3.2.1. Description:

*The createList function opens and reads the input test1.txt file to extract the pid and timer information. This info is used to add nodes to the linked list structure and return the endPointer.*

*To extract, the function opens the file by using:* `fp = fopen(filename, "r")` *where "r" indicates "read-only" and fp is the returned value used to check if the file was opened properly. Once it opened, each line is analyzed using* `fgets(line, charMax, fp)`*. The 1st string between the space (defined as #32 per ASCII) is collected inside the pidString(char array) and the 2nd word after the space is collected inside the timerString.*

*Before the next line is scanned, the extracted pid/timer information is then added to the linked list. If it's the first node, it will allocate space for itself before inserting process information. If it's a remaining node, it will allocate space to the next node, before inserting process information. To allocate space, the following line was used:* `malloc(sizeof(struct node))`*. Next, the pid and timer information from the arrays is converted to integers using*

*atoi(), and assigned to each added node. The new processes are always added at the end of the list, as shown through the iterative print of the list. After the last process is added, the function returns the endPointer.*

#### *2.2.3.2.2.* ***Code:***

```
44  //Auxiliary function(s)
45    struct node* createList()
46      {//Start of createList
47       //Variables
48         FILE *fp;
49         int split = 0;
50         int charMax = 1000;
51         char pidString[charMax];
52         char timerString[charMax];
53         char line[charMax];
54         char* filename = "test1.txt";
55         struct node *currentPointer;
56         struct node *startNode;
57         struct node *endPointer;
58       //Notify START
59         printf("*****************PART I: CREATE LIST*****************\n");
60       //Open the file
61         fp = fopen(filename, "r"); //Read-only
62       //If file did not open successfully...
63         if(fp == NULL)
64           {//start of if
65             printf("ERROR! File could not be opened!");
66           }//end of if
67       //If file opened successfully...
68         else
69           {//Start of master else
70            //Extract the file information
71              while(fgets(line, charMax, fp) != NULL)
72                {//Start of while fgets
73                 //Reset split
74                   split = 0;
75                 //Separate the pidString and timerString
76                   for(i = 0; i < strlen(line); i++)
77                     {//Start of for
78                      //Split at the space
79                        if(line[i] == 32) //ASCII#32 is a space
80                          {//Start of if
81                            split = 1;
82                            pidString[i] = '\0';
83                          }//End of if
84                      //If splitting 1st string, then its a pidString
85                        if(split == 0)
86                          {//Start of if
87                            pidString[i] = line[i];
88                          }//End of if
89                      //If splitting 2nd string, then its a timerString
90                        else if(split == 1)
91                          {//Start of else if
92                            timerString[i - strlen(pidString)] = line[i + 1]; //+1 to account for space
93                          }//End of else if
94                     }//End of for
95                 //Close the timerString string
96                   timerString[i] = '\0';
97                 //Add the node(after the extraction is complete)
98                   //Increase the processCount
99                     processCount++;
100                  //If it's the 1st node...
101                    if (processCount == 1)
102                      {//start of if
103                       //Set the space for this new startNode
104                         startNode = malloc(sizeof(struct node));
105                       //Assign the values to this new startNode
106                         (*startNode).next = startNode; //Set first one to itself
107                         (*startNode).pid = atoi(pidString);
108                         (*startNode).timer = atoi(timerString);
109                         endPointer = startNode; //Connect the endPointer to the startNode
110                      }//End of if
111                  //If it's a remaining node...
112                    else
113                      {//Start of else
114                         //Traverse until you see the end of the list
115                         //Go to the endPointer
116                           currentPointer = endPointer; //Go to the endPointer
117                         //Add a node
118                           (*currentPointer).next = malloc(sizeof(struct node));
119                           currentPointer = (*currentPointer).next; //index 1 node
120                         //Add the values to this new node
121                           add(atoi(pidString), atoi(timerString), currentPointer);
122                           endPointer = currentPointer; //The endPointer now becomes this new node added
123                           (*endPointer).next = startNode; //connect the endpointer to the startnode in a loop
124                      }//End of else
125                //Notify that node is added
126                  printf("Adding node#%d to list...", processCount);
127                //Print the list
128                  printList(endPointer);
```

#### 2.2.3.3. add function:

##### 2.2.3.3.1. Description:

*The add function assigns the pid and timer information to the provided pointer node. It doesn't return anything.*

##### 2.2.3.3.2. Code:

```
140   //Add the startNode
141     void add(int pidEntry, int timerEntry, struct node* endPointerEntry)
142       {//Start of add
143          (*endPointerEntry).pid = pidEntry;
144          (*endPointerEntry).timer = timerEntry;
145       }//End of add
```

#### 2.2.3.4. printList function:

##### 2.2.3.4.1. Description:

*The printList function prints the content of the linked list. It takes the endPointer to initialize the print location, indexes one to reach the start node, and begins printing and traversing until it reaches the end of the list.*

##### 2.2.3.4.2. Code:

```
147   //Print the listvalue
148     void printList(struct node* PointerEntry)
149       {//Start of printList
150         //Variables
151           struct node *currentPointer;
152         //print text
153           printf("\ncontent:\n");
154         //Only print if the list is not empty
155           if(isEmpty() == false)
156             {//Start of if
157              //Update the currentPointer
158                currentPointer = PointerEntry;
159                currentPointer = (*currentPointer).next; //Go to start
160              //Print the pointer content
161                for (i = 0; i < processCount; i++)
162                   {//Start of for
163                     printf(">NODE(pid:%d / timer:%d)\n", (*currentPointer).pid, (*currentPointer).timer);
164                     currentPointer = (*currentPointer).next; //Index to next pointer
165                   }//End of for
166             }//End of if
167         printf("\n");
168       }//End of printList
```

#### 2.2.3.5. isEmpty function:

##### 2.2.3.5.1. Description:

*The isEmpty() function checks to see if the list is empty. If the processCount is more than zero, than it is not empty, and it returns false, otherwise it returns true.*

#### 2.2.3.5.2. ***Code:***

```
170   //Check that list is empty
171     bool isEmpty()
172       {//Start of isEmtpy
173         if (processCount > 0)
174           {return false;}
175         else
176           {return true;}
177       }//End of isEmpty
```

### 2.2.3.6. ***scanList function:***

#### 2.2.3.6.1. ***Description:***

*The scanList function scans the linked list every second, deducts 1 second from each process timer, and removes a process that has a timer that is less than 1.*

*To begin, the function runs through a while loop that checks if the list is empty. If it is empty, the function exits, otherwise, it runs through the scan routine. By using* `sleep(1)`*, the function waits 1 second, and runs through a for loop to deduct 1 from each process timer.*

*When it enters the "for" loop, it starts by updating a previousPointer and a currentPointer. The previousPointer always lags the currentPointer by one node, and is used primarily for stitching when a process node is removed. Once the pointers are updated, the currentPointer indexes to the next node, and reduces the timer by one node using* `(*currentPointer).timer--` *. As soon as the timer is reduced, it is then assessed to see if it is less than 1. If it is not less than one, the "for" loop continues, otherwise, the process needs to be removed.*

*To remove the process, the function calls the removeCurrent function, which returns the nextPointer of the currentPointer that is to be removed. This nextPointer is going to be used for stitching, because it can now connect to the previousPointer and complete*

*the loop. Once this is done, the "deduct" variable is increased (which is later decremented from processCount), so that on the next "for" cycle, the processCount is updated to iterate for only as long as there are nodes in the linked list.*

#### 2.2.3.6.2. Code:

```
179  //Scan function
180    struct node* scanList(struct node* endPointerEntry)
181      {//Start of scanList
182       //Variables
183         int scanCount = 1;
184        int deduct = 0;
185         struct node *currentPointer;
186         struct node *previousPointer;
187         struct node *nextPointer;
188         currentPointer = endPointerEntry;
189       //Notify start
190         printf("******************PART II: SCAN LIST******************\n");
191       //Print the original list
192         printf("Original list content...");
193         printList(endPointerEntry);
194       //Print remaining nodes
195         printf("Now scanning through list...\n");
196       //Perform scans
197         while (isEmpty() == false)
198           {//Start of while
199              sleep(1);
200              printf("running scan#%d...", scanCount);
201              scanCount++;
202            //Scan the list
203              for(i = 0; i < processCount; i++)
204                {//Start of for
205                   //Update the pointers
206                     previousPointer = currentPointer; //The prevPointer will lag 1 node behind
207                     currentPointer = (*currentPointer).next; //index
208                     (*currentPointer).timer--; //Decrement timer
209                   //If the timer is < 1, remove the process
210                     if ((*currentPointer).timer < 1)
211                       {//Start of if
212                        //remove the process
213                          nextPointer = removeCurrent(currentPointer);
214                        //Stitch
215                          (*previousPointer).next = nextPointer; //connect the endpointer to the startnode to complete the loop
216                          currentPointer = previousPointer; //Update the currentPointer to the nextPointer
217                          deduct++; //Update the deduction
218                       }//End of if
219                }//End of for
220                //Update the processCount
221                  processCount -= deduct;
222                //Reset the deductions
223                  deduct = 0;
224              //Print the list
225                printList(currentPointer);
226           }//End of while
227        return endPointerEntry;
228      }//End of scanList
```

### 2.2.3.7. removeCurrent function:

#### 2.2.3.7.1. Description:

*The removeCurrent function takes the pointer of the node to delete, and returns a pointer to the node next to it.*

*It first checks to make sure that the list is not empty. If it is empty, it returns NULL, it if still has one node left, the node will be removed and the curretPointer will refer to NULL. If there are more than 1 nodes in the list, the nextPointer is assigned to the next node of the*

*currentPointer, and when the current node is deleted, the function returns the nextPointer.*

#### *2.2.3.7.2. **Code:***

```
230  //Remove process function
231    struct node* removeCurrent(struct node* currentPointerEntry)
232      {//Start of removeCurrent
233       //Variables
234         struct node *currentPointer;
235         struct node *nextPointer;
236       //Remove the node if the structure is not empty and has more than 1 item
237         if(isEmpty() == false && processCount > 1)
238           {//Start of if
239            //update the pointers
240              nextPointer = currentPointerEntry;
241              nextPointer = (*nextPointer).next; //index
242            //Make the node NULL to remove it
243              currentPointerEntry = NULL;
244              free(currentPointerEntry);
245            //Return a reference to next node so it can stitch
246              return nextPointer;
247           }//End of if
248       //Remove the node and refer current to NULL if structure is not empty and has 1 node left
249         else if(isEmpty() == false && processCount == 1)
250           {//Start of if
251            //Make the node NULL to remove it
252              currentPointerEntry = NULL;
253              free(currentPointerEntry);
254            //Return a reference to next node so it can stitch
255              return currentPointerEntry;
256           }//End of if
257       //If the structure is empty, return NULL
258         else
259           {return NULL;}
260    }//End of removeCurrent
```

## *2.3. Compile Instructions:*

### *2.3.1. Setup:*

*The folder scheduler should contain 3 files (scheduler.c, test1.txt, and makefile). The test1.txt file must contain each process on a unique line, where the pid and timer are separated by one space. There should not be any auxiliary characters outside each line. For example, for a list to contain 3 processes: (pid = 1 / timer = 3), (pid = 2 / timer = 1), and (pid = 3 / timer = 2), the test1.txt file should look like this:*

```
1 3
2 1
3 2
```

### *2.3.2. Compiling:*

*Browse for the directory of the project and type* `make` *inside the terminal.*

### 2.3.3. ***Executing:***

*Execute the program by typing* `./scheduler` *inside the terminal.*

## 2.4. ***Sample Test Run:***

***(NOTE: Additional test conditions are found under section 2.5.3)***

### 2.4.1. ***Txt file content:***

```
1 3
2 1
3 2
```

### 2.4.2. ***Output:***

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ gcc scheduler.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ ./a.out

**************************START************************
**********************SCHEDULER**********************
******************PART I: CREATE LIST*****************
Adding node#1 to list...
Content:
>NODE(pid:1 / timer:3)

Adding node#2 to list...
Content:
>NODE(pid:1 / timer:3)
>NODE(pid:2 / timer:1)

Adding node#3 to list...
Content:
>NODE(pid:1 / timer:3)
>NODE(pid:2 / timer:1)
>NODE(pid:3 / timer:2)

******************PART II: SCAN LIST******************
Original list content...
Content:
>NODE(pid:1 / timer:3)
>NODE(pid:2 / timer:1)
>NODE(pid:3 / timer:2)

Now scanning through list...
running scan#1...
Content:
>NODE(pid:1 / timer:2)
>NODE(pid:3 / timer:1)

running scan#2...
Content:
>NODE(pid:1 / timer:1)

running scan#3...
Content:

**************************END************************
```

### *2.5.1. Development:*

*The program began with understanding customer requirements. This required detail review of the assignment sheet, and confirming the dos and don'ts on the items that are not clear. The behavior of the round-robin was the first concept addressed, as well as the functionality and dynamics of its sub-functions.*

*Once the requirements were clear, development began with pseudo-code on the architecture. At this stage, the code had general comments inside each function block.*

*Next, the actual coding process began using the standard syntax and semantics learned in class. Research was required from the C documentation, the book, and the slides provided in class.*

### *2.5.2. Debugging:*

*To debug the program, the C file was compiled using –g, so that the terminal can run the program in debugging mode. Once inside gdb, several debugging tools were used to diagnose the program. This included breaks, displays, and stepping one line at a time. Another technique used was the exit(0) function to exit the program at certain locations, and printing variable values inside loops, to observe their progression.*

*The 80/20 rule was strong in this assignment, where 2 bugs took longer to resolve then the actual development of the assignment. One of the bugs was towards the end of the list, where it had trouble giving proper results when 1 or 2 nodes were left. The issue was that iterators were not placed inside the right scope location of for and while loops. Also, incorrect pointers were being passed to functions.*

### *2.5.3. Testing:*

*The program went through several tests to assure the quality and reliability of the final product. Some tests were done for compatibility (i.e. running on school computers versus home computers) and others were done for functionality.*

*Three sample test results are shown below. The 1st test has the last node removed first, the 2nd test has the first node removed first, and the 3rd test has 2 intermediate nodes removed first, so that the intermediate stitching can be tested. The results are shown below.*

***Test#01: (end node removes first):***

*2.5.3.1.1.* ***Txt file content:***

```
1 5
2 4
3 3
4 2
5 1
```

*2.5.3.1.2.* ***Output:***

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ gcc scheduler.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ ./a.out

************************START************************
**********************SCHEDULER**********************
******************PART I: CREATE LIST*****************
Adding node#1 to list...
Content:
>NODE(pid:1 / timer:5)

Adding node#2 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:4)

Adding node#3 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:4)
>NODE(pid:3 / timer:3)

Adding node#4 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:4)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)

Adding node#5 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:4)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)
>NODE(pid:5 / timer:1)

******************PART II: SCAN LIST******************
Original list content...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:4)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)
>NODE(pid:5 / timer:1)

Now scanning through list...
running scan#1...
Content:
>NODE(pid:1 / timer:4)
>NODE(pid:2 / timer:3)
>NODE(pid:3 / timer:2)
>NODE(pid:4 / timer:1)

running scan#2...
Content:
>NODE(pid:1 / timer:3)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:1)

running scan#3...
Content:
>NODE(pid:1 / timer:2)
>NODE(pid:2 / timer:1)

running scan#4...
Content:
>NODE(pid:1 / timer:1)

running scan#5...
Content:

************************END************************
```

#### 2.5.3.2. Test#02: (first node removes first):

##### 2.5.3.2.1. Txt file content:

```
1 1
2 2
3 3
4 4
5 5
```

##### 2.5.3.2.2. Output:

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ gcc scheduler.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ ./a.out
************************START************************
**********************SCHEDULER**********************
******************PART I: CREATE LIST*****************
Adding node#1 to list...
Content:
>NODE(pid:1 / timer:1)

Adding node#2 to list...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:2 / timer:2)

Adding node#3 to list...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)

Adding node#4 to list...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:4)

Adding node#5 to list...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:4)
>NODE(pid:5 / timer:5)

******************PART II: SCAN LIST*****************
Original list content...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:4)
>NODE(pid:5 / timer:5)

Now scanning through list...
running scan#1...
Content:
>NODE(pid:2 / timer:1)
>NODE(pid:3 / timer:2)
>NODE(pid:4 / timer:3)
>NODE(pid:5 / timer:4)

running scan#2...
Content:
>NODE(pid:3 / timer:1)
>NODE(pid:4 / timer:2)
>NODE(pid:5 / timer:3)

running scan#3...
Content:
>NODE(pid:4 / timer:1)
>NODE(pid:5 / timer:2)

running scan#4...
Content:
>NODE(pid:5 / timer:1)

running scan#5...
Content:

************************END************************
```

#### 2.5.3.3. Test#03: (multiple removals and inter-list stitching):

##### 2.5.3.3.1. Txt file content:

```
1 5
2 2
3 3
4 2
5 5
```

##### 2.5.3.3.2. Output:

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ gcc scheduler.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/scheduler$ ./a.out

************************START************************
**********************SCHEDULER**********************
******************PART I: CREATE LIST*****************
Adding node#1 to list...
Content:
>NODE(pid:1 / timer:5)

Adding node#2 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:2)

Adding node#3 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)

Adding node#4 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)

Adding node#5 to list...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)
>NODE(pid:5 / timer:5)

******************PART II: SCAN LIST******************
Original list content...
Content:
>NODE(pid:1 / timer:5)
>NODE(pid:2 / timer:2)
>NODE(pid:3 / timer:3)
>NODE(pid:4 / timer:2)
>NODE(pid:5 / timer:5)

Now scanning through list...
running scan#1...
Content:
>NODE(pid:1 / timer:4)
>NODE(pid:2 / timer:1)
>NODE(pid:3 / timer:2)
>NODE(pid:4 / timer:1)
>NODE(pid:5 / timer:4)

running scan#2...
Content:
>NODE(pid:1 / timer:3)
>NODE(pid:3 / timer:1)
>NODE(pid:5 / timer:3)

running scan#3...
Content:
>NODE(pid:1 / timer:2)
>NODE(pid:5 / timer:2)

running scan#4...
Content:
>NODE(pid:1 / timer:1)
>NODE(pid:5 / timer:1)

running scan#5...
Content:

************************END************************
```

**File Manager**
**See provided *manager* folder for files.**

# 3. File Management System:

## 3.1. Summary of code:

*This program is designed to simulate the ls and cp commands of linux. The ls command lists file properties and the cp command makes copies of files.*

*The ls command takes 3 or 4 arguments and displays files and directories of either the current directory, or of a directory provided as a path for an optional 4th argument. If the 4th argument is not provided, the current directory is assumed. Seven file/directory entities are displayed with ls: permission bits, number of links, uid, gid, file size, last modified date, and the filename. Name, Size, and date can all be sorted by using –l, -s, and –t respectively.*

*The cp command takes 4 arguments only and makes a copy of the source file to the destination file. If the source file name is the same as the destination file name, it will create a file with (new) next to it. If a path is provided for the 4th argument, it will save the new file to that destination. Otherwise, it will create the file at the current location if only a name is provided.*

## 3.2. Detail Description:

### 3.2.1. Header files (libraries & prototypes):

#### 3.2.1.1. Description:

*The header files include all of the libraries required to properly run the methods used to run the program. This includes the <string.h> library for using strcmp and strcpy(useful for sorting file names), and the sys/stat.h library for reading files properties. The header also includes the prototypes of the functions so that they can run properly during execution.*

#### 3.2.1.2. Code:

```
1  //Libraries
2      #include <stdio.h>
3      #include <string.h>
4      #include <stdlib.h>
5      #include <sys/types.h>
6      #include <sys/stat.h>
7      #include <unistd.h>
8      #include <fcntl.h>
9      #include <dirent.h>
10     #include <pwd.h>
11     #include <grp.h>
12
13 //Variables
14     #define BUFFERSIZE 4096
15     #define COPYMODE 0644
16
17 //Prototypes
18     void do_ls(char dirname[], int sortEntry);
19     void dostat(char *);
20     void show_file_info(char *, struct stat *);
21     void mode_to_letters(int, char[]);
22     char *uid_to_name(uid_t);
23     char *gid_to_name(gid_t);
24     void oops(char *, char *);
25
```

### 3.2.2. Functions:

#### 3.2.2.1. Main function:

##### 3.2.2.1.1. Error checking:

###### 3.2.2.1.1.1. Description:

*The main function begins with a series of error checks that exit the code as soon as errors are detected. This design allows better readability, as opposed to nested if-else statement where the programmer has to scroll all the way to the bottom to find the "else" condition. These checks evaluate the number of arguments and whether the commands are spelled correctly.*

###### 3.2.2.1.1.2. Code:

```
31              //User-input control
32                  //Check that 1st argument calls the program myFS
33                      if( strcmp(argv[0], "./myFS") != 0)
34                          {//Start of if
35                              printf("ERROR! 1st argument needs to be [./myFS].");
36                              exit(0);
37                          }//End of if
38
39                  //Check that there 3 or 4 arguments only
40                      if( argc != 3 && argc != 4)
41                          {//Start of if
42                              printf("ERROR! needs 3 or 4 arguments only.\n");
43                              exit(0);
44                          }//End of if
45
46                  //Check that 2nd argument is cp or ls
47                      if(//Start of if condition
48                          strcmp(argv[1], "cp") != 0 &&
49                          strcmp(argv[1], "ls") != 0
50                        )//End of if condition
51                          {//Start of if
52                              printf("ERROR! 2nd argument needs to be [cp] or [ls] only.\n");
53                              exit(0);
54                          }//End of if
55
56                  //Check that if cp, there are 4 arguments only
57                      if( argc != 4 && strcmp(argv[1], "cp") == 0)
58                          {//Start of if
59                              printf("ERROR! the cp command needs exactly 4 arguments.\n");
60                              exit(0);
61                          }//End of if
62
63                  //If ls, check that 2nd argument is -l, -t, or -s.
64                      if(//Start of condition
65                          (strcmp(argv[1], "ls") == 0) &&
66                          (strcmp(argv[2], "-l") != 0) &&
67                          (strcmp(argv[2], "-t") != 0) &&
68                          (strcmp(argv[2], "-s") != 0)
69                        )//End of condition
70                          {//Start of if
71                              printf("ERROR! the 3rd argument must be [-l], [-t], or [-s].\n");
72                              exit(0);
73                          }//End of if
```

#### 3.2.2.1.2. LS – list files/directories:

##### 3.2.2.1.2.1. Description:

*Next, depending whether the user typed "ls" or "cp", the code will go different directions. If "ls" is typed, the code will further investigate for the type of sorting i.e. "-l", "-s", or "-t", and the path provided. If the 4th argument is not provided, it returns a dot for the path, otherwise it returns the path provided. The function then passes the path and sorting method to the do_ls command.*

##### 3.2.2.1.2.2. Code:

```
75              //LS
76                  if(strcmp(argv[1], "ls") == 0)
77                      {//Start of LS
78                          //Variables
79                              char path [500];
80                              int sortStyle = 0;
81                          //Determine path
82                              //If path is not provided, ls current...
83                                  if (argc == 3)
84                                  {strcpy(path, ".");}
85                              //If path is provided, ls path...
86                                  else if(argc == 4)
87                                  {strcpy(path, argv[3]);}
88                          //Determine sort style
89                              //-l sort by filename
90                                  if(strcmp(argv[2], "-l") == 0)
91                                  {sortStyle = 0;}
92                              //-t sort by time...
93                                  else if(strcmp(argv[2], "-s") == 0)
94                                  {sortStyle = 1;}
95                              //-s sort by size...
96                                  else if(strcmp(argv[2], "-t") == 0)
97                                  {sortStyle = 2;}
98                          //Stuff
99                              do_ls(path, sortStyle);
100                     }//End of LS
```

#### 3.2.2.1.3. CP – copy files:

*If "cp" is typed, the code will first check to see if the source and destination name is the same. If it is the same, it will add (new), next to the new name, so that a copy of the file can still be created. If the source and destination is not the same, then cp will create a copy of the source file to the destination with the provided new name. Before writing to the system, cp needs to run several tests*

*to make sure that read(), open(), and creat() can work properly. It does this by checking to see if the file descriptor is -1, otherwise the system call was a success.*

#### 3.2.2.1.4. ***Code:***

```
102             //CP
103                 else if(strcmp(argv[1], "cp") == 0)
104                     {//Start of cp
105                         //Variables
106                             int in_fd, out_fd, n_chars;
107                             char buf[BUFFERSIZE];
108                             int i, j, k;
109                             int breakFileName = 0;
110                             char fileName[500];
111                             char fileExtension[500];
112                         //If the source and destination file is the same...
113                             if(strcmp(argv[2], argv[3]) == 0)
114                                 {//start of if
115                                     for (i = 0; i < strlen(argv[3]); i++)
116                                         {//Start of for
117                                             //Once it finds the . separator, split name and extension
118                                                 if(argv[3][i] == '.')
119                                                     {breakFileName = 1;}
120                                             //Store the fileName
121                                                 if(breakFileName == 0)
122                                                     {fileName[i] = argv[3][i];}
123                                             //Store the fileExtension
124                                                 else if(breakFileName == 1)
125                                                     {fileExtension[i - strlen(fileName)] = argv[3][i];}
126                                         }//End of for
127                                     //Close the name and extension strings with '\0'
128                                         fileName[i] = '\0';
129                                         fileExtension[i - strlen(fileName)] = '\0';
130                                     //Concatenate the destination file with [new]
131                                         strcat(fileName, "[new]");
132                                         strcat(fileName, fileExtension);
133                                         strcpy(argv[3], fileName);
134                                 }//end of if
135                         //Check that files can be opened
136                             if((in_fd = open(argv[2], O_RDONLY)) == -1)
137                                 {oops("Cannot open ", argv[2]);}
138                         //Check that files can be created
139                             if((out_fd = creat(argv[3], COPYMODE)) == -1)
140                                 {oops( "Cannot creat", argv[3]);}
141                         //Read from source to buffer and transfer to destination
142                             while((n_chars = read(in_fd , buf, BUFFERSIZE)) > 0)
143                                 {//Start of while
144                                     if(write(out_fd, buf, n_chars) != n_chars)
145                                     {oops("Write error to ", argv[3]);}
146                                 }//End of while
147                         //Check for file read
148                             if(n_chars == -1)
149                                 {oops("Read error from ", argv[2]);}
150                         //Check for file close
151                             if(close(in_fd) == -1 || close(out_fd) == -1)
152                                 {oops("Error closing files","");}
153                     }//End of cp
154         }//End of main
```

### 3.2.2.2. Do_ls:

#### 3.2.2.2.1. Description:

*The do_ls function will run through the files/directories inside the provided path using readdir(), and store the size, name, and time of each file/directory into a separate array. An index array is also populated to serve as a master reference when sorting. Next, it looks at the sortStyle that the user provided i.e. -l, -s, or –t, and sorts the arrays from smallest to largest accordingly. The bubble sort algorithm will not only sort the corresponding array, but also sort the master index array. After sorting is complete, the function then prints the values in proper order, by iterating through the sorted index array, and passing the nameArray to the dostat function, since the name serves as the key for printing properties.*

#### 3.2.2.2.2. Code:

```
156  //do_ls
157      void do_ls(char dirname[], int sortEntry)
158          {//Start of do_ls
159              //If path is given, move to directory . file
160                  chdir(dirname);
161                  strcpy(dirname, ".");
162
163              DIR *dir_ptr;
164              struct dirent *direntp;
165              if((dir_ptr = opendir(dirname)) == NULL)
166                  {fprintf(stderr, "cannot open %s\n", dirname);}
167              else
168                  {//Start of else
169                      //Variables
170                          int listingCount = 0;
171                          int i, j, k;
172                          int swapInt;
173                          char swapString[500];
174                          struct stat info;
175                          //Count the array size
176                              while((direntp = readdir(dir_ptr)) != NULL)
177                                  {listingCount++;}
178                          //Close directory pointer
179                              closedir(dir_ptr);
180                      //Declare arrays
181                          char nameArray[listingCount][500];
182                          int indexArray[listingCount];
183                          long int sizeArray[listingCount];
184                          int timeArray[listingCount];
185                      //Build the arrays
186                          //Reset the pointer to the beginning
187                              dir_ptr = opendir(dirname);
188                          //Build the size array
189                              for (i = 0; i < listingCount; i++)
190                                  {//Start of for
191                                      direntp = readdir(dir_ptr);
192                                      stat(direntp->d_name, &info);
193                                      sizeArray[i] = info.st_size;
194                                      timeArray[i] = info.st_mtime;
195                                      strcpy(nameArray[i], direntp->d_name);
196                                      indexArray[i] = i;
197                                  }//End of for
198                          //Close directory pointer
199                              closedir(dir_ptr);
200
201                      //Sort the index array depending on entry
202                          //Sort by name
203                              if (sortEntry == 0)
204                                  {//Start of sort by name
205                                      for (i = 0; i < listingCount - 1; i++)
206                                          {//Start of outer for
207                                              for (j = 0; j < (listingCount - 1 - i); j++)
208                                                  {//Start of inner for
209                                                      if (strcmp(nameArray[j], nameArray[j + 1]) > 0)
210                                                          {//Start of if
211                                                              //Swap timeArray values
212                                                                  strcpy(swapString, nameArray[j]);
213                                                                  strcpy(nameArray[j], nameArray[j + 1]);
214                                                                  strcpy(nameArray[j + 1], swapString);
215                                                          }//End of if
216                                                  }//End of inner for
217                                          }//End of outer for
218                                  }//End of sort of name
```

**code continued on next page…**

```
                    //Sort by size
                        else if (sortEntry == 1)
                            {//Start of sort by size
                                for (i = 0; i < listingCount - 1; i++)
                                    {//Start of outer for
                                        for (j = 0; j < (listingCount - 1 - i); j++)
                                            {//Start of inner for
                                                if (sizeArray[j] > sizeArray[j + 1])
                                                    {//Start of if
                                                        //Swap timeArray values
                                                            swapInt = sizeArray[j];
                                                            sizeArray[j] = sizeArray[j + 1];
                                                            sizeArray[j + 1] = swapInt;
                                                        //Swap indexArray indices
                                                            swapInt = indexArray[j];
                                                            indexArray[j] = indexArray[j + 1];
                                                            indexArray[j + 1] = swapInt;
                                                    }//End of if
                                            }//End of inner for
                                    }//End of outer for
                            }//End of sort by size

                    //Sort by time
                        else if (sortEntry == 2)
                            {//Start of sort by time
                                for (i = 0; i < listingCount - 1; i++)
                                    {//Start of outer for
                                        for (j = 0; j < (listingCount - 1 - i); j++)
                                            {//Start of inner for
                                                if (timeArray[j] > timeArray[j + 1])
                                                    {//Start of if
                                                        //Swap timeArray values
                                                            swapInt = timeArray[j];
                                                            timeArray[j] = timeArray[j + 1];
                                                            timeArray[j + 1] = swapInt;
                                                        //Swap indexArray indices
                                                            swapInt = indexArray[j];
                                                            indexArray[j] = indexArray[j + 1];
                                                            indexArray[j + 1] = swapInt;
                                                    }//End of if
                                            }//End of inner for
                                    }//End of outer for
                            }//End of sort by time

                //Print the sorted array(has to be the nameArray because name is the keyword)
                    int sortedIndex;
                    for(i = 0; i < listingCount; i++)
                        {//Start of for
                            //Update the sortedIndex
                                sortedIndex = indexArray[i];
                            //Print the file info per the sortedIndex
                                if (sortEntry != 0) //If not sorting by name
                                {dostat(nameArray[sortedIndex]);}
                                else {dostat(nameArray[i]);}
                        }//End of for
            }//End of else
        }//End of do_ls
```

### *3.2.2.3. DoStat:*

#### *3.2.2.3.1. Description:*

*Dostat is the gateway between the show_file_info() function and the do_ls() function. It checks to see if the file/directory can properly be extracted for properties, by using the stat(filename, &info) function and checking that it does not return -1. If it passes, it runs through the show_file_info() function.*

#### 3.2.2.3.2. ***Code:***

```
281   //dostat
282       void dostat(char *filename)
283       {//Start of dostat
284           struct stat info;
285           if(stat(filename, &info) == -1)
286               {perror(filename);}
287           else
288               {show_file_info(filename, &info);}
289       }//End of dostat
```

### 3.2.2.4. ***Show_file_info:***

#### 3.2.2.4.1. ***Description:***

*The show_file_info() function prints the properties of the filename. This includes the permission bits, # of links, uid, gid, file size, last modified time, and the filename. The # of links, size, and name are left as is. The uid and gid are converted to strings by using the uid_to_name() and gid_to_name() functions. The permission bits are converted to legible symbols using the mode_to_letters function by converting the mode bits into the 9 letters they correspond to. Finally, the mtime is converted to a readable format. It uses the ctime function to print out the 12 characters starting at offset 4.*

#### 3.2.2.4.2. ***Code:***

```
291   //Show the info of the provided file
292       void show_file_info(char *filename, struct stat * info_p)
293           {//Start of show_file_info
294               char *uid_to_name(), *ctime(), *gid_to_name(), *filemode();
295               void mode_to_letters();
296               char modestr[11];
297               mode_to_letters(info_p->st_mode, modestr);
298               printf("%s", modestr);
299               printf("%4d ", (int)info_p->st_nlink);
300               printf("%-8s ", uid_to_name(info_p->st_uid));
301               printf("%-8s ", gid_to_name(info_p->st_gid));
302               printf("%8ld ", (long)info_p->st_size);
303               printf("%.12s ", 4 + ctime(&info_p->st_mtime));
304               printf("%s\n", filename);
305           }//End of show_file_info
```

### 3.2.2.5. Mode_to_Letters:

#### 3.2.2.5.1. Description:

*As discussed in the previous section, the mode_to_letters is used for converting the permission bits to readable r, w, and x symbols.*

#### 3.2.2.5.2. Code:

```
307  //mode_to_letters
308      void mode_to_letters(int mode, char str[])
309          {//Start of mode_to_letters
310              strcpy(str, "----------");
311              if (S_ISDIR(mode)) str[0] = 'd';
312              if (S_ISCHR(mode)) str[0] = 'c';
313              if (S_ISBLK(mode)) str[0] = 'b';
314              if (mode & S_IRUSR) str[1] = 'r';
315              if (mode & S_IWUSR) str[2] = 'w';
316              if (mode & S_IXUSR) str[3] = 'x';
317              if (mode & S_IRGRP) str[4] = 'r';
318              if (mode & S_IWGRP) str[5] = 'w';
319              if (mode & S_IXGRP) str[6] = 'x';
320              if (mode & S_IROTH) str[7] = 'r';
321              if (mode & S_IWGRP) str[8] = 'w';
322              if (mode & S_IXOTH) str[9] = 'x';
323          }//End of mode_to_letters
```

### 3.2.2.6. Uid_to_name / Gid_to_name:

#### 3.2.2.6.1. Description:

*The uid_to_Name and gid_to_name functions are used to convert the uid(user id#) and guid(group id#) numbers into readable strings. They both take advantage of the getpwuid() and getgrgid() functions stored inside the header files.*

#### 3.2.2.6.2. Code:

```
325  //uid_to_name
326      char * uid_to_name(uid_t uid)
327          {//Start of uid_to_name
328              struct passwd * getpwuid(), *pw_ptr;
329              static char numstr[10];
330              if((pw_ptr = getpwuid(uid)) == NULL)
331                  {//Start of if
332                      sprintf(numstr, "%d", uid);
333                      return numstr;
334                  }//End of if
335              else
336                  {return pw_ptr->pw_name;}
337          }//End of uid_to_name
338
339  //gid_to_name
340      char *gid_to_name(gid_t gid)
341          {//Start of gid_to_name
342              struct group * getgrgid(), *grp_ptr;
343              static char numstr[10];
344              if((grp_ptr = getgrgid(gid)) == NULL)
345                  {//Start of if
346                      sprintf(numstr, "%d", gid);
347                      return numstr;
348                  }//End of if
349              else
350                  {return grp_ptr->gr_name;}
351          }//End of gid_to_name
```

#### 3.2.2.7. Oops:

##### 3.2.2.7.1. Description:

*The oops function is used to organize error messages throughout the code, similar to the catch exception variable in Java and C#. The program exits immediately after printing the error message.*

##### 3.2.2.7.2. Code:

```
353  //Oops function
354      void oops(char *s1, char *s2)
355          {//Start of oops
356              fprintf(stderr,"ERROR! %s ", s1);
357              perror(s2);
358              exit(1);
359          }//End of oops
```

### 3.3. Compile Instructions:

#### 3.3.1. Compiling:

*Browse for the directory of the project and type* `make` *inside the terminal.*

#### 3.3.2. Executing:

##### 3.3.2.1. To run LS:

```
./myFS ls -l [with optional path]
./myFS ls -s [with optional path]
./myFS ls -t [with optional path]
```

##### 3.3.2.2. To run CP:

```
./myFS cp sourceName destinationName [with optional
path]
```

### 3.4. Sample Test Run:

***(NOTE: Additional tests are shown under section 3.5.3)***

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ make
gcc -o myFS myFS.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls
makefile  myFS  myFS.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS ls -l
drwxrwxrwx   2 dizad    dizad        4096 Nov 25 20:26 .
drwxrwxrwx   4 dizad    dizad        4096 Nov 25 17:13 ..
-rw-rw-rw-   1 dizad    dizad          32 Nov 25 17:14 makefile
-rwxrwxrwx   1 dizad    dizad       18392 Nov 25 20:26 myFS
-rw-rw-rw-   1 dizad    dizad       11238 Nov 25 19:30 myFS.c
```

CONTROL

## 3.5. Experiences in debugging and testing:

### 3.5.1. Development:

*The development started with gathering assignment requirements, writing pseudo-code, completing the code, testing for functionality, and releasing.*

### 3.5.2. Debugging:

*One of the main challenges with this program was to understand the built-in functions used for managing files in Linux. There are many reserved words that can easily be confused for variable names in file management, and understanding how these tools work inside and out required a lot of trial and error, research, and reviewing slide decks#07 and 08.*

### 3.5.3. Testing:

*The following tests will show the outputs for both the ls and cp commands. The ls section will show test outputs for sorting and the optional path argument. The cp section will show test outputs for multiple combination of source and destination file names and paths.*

#### 3.5.3.1. LS:

##### 3.5.3.1.1. Sorting Test(s):

###### 3.5.3.1.1.1. Test#01: *(sorting name: ls –l)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ make
gcc -o myFS myFS.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls -l
total 48
-rw-rw-r-- 1 dizad dizad   112 Nov 25 19:54 a.txt
-rw-rw-r-- 1 dizad dizad   742 Nov 25 19:54 b.txt
-rw-rw-r-- 1 dizad dizad   260 Nov 25 19:54 c.txt
-rw-rw-r-- 1 dizad dizad    32 Nov 25 17:14 makefile
-rwxrwxr-x 1 dizad dizad 18392 Nov 25 19:54 myFS
-rw-rw-r-- 1 dizad dizad 11238 Nov 25 19:30 myFS.c
-rw-rw-r-- 1 dizad dizad     0 Nov 25 19:53 testDirectory
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS ls -l
drwxrwxrwx   2 dizad    dizad        4096 Nov 25 19:54 .
drwxrwxrwx   4 dizad    dizad        4096 Nov 25 17:13 ..
-rw-rw-rw-   1 dizad    dizad         112 Nov 25 19:54 a.txt
-rw-rw-rw-   1 dizad    dizad         742 Nov 25 19:54 b.txt
-rw-rw-rw-   1 dizad    dizad         260 Nov 25 19:54 c.txt
-rw-rw-rw-   1 dizad    dizad          32 Nov 25 17:14 makefile
-rwxrwxrwx   1 dizad    dizad       18392 Nov 25 19:54 myFS
-rw-rw-rw-   1 dizad    dizad       11238 Nov 25 19:30 myFS.c
-rw-rw-rw-   1 dizad    dizad           0 Nov 25 19:53 testDirectory
```

CONTROL

#### 3.5.3.1.1.2. ***Test#02:*** *(sorting size: ls –s)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS ls -s
-rw-rw-rw-   1 dizad    dizad          0 Nov 25 19:53 testDirectory
-rw-rw-rw-   1 dizad    dizad         32 Nov 25 17:14 makefile
-rw-rw-rw-   1 dizad    dizad        112 Nov 25 19:54 a.txt
-rw-rw-rw-   1 dizad    dizad        260 Nov 25 19:54 c.txt
-rw-rw-rw-   1 dizad    dizad        742 Nov 25 19:54 b.txt
drwxrwxrwx   4 dizad    dizad       4096 Nov 25 17:13 ..
drwxrwxrwx   2 dizad    dizad       4096 Nov 25 19:54 .
-rw-rw-rw-   1 dizad    dizad      11238 Nov 25 19:30 myFS.c
-rwxrwxrwx   1 dizad    dizad      18392 Nov 25 19:54 myFS
```

#### 3.5.3.1.1.3. ***Test#03:*** *(sorting time: ls –t)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS ls -t
drwxrwxrwx   4 dizad    dizad       4096 Nov 25 17:13 ..
-rw-rw-rw-   1 dizad    dizad         32 Nov 25 17:14 makefile
-rw-rw-rw-   1 dizad    dizad      11238 Nov 25 19:30 myFS.c
-rw-rw-rw-   1 dizad    dizad          0 Nov 25 19:53 testDirectory
-rw-rw-rw-   1 dizad    dizad        742 Nov 25 19:54 b.txt
-rw-rw-rw-   1 dizad    dizad        112 Nov 25 19:54 a.txt
-rw-rw-rw-   1 dizad    dizad        260 Nov 25 19:54 c.txt
drwxrwxrwx   2 dizad    dizad       4096 Nov 25 19:54 .
-rwxrwxrwx   1 dizad    dizad      18392 Nov 25 19:54 myFS
```

### 3.5.3.1.2. ***Path Test(s):***

#### 3.5.3.1.2.1. ***Test#04:*** *(ls –l path)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ make
gcc -o myFS myFS.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls -l testDir01/testDir02
total 12
-rw-rw-r-- 1 dizad dizad 112 Nov 25 19:54 a1.txt
-rw-rw-r-- 1 dizad dizad 742 Nov 25 19:54 b1.txt
-rw-rw-r-- 1 dizad dizad 260 Nov 25 19:54 c1.txt
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS ls -l testDir01/testDir02
drwxrwxrwx   2 dizad    dizad        4096 Nov 25 20:17 .
drwxrwxrwx   3 dizad    dizad        4096 Nov 25 20:17 ..
-rw-rw-rw-   1 dizad    dizad         112 Nov 25 19:54 a1.txt
-rw-rw-rw-   1 dizad    dizad         742 Nov 25 19:54 b1.txt
-rw-rw-rw-   1 dizad    dizad         260 Nov 25 19:54 c1.txt
```

**CONTROL**

#### 3.5.3.2. ***CP:***

##### 3.5.3.2.1. ***Test#05:*** *(cp source destination)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ make
gcc -o myFS myFS.c
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls
makefile  myFS  myFS.c  testDir01  testFile01.txt
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS cp testFile01.txt testFile02.txt
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls
makefile  myFS  myFS.c  testDir01  testFile01.txt  testFile02.txt
```

##### 3.5.3.2.2. ***Test#06:*** *(cp source source)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS cp testFile01.txt testFile01.txt
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls
makefile  myFS  myFS.c  testDir01  testFile01[new].txt  testFile01.txt  testFile02.txt
```

##### 3.5.3.2.3. ***Test#07:*** *(cp source destinationPath)*

```
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ./myFS cp testFile01.txt testDir01/testDir02/testFile02.txt
dizad@dizad-HP-EliteBook-8560p:~/Desktop/cis340/project/manager/submit$ ls testDir01/testDir02
testFile02.txt
```