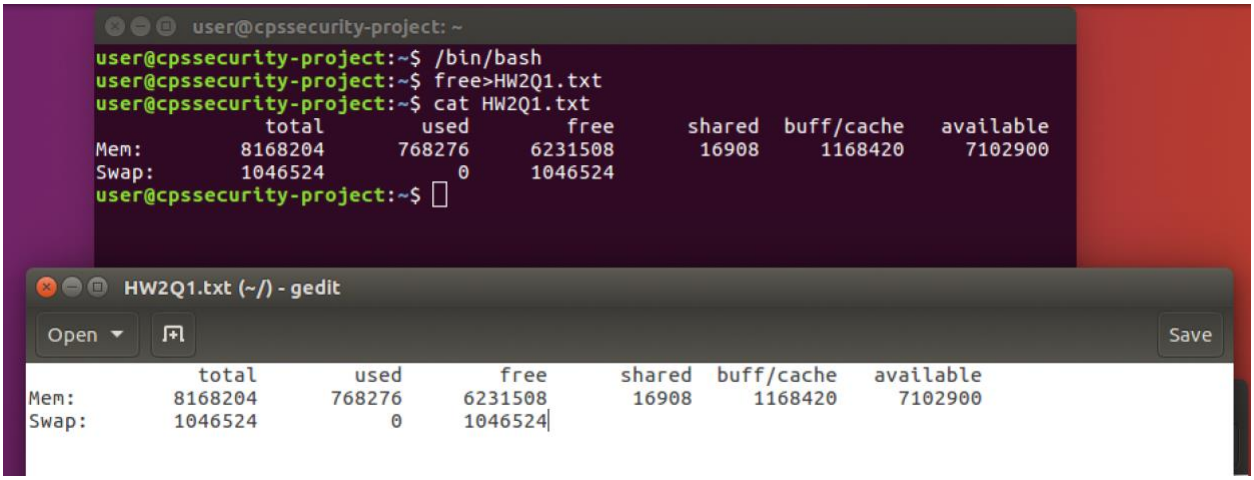
CS 446 Homework 2

Chapter 3

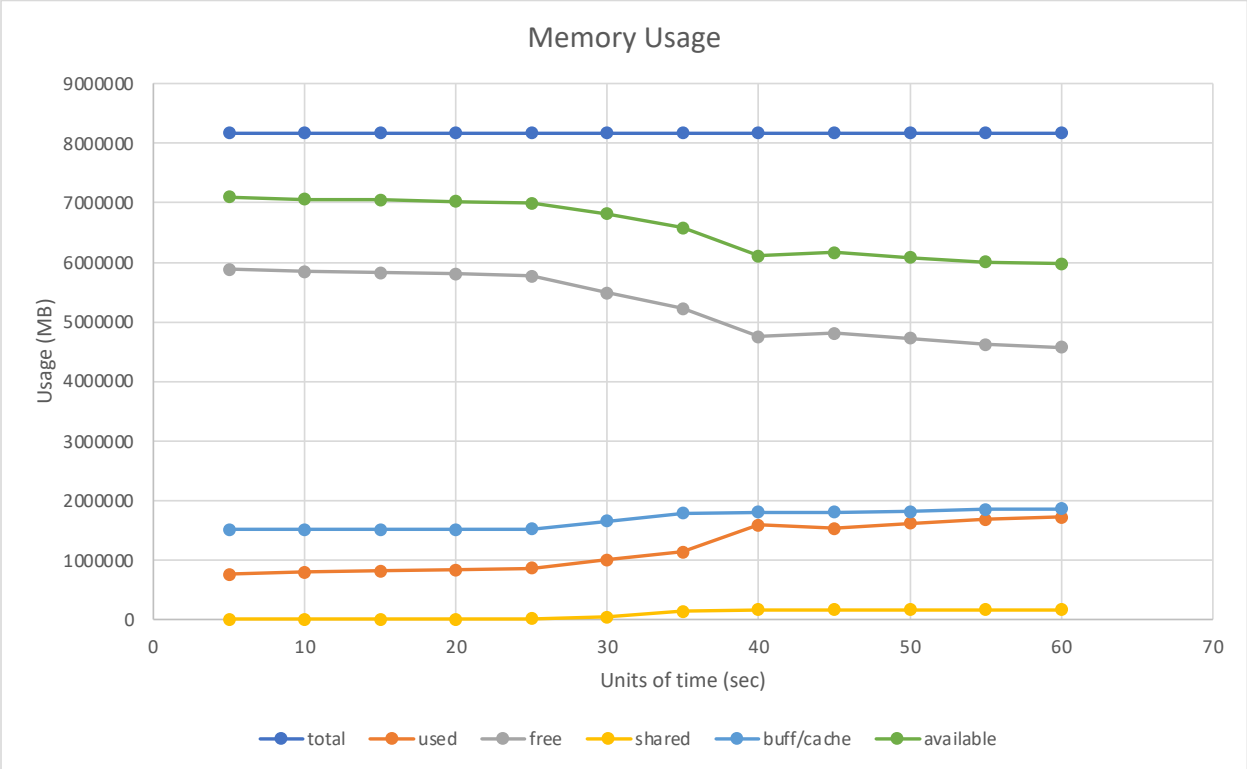
1. [Linux Operating System] free is a command that displays used and available memory in your system. Read man page of free command. Run the command free –o several times, running other programs in between, and store the results in a file. Draw a graph as follows: X-axis: MB-used; for the Y-axis, use (i) Memory Used per unit time; (ii) (Memory Used – Memory Buffered – Memory Cached) per unit time; and (3) Swap Used per unit time. Explain the behavior of this graph with respect to memory utilization in the presence of running various applications.

Answer:



	Time (sec)	total	used	free	shared	buff/cache	available
Mem:	5	8168204	763984	5883056	16952	1521164	7101804
Mem:	10	8168204	803228	5843220	17352	1521756	7062060
Mem:	15	8168204	818780	5827348	17372	1522076	7046336
Mem:	20	8168204	839048	5807012	17372	1522144	7026036
Mem:	25	8168204	871452	5772184	19112	1524568	6991836
Mem:	30	8168204	1014432	5489884	48892	1663888	6817280
Mem:	35	8168204	1146116	5229216	149604	1792872	6583672
Mem:	40	8168204	1598040	4756548	170188	1813616	6107060
Mem:	45	8168204	1541084	4813244	170184	1813876	6163852
Mem:	50	8168204	1622664	4727468	170248	1818072	6081560
Mem:	55	8168204	1691480	4619892	171404	1856832	6010348
Mem:	60	8168204	1726908	4578532	171936	1862764	5973900

	Time (sec)	total	used	free	shared	buff/cache	available
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			
Swap:		1046524	0	1046524			



2. [Any System] Plot a histogram and calculate the mean and median of the sizes of the executable binary files on a computer to which you have access. On a Windows system, look at all .exe and .dll files; on a UNIX system look at all executable files in */bin*, */usr/bin*, and */local/bin* that are not scripts (or use the file utility to find all executables). Determine the optimal page size for this computer just considering the code (not data). Consider internal fragmentation and page table size, making some reasonable assumption about the size of a page table entry. Assume that all programs are equally likely to be run and thus should be weighted equally.

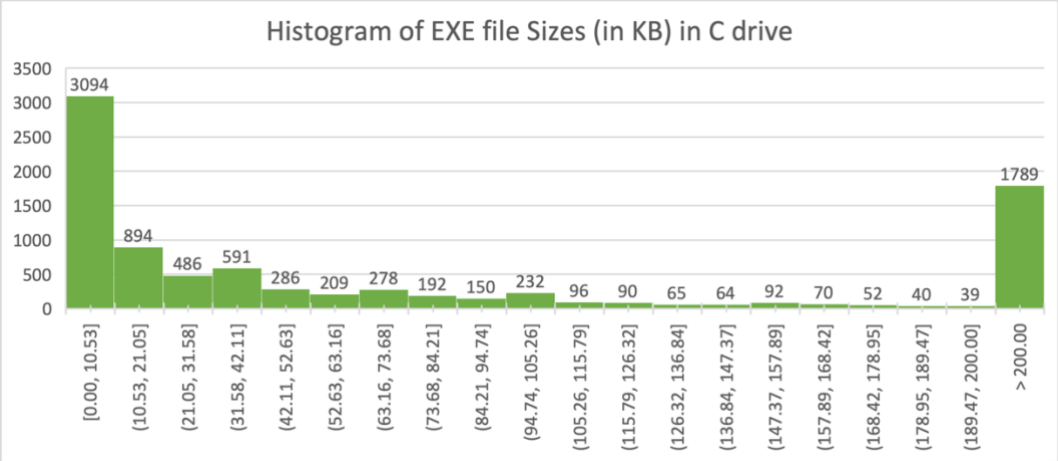
Answer:

I have below command to fetch the size of all the executable files (.exe and .dll) in C drive:

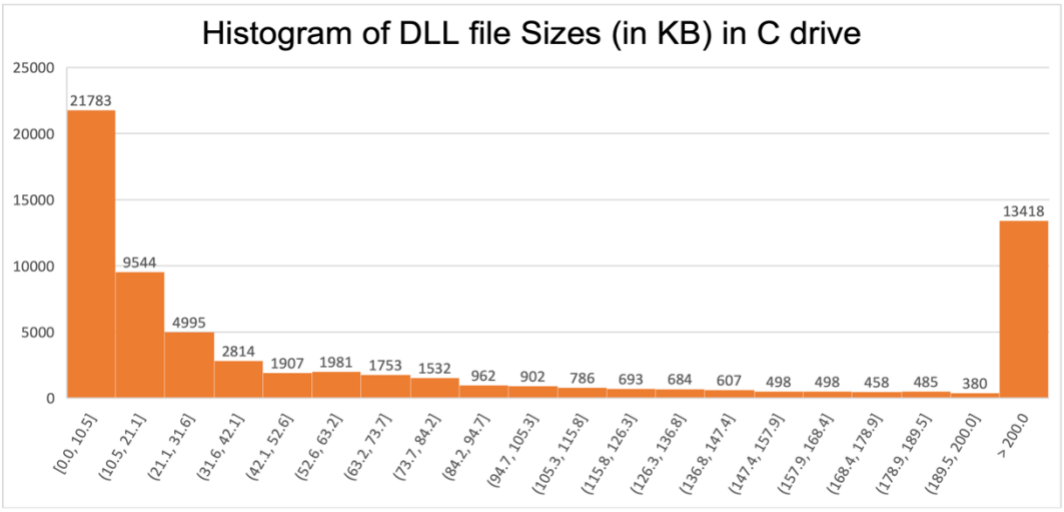
EXE: *dir /s /n *.exe | findstr /v .exe.>/Users/mdtamjidh/c_exe.txt*
DLL: dir /s /n *.dll | findstr /v .dll.>/Users/mdtamjidh/c_dll.txt

The mean and median of the sizes of the executable binary files are also given in the attached c_exe.xlsx file (worksheet exe_2 and dll_2).

For EXE,
the mean is **804.259 KB** and median is **29.56934 KB**
Optimal page size
= (2 x Process size x Page table entry size)$^{1/2}$
= (2 x 804.259 KB x 8 bytes)$^{1/2}$
= (2 x (804.259 x 2^10) bytes x 2^8 bytes)$^{1/2}$
= 28.359 x $2^{9.5}$ bytes
Thus, Optimal page size = 28.359 x $2^{9.5}$ bytes



For DLL,
the mean is **446.8835 KB** and median is **25.09668 KB**
Optimal page size
= (2 x Process size x Page table entry size)$^{1/2}$
= (2 x **446.8835** KB x 8 bytes)$^{1/2}$
= (2 x (**446.8835** x 2^10) bytes x 2^8 bytes)$^{1/2}$
= 21.139 x $2^{9.5}$ bytes
Thus, Optimal page size = 21.139 x $2^{9.5}$ bytes

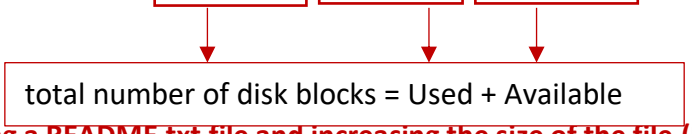


Chapter 2
3. [Linux Operating System] df is a command that displays the amount of disk space available on the file system containing each file name argument. Read man page of df command. Run the command df to find out how many disk blocks are available and how many are in use. Does the sum of these equals the total number of disk blocks on the disk? If not, explain why there is a difference. Next run the command df –i to find out how many i-nodes are available and in use. Now create a new file with just a few characters in it, and again run df and df –i commands. Explain the effect of creating this new file. Now increase the size of this new file by entering a large number (> 5000) of characters, and again run df and df –i commands. Explain the effect of increasing the size of the new file.

Answer:

```
user@cpssecurity-project:~$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             4053348        0   4053348   0% /dev
tmpfs             816824     9336    807488   2% /run
/dev/sda1       19478204 15396544   3069180  84% /
tmpfs            4084100      280   4083820   1% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
```

```
user@cpssecurity-project:~$ df -i
Filesystem      Inodes  IUsed   IFree IUse% Mounted on
udev           1013337    415 1012922    1% /dev
tmpfs          1021025    603 1020422    1% /run
/dev/sda1      1245184 532923  712261   43% /
tmpfs          1021025     10 1021015    1% /dev/shm
tmpfs          1021025      5 1021020    1% /run/lock
tmpfs          1021025     17 1021008    1% /sys/fs/cgroup
tmpfs          1021025     39 1020986    1% /run/user/1000
```



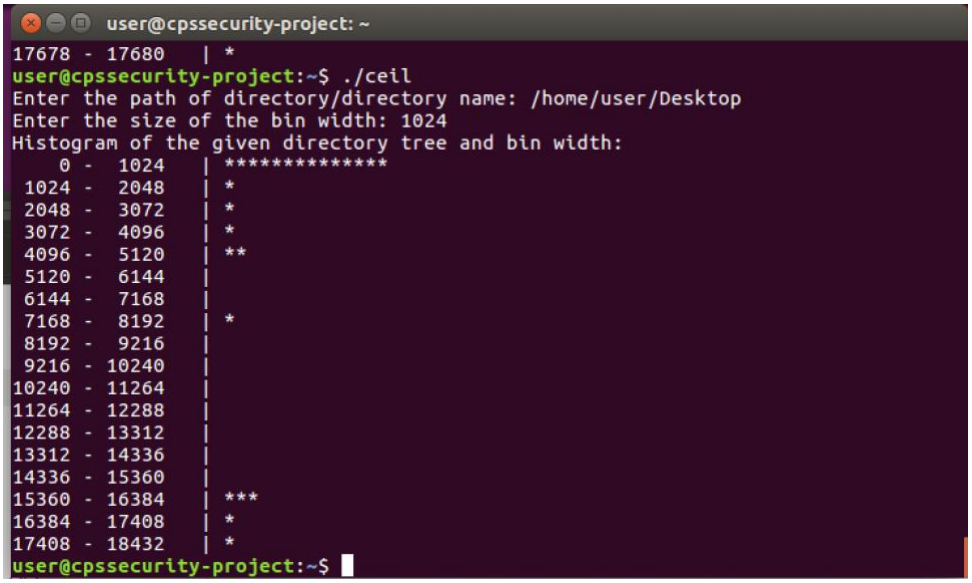
After creating a README.txt file and increasing the size of the file (5000M), we can see the changes in the values

```
user@cpssecurity-project:~$ df -i
Filesystem      Inodes  IUsed   IFree IUse% Mounted on
udev           1013337    415 1012922    1% /dev
tmpfs          1021025    601 1020424    1% /run
/dev/sda1      1245184 532936  712248   43% /
tmpfs          1021025     10 1021015    1% /dev/shm
tmpfs          1021025      5 1021020    1% /run/lock
tmpfs          1021025     17 1021008    1% /sys/fs/cgroup
tmpfs          1021025     37 1020988    1% /run/user/1000
```

```
user@cpssecurity-project:~$ truncate -s 5000M README.txt
user@cpssecurity-project:~$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             4053348        0   4053348   0% /dev
tmpfs             816824     9332    807492   2% /run
/dev/sda1       19478204 15397140   3068584  84% /
tmpfs            4084100      280   4083820   1% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
tmpfs            4084100        0   4084100   0% /sys/fs/cgroup
tmpfs             816824       80    816744   1% /run/user/1000
```

4. [Programming Problem] Write a program that starts at a given directory and descends the file tree from that point, recording the sizes of all the files it finds. When the traversal is complete, the program should print a histogram of the file sizes using a bin width specified as a parameter into the program (e.g., with 1024, file sizes of 0 to 1023 to in one bin, 1024 to 2047 go in the next, etc.

Answer:



Code:

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <math.h>
#include <sys/stat.h>
```

```
#include <unistd.h>

// define a structure which holds
// stat structure pointer and next pointer
typedef struct fileTreeNode
{
   struct stat *stat_ptr;
   struct fileTreeNode *next;
} StatTreeNode;

// define another structure to hold the list of
// StatTreeNode
typedef struct stat_list
{
   StatTreeNode *start_ptr;
} StatFileList;

// declare a function, that takes the StatFileList initial node
// pointer and returns the maximum file size
off_t getMax_FileSize(StatTreeNode *statTreeNode);

// declare the function that takes a pointer of StatTreeNode and
// string containing the name of the directory and returns the
// int value
int decend_Tree_Dir(StatTreeNode *statTreeNode, const char *pathname);

// declare a function, that takes the StatTreeNode and update the bin array
void upDateBin(StatTreeNode *statTreeNode, int binArray[], int binWidth);

// declare a function to print the histogram
void printHistogram(int binArray[], int num_bins, int bin_width);

// define the main function
int main(int argc, char *argv[])
{
   // declare the stat structure variable
   struct stat stat_struct;

   // declare the StatFileList variable
   StatFileList filesList;

   // declare a pointer to the StatTreeNode
   StatTreeNode *statTreeNode;
   char directoryName[50];

   // declare the variable to hold the bin size
```

```
int bin_width = 0, num_bins;

// declare a variable to hold the return value of the function
int func_ReturnValue = 0;

// declare a off_t value
off_t max_Size = 0;

// prompt the user for the name of the directory
// directory path name
printf("Enter the path of directory/directory name: ");
scanf("%s", directoryName);

// check the condition whether an object can be created
// with the given directoryName
if (stat(directoryName, &stat_struct) == -1)
{
    perror("The given input is invalid");
    exit(EXIT_FAILURE);
}

// check the condition that whether the given
// directoryName is a directory
if (!S_ISDIR(stat_struct.st_mode))
{
    fputs("The given input path/directory name is not a directory\n", stderr);
    exit(EXIT_FAILURE);
}

// prompt the user for the bin width
printf("Enter the size of the bin width: ");
scanf("%d", &bin_width);

// initialize the first node of the list with
// initial size of StartTreeNode
filesList.start_ptr = malloc(sizeof(StatTreeNode));

// call the function decend_Tree_Dir() by passing
// initial fileList pointer and name of the directory
func_ReturnValue = decend_Tree_Dir(filesList.start_ptr, directoryName);

// check the condition whether the return value is not zero
// if not zero, exit from the program
if (func_ReturnValue != 0)
{
    exit(EXIT_FAILURE);
```

```
    }

    // set the filesList start_prt to statTreeNode
    statTreeNode = filesList.start_ptr;

    // call the function getMax_FileSize() by passing
    // the statTreeNode and store the return value in
    // max_Size
    max_Size = getMax_FileSize(statTreeNode);

    // re-set the statTreeNode
    statTreeNode = filesList.start_ptr;

    // find the number of bins that are required
    num_bins = (int)ceil(max_Size / bin_width) + 1;

    // declare an array of bins
    int bin_array[num_bins];

    // initialize each bin to zero value
    int i = 0;
    for (i = 0; i < num_bins; i++)
    {
        bin_array[i] = 0;
    }

    // call the function uupDateBin, to update the aray
    // bins
    upDateBin(statTreeNode, bin_array, bin_width);

    // re-set the statTreeNode
    statTreeNode = filesList.start_ptr;

    // call the function printHistogram() to diaply the histogram
    printHistogram(bin_array, num_bins, bin_width);

    return 0;
}

/**
* printHistogram() function, this accepts an array of int, and two int
* variables.
* This function is used to display the histogram of the given bin width.
*/
void printHistogram(int bin_array[], int num_bins, int bin_width)
{
```

```
    puts("Histogram of the given directory tree and bin width:");
    for (int i = 0; i < num_bins; i++)
    {
        printf("%5d - %5d\t| ", bin_width * i, bin_width * (i + 1));
        for (int j = 0; j < bin_array[i]; j++)
        {
            printf("%s", "*");
        }
        printf("\n");
    }
}

/**
* getMax_FileSize() function that accepts a StatTreeNode pointer
* and returns the off_t.
* This function, loops through each node in the StatTreeNode(file size),
* finds the maximum size of the file in the list and returns the maximum
* size.
*/
off_t getMax_FileSize(StatTreeNode *statTreeNode)
{
    // declare a variable of type off_t
    off_t largeFileSize = 0;

    // define a variable pointer of StatTreeNode, which is
    // initialized with the statTreeNode
    StatTreeNode *current = statTreeNode;

    // loop through each node
    while (current->next != NULL && current->stat_ptr != NULL)
    {
        // condition to check the largest file size
        if (current->stat_ptr->st_size > largeFileSize)
        {
            // if the current's stat size is larger than the
            // largeFileSize then set the current's stat size
            // to largeFileSize
            largeFileSize = current->stat_ptr->st_size;
        }

        // move pointer to the next node
        current = current->next;
    }

    // return the largeFileSize
    return largeFileSize;
```

```
}

/**
* upDateBin() function that accepts a StatTreeNode pointer, an int array
* and an int value
*/
void upDateBin(StatTreeNode *statTreeNode, int binArray[], int binWidth)
{
   // define a variable pointer of StatTreeNode, which is
   // initialized with the statTreeNode
   StatTreeNode *current = statTreeNode;

   // declare a variable to hold the index
   off_t index = 0;
   // loop through each node
   while (current->next != NULL && current->stat_ptr != NULL)
   {
      // set the index value
      index = current->stat_ptr->st_size / binWidth;

      // increment the binArray value at the index
      binArray[index]++;

      // move pointer to the next node
      current = current->next;
   }
}

/**
* decend_Tree_Dir() recursive function accepts a StatTreeNode and a string
* holding path name and returns an int value.
* This function goes through each directory and file and sets the
* size of the file to the statTreeNode node and returns an int value
* if there is any error.
*/
int decend_Tree_Dir(StatTreeNode *statTreeNode, const char *directory_PathName)
{
   // declare the variable of type DIR
   DIR *directoryInput;

   // declare an int variable
   int dir_fd;

   // declare the variable to hold the file status
   int file_status = -1;
```

```
// declare a variable to hold the error status
int err_status = -1;

// declare the dirent structure pointer
struct dirent *direntPtr;

// declare the stat structure pointer
struct stat *stat_buffer;

// Check whether the given directory name is able to open or not
if ((directoryInput = opendir(directory_PathName)) == NULL)
{
    fprintf(stderr, "Unable to open \"%s\" directory.\n", directory_PathName);
    return errno;
}

// Check whether the given directory is able to open the file descriptor
if ((dir_fd = dirfd(directoryInput)) == -1)
{
    fprintf(stderr, "Could not able to obtain directory file descriptor "
                    "of: %s\n", directory_PathName);
    return errno;
}

// if able to open up the directory then loop through and get the size of the file
while ((direntPtr = readdir(directoryInput)) != NULL)
{
    // if the directory name contains "." or ".." then skip the directories
    if (strcmp(direntPtr->d_name, ".") == 0 || strcmp(direntPtr->d_name, "..") == 0)
    {
        continue;
    }

    // initialize the stat_buffer
    stat_buffer = malloc(sizeof(struct stat));

    // get the status of the file
    file_status = fstatat(dir_fd, direntPtr->d_name, stat_buffer, 0);

    // if the file_status is -1, then display an error message and return
    // error value
    if (file_status == -1)
    {
        // get the error number
        err_status = errno;
```

```
            // free the stat_buffer
            free(stat_buffer);

            // display the error message
            fprintf(stderr, "Unable to get the file status related to "
                    "the file \"%s\" descriptor. \n", direntPtr->d_name);

            // return the error status
            return err_status;
        }

        // use switch structure to invoke the related case
        switch (stat_buffer->st_mode & S_IFMT)
        {
            // if the file is a regular file, then set the size of the
            // to the
        case S_IFREG:
            statTreeNode->next = malloc(sizeof(StatTreeNode));
            statTreeNode->stat_ptr = stat_buffer;
            statTreeNode = statTreeNode->next;
            statTreeNode->next = NULL;
            continue;

            // if the file is a directory with in a directory, the
            // go through each sub-directory for the files by calling
            // current function(recursive)
        case S_IFDIR:
        {
            // add the path
            char *sub_path = malloc(strlen(directory_PathName) + strlen(direntPtr->d_name)
+ 2);

            // display the subpath
            sprintf(sub_path, "%s/%s", directory_PathName, direntPtr->d_name);

            // get the error status returned by the function decend_Tree_Dir()
            err_status = decend_Tree_Dir(statTreeNode, sub_path);

            //
            if (err_status != 0)
            {
                fprintf(stderr, "Unable to open the sub_directory: %s\n", direntPtr->d_name);
                return err_status;
            }

            // free the pointer
```

```
            free(sub_path);
            break;
        }
        }

        // free the stat_buffer
        free(stat_buffer);
    }

    // close the directory
    if (closedir(directoryInput) == -1)
    {
        fprintf(stderr, "Could not close the \"%s\" directory.\n", directory_PathName);
        return errno;
    }
    return 0;
}
```