Orange Wave

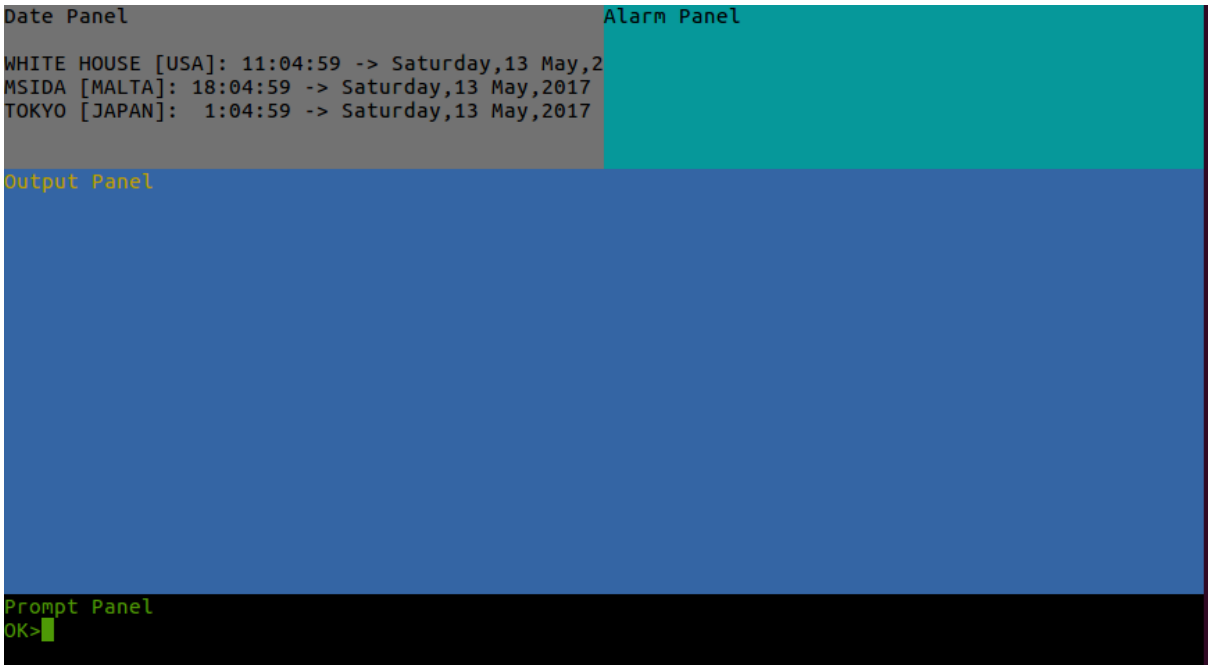
CPS 1012 – Operating System and Systems Programming 1

Franklyn Sciberras – ID: 441498(m)

CONTENTS

TASK 1 – PROMPT AND OUTPUT PANELS

Task 1 consisted of the construction of the necessary infrastructure in order for a shell to operate. This would consist of designing and implementing the necessary in built functions, as well as the recognition of external commands, whilst having the appropriate panels in which to operate and apply some sort of graphical user interface. One panel was initialised for the user to input his commands, and another panel for the following outputs to be displayed. Even more three additional panels, one to display the current time for three different cities, one to display the alarms, and another one to display the colour code of the alarm, were initialised (will be discussed in another chapter). A main panel which would act as a background screen and hold all the other panels was also initialised. The following was the achieved shell infrastructure:



General Infrastructure

INITIALISER ()

In order to achieve the previous output, a pointer for each separate panel was declared globally (in order to be accessible by all the methods) and a function called initialiser() was set up. Such method consisted of the initialisation of the screen in the Main panel. This would open up and take the same size of the current terminal invoking it. Then its maximum x and y co ordinates were retrieved using the getmaxyx() method and stored in the global variables x and y. Such values were then used in order to calculate various sizes (which were stored in separate variables) which then would be used by different panels in order to achieve the previous result. Methods echo() and start_color() were called in order to enable the user to see what he is typing and the default colour scheme would be initialised, respectively. Init_color() was called such that the default shades of black and white would be altered, and then the new valued colors were used in order to create colour pairs, each having a different id, using the function call init_pair(). Four different colour pairs were initialised in order to differentiate between one panel and another, with each colour pair being attributed to a specific panel using the function wbkgd(). Note that the actual creation of the panels occurred when the previously mentioned panel pointers were attributed the returning value of the newwin() function, which would actually create a new window at the specific co-ordinates having a specific size (using the previously mentioned variables). The appropriate panel titles were written to the panels accordingly and the wrefresh() function call was invoked in order to display the whole initialiser() method on screen.

```
void initialiser(){
    if((Main = initscr()) == NULL){
        fprintf(stderr, "Error initialising ncurses.\n");
        exit(EXIT_FAILURE);
    };


    getmaxyx(Main,y,x);
    sizeSmallPanels_x = x/2;
    sizeSmallPanels_y = y/4;
    sizeLargePanels_x = x;
    sizeLargePanels_y = (13*y)/20;
    smallestPanels_y = y/10;
    echo();
    start_color();
    init_color(COLOR_WHITE,450,450,450);
    init_color(COLOR_BLACK,0,0,0);
    init_pair(1,COLOR_GREEN,COLOR_BLACK);
    init_pair(2,COLOR_BLACK,COLOR_WHITE);
    init_pair(3,COLOR_BLACK,COLOR_CYAN);
    init_pair(4,COLOR_YELLOW,COLOR_BLUE);
    Date_Panel = newwin(sizeSmallPanels_y,sizeSmallPanels_x,0,0);
    Alarm_Panel = newwin(sizeSmallPanels_y,sizeSmallPanels_x,0,sizeSmallPanels_x);
    Output_Panel = newwin(sizeLargePanels_y,sizeLargePanels_x,sizeSmallPanels_y,0);
    Prompt_Panel = newwin(sizeSmallPanels_y,sizeLargePanels_x,sizeLargePanels_y+sizeSmallPanels_y,0);
    wprintw(Date_Panel,"Date Panel\n");
    wprintw(Alarm_Panel,"Alarm Panel\n");
    wprintw(Output_Panel,"Output Panel\n");
    wprintw(Prompt_Panel,"Prompt Panel\n");
    wbkgd(Date_Panel,COLOR_PAIR(2));
    wbkgd(Alarm_Panel,COLOR_PAIR(3));
    wbkgd(Output_Panel,COLOR_PAIR(4));
    wbkgd(Prompt_Panel,COLOR_PAIR(1));
    wrefresh(Date_Panel);
    wrefresh(Alarm_Panel);
    wrefresh(Output_Panel);
    wrefresh(Prompt_Panel);
}
```

Getting User Input

Any user operated infrastructure, needs to have some sort of mechanism which enables interaction between the user and the machine. To be able to operate our shell, the user will be interacting with it through commands inputted in the prompt panel. Such commands will be read and parsed in order to make it readable to the machine, by two methods; tokenizer() (which takes care of parsing, thus is discussed in the next section) and getInput().

The method getInput() first prints the current prompt variable (default being ok>) to the first row of the prompt panel, and then moves the cursor to the next coordinate, in order for the user to start inputting his commands.

User input is taken using a while loop which continues to loop until the user presses enter. Up until then, the function mvwgetch(), will continue to be invoked at the first row and next column position, where each time it records a single character. Note that if the character pressed by the user is backspace, its equivalent characters will be displayed on screen. However the user doesn't want that, and for this reason these useless characters displayed have to be compensated for by substituting them with empty spaces. Even more it has to be made sure that the last character inputted is not saved (the one for which the user has pressed backspace to eliminate) and neither the backspace characters which are attributed to the key. Even more in such a condition, the user was restricted from being able to erase the prompt display.

If none of the previous occurred and the user still had not pressed enter, the while loop would keep on looping and saving the characters inputted into an array of characters named commandBuffer. Such an array makes use of the variable tmp, which serves as an index used to update the array. With each character input, tmp is incremented to point to the next empty space of the array, as well column, which points to the next empty space on screen. After the user presses enter, which signifies the loop to stop looping since there was no more user input to be received, NULL was appended to the last position of the string, in order to signify to the parser (discussed in next chapter) that the command has ended.

```
void getInput(){
    int tmp = 0;
    //tmp is an index used to update last entry in array of chars with a null to terminate input
    //it is also used to restart command buffer at position 0
    char c; // c will store the current character
    mvwprintw(Prompt_Panel,row,col,prompt);
    col+=strlen(prompt);
    column[0] = col;
    while((c = (char)mvwgetch(Prompt_Panel,row,col))!= 10){
        if(c==127 && col >strlen(prompt)) {
            mvwprintw(Prompt_Panel, row, col, " ");
            mvwprintw(Prompt_Panel, row, col - 1, " ");
            mvwprintw(Prompt_Panel, row, col + 1, " ");
            col -= 1;
            wrefresh(Prompt_Panel);
            commandBuffer[tmp - 1] = '\0';
            tmp--;
        }else{
            commandBuffer[tmp] = c;
            col++;
            tmp++;
        }
        column[0] = col; //stores current column position in the shared memory segment
    }
    col =0; // resets column to position 0
    column[0] = 0; //resets shared memory segment's column position to position 0
    commandBuffer[tmp] = NULL; //ensures that the last string literal of the commandbuffer is a null
}
```

Parsing User Input

After the user presses enter and a NULL is appended to the commandBuffer, once can say that the user input has finished. At this point, the command that the user issued has to be parsed and translated in order for the software to make sense out of it. In order to achieve this, a while loop was initialised in order to loop through the whole commandBuffer until a NULL is encountered.

The first conditional statement in the while loop is that of incrementing a counter named spaces, with each space that is found in the commandBuffer. This will help keep track in which part of the command the loop has arrived, and thus enabling it to store the following contents of the commandBuffer into the appropriate smaller character arrays (which will then be passed to the appropriate built in functions as a parameter). Such a method was used, since different functions take different amount of parameters, however a common thing between all parameters is that they are delimited by a space.

If the amount of spaces amounts to zero, means that no space has been met yet, thus the parser is reading a command rather than a parameter. For this reason the parser will go through the commandBuffer contents and save its contents into another array called command, up until the first space is found.

When the amount of spaces found is equal to one, there are four possibilities of input;

- it is either a file path which the user wants to make use of, thus it contains no spaces
- it can be an output message that the user wants to display in the output buffer, thus it can contain spaces
- it can be the second part of the set command which can be considered as part of the command itself(will be discussed in a later section)
- it can be the output redirection command ‘>’

If it is the first and second cases, the parameters will be saved in an array called filepath and an array called output, respectively. However when it comes to the third possibility, the game changes a little

bit. Since the user specified that between the set command and the variable to be taken as a parameter, the delimiter will change from a space to an '=', the parser had to accommodate this. To be able revolve around this condition, a Boolean variable called isSetCommand was initialised to false. Whilst such a Boolean was set to false, the second parameter was being saved in an array called setcommand. However as soon as the '=' character was encountered, the Boolean variable was switched to true, which meant that the set command was completed, and the input following the '=' sign, was the parameter for the previously saved set command. Thus the rest of the input found up until the next space, is saved in an array called setinput (being the parameter for the previous set command).

> The `set` built-in command is special in that it allows customisation of parts of the shell. The command syntax is `set variable=value`, and the following variables need to be supported:

For the fourth and final case, if such a character was found, it meant that the user rather than wanting to display the output into the output buffer, he wanted to save the output into the file specified by the next parameter.

Note that the previously mentioned output redirection command implies that after such a character there will be the filename towards which the output redirection will occur. Since this type of command is still delimited by spaces (as per customer specification), the amount of spaces will be incremented to 2.

> By default, any output goes to the `output panel`; this can be overridden through output redirection: `cmd > $filename`; so for example, typing `shdir` will output `/home/user/development` to the `output panel`, but typing `shdir > myfile.txt` will create or overwrite a file called `myfile.txt` with the text.

In this case there are three possibilities if the number of spaces are larger than one:

- the rest of the command buffer is part of the output message issued by the user (since messages usually contain more than one space)
- it can be the filename towards which the output redirection will occur
- if it is larger than 2, than that is the parameter which will be passed to the command which issues an output, and makes use of the output redirection function

To cover the first possibility, the rest of the output will be saved in the output array which was already initialised in the previous condition. If the amount of spaces is exactly to two, the following contents of the commandBuffer are stored in the filename array. Finally if the amount of spaces is greater than two, the rest of the contenst of the commandBuffer are stored in an array called output2, whose function is already explained in the previous bullet points. Note that in this case, the index of the commandBuffer is incremented by one such as to avoid saving the delimiting space character.

```
void tokenizer(){
    //counters for different arrays used
    int counter_commandBuff = 0;
    int counter_command = 0;
    int counter_output = 0;
    int counter_filepath = 0;
    int counter_setcommand = 0;
    int counter_setinput = 0;
    int counter_filename = 0;
    int counter_output2 = 0;
    //counter_output2 for buffer which will be filled only by text file for the print function
    int spaces = 0;
    bool isSetCommand = false;
    while(commandBuffer[counter_commandBuff] != NULL){
        if(commandBuffer[counter_commandBuff] == 32){
            spaces++;
        }
        if(spaces == 0){
            command[counter_command] = commandBuffer[counter_commandBuff];
            counter_command++;
        }else if(spaces == 1 && commandBuffer[counter_commandBuff] != 32){
            filepath[counter_filepath] = commandBuffer[counter_commandBuff];
            output[counter_output] = commandBuffer[counter_commandBuff];
            counter_filepath++;
            counter_output++;
            //taking this approach avoids saving the'=' : 61
            if(commandBuffer[counter_commandBuff] == 61){
                isSetCommand = true;
                goto Escape;
            }
            if(!isSetCommand) {
                setcommand[counter_setcommand] = commandBuffer[counter_commandBuff];
                counter_setcommand++;
            }else{
                setinput[counter_setinput] = commandBuffer[counter_commandBuff];
                counter_setinput++;
            }
        }else if (spaces > 1) {//saves spaces as well
            output[counter_output] = commandBuffer[counter_commandBuff];
            counter_output++;
            if (spaces == 2){//saves the new file name
                filename[counter_filename] = commandBuffer[counter_commandBuff];
                counter_filename++;
            }else if (spaces>2){
                int localcounter_commandBuff = counter_commandBuff;
                localcounter_commandBuff++;
                //the previous 2 lines avoid saving the first ' ' by moving the commndbuff by 1 offset
                output2[counter_output2] = commandBuffer[localcounter_commandBuff];
                counter_output2++;
            }
        }
        Escape:
        counter_commandBuff++;
    }
}
```

Due to the large amount of data, and the large amount of arrays initialised to store it, it would be a strenuous task to clear each array one by one after each loop iteration, to prevent from messing the next user input to be parsed. For such reasons arraycleaner() was constructed.

Such method takes an array of characters as a parameter, and iterates through all of its contents, and clearing them using the standard method memset(). In conjunction with this, method clean_and_update() was constructed in order to invoke the method arraycleaner() on each array of characters which are storing the parsing results, clear the user input from the prompt panel, after he presses enter (to start parsing it), and refreshing all of the panels with the latest outputs. Such a method is invoked after each iteration of the loop, such as to enable the user to input new commands without being affected by the previous operations.

```
void arraycleaner(char buffer[]){
    memset(&buffer[0],0,strlen(buffer));
}

void clean_and_update(){
    arraycleaner(&commandBuffer[0]);
    arraycleaner(&command[0]);
    arraycleaner(&filepath[0]);
    arraycleaner(&output[0]);
    arraycleaner(&setcommand[0]);
    arraycleaner(&setinput[0]);
    arraycleaner(&filestorage[0]);
    arraycleaner(&filename[0]);
    arraycleaner(&output2[0]);

    wclrtobot(Prompt_Panel);
    wnoutrefresh(Prompt_Panel);
    wnoutrefresh(Output_Panel);
    wnoutrefresh(Date_Panel);
    doupdate();
}
```

Saving output: Buffer

As per requirement, there are some commands which will be discussed in the next section, which return an output to the user. Even more whenever a command fails for some reason or another, the user needs to be prompted accordingly. This sort of output is what makes the shell user friendly, making the user able to see what is going on, without having to go into the actual code. For such reasons, the output panel was constructed. However the output panel per default does not keep track of what the output was, which is a bit of a problem. Thus, the customer required a buffer with a dynamic size, to be built in order to store the output and enable the user to scroll through the history of outputs.

Output Panel

The `output panel` is a large horizontal strip (multiple lines) on the screen displaying the output buffer in part or whole. The output buffer size (determined by the `buffer` shell variable) can span the size of multiple panels; your buffer might contain more lines that are being displayed by the `output panel`. Therefore the `move` shell command determines which part of the buffer is displayed in the panel.

To achieve this an array of structures was constructed, where each structure consisted of an array of characters (this would mimic one line). Even though the maximum size of the array of characters as well as the array of structures, were fixed, the size which the user had access to, was dynamic and could be set by the user himself (using the in-built command "set buffer", which will be discussed in the next section). The default user size of 80, was stored in a variable called userSize.

```
//building buffer
struct line{
    char columns [COLUMNS];
}bufferRows[9999];
int buffer_rowCounter = 0;
int userSize = 80;
```

However, to be able to save and retrieve the output to and fro, from such a structure, two methods were written; bufferWriter() and bufferReader().

bufferWriter()

This method takes an array of characters as a parameter, and saves its contents into the buffer structure. This is done by using a while loop which iterates through the length of the array which stores the output which needs to be written to the buffer, and copies its contents into the next available buffer row. Note that a global buffer row counter; "buffer_rowCounter" was initialised in order to keep track which is the next row available and increments with each line written.

In order to prevent form overflowing the structure, a condition was introduced, that forces the buffer counter to switch to the next available line (incrementing array of structure index), and keep on storing there. This also occurs if a new line '/n' character is encountered.

To prevent the whole buffer overflow(since the user can modify the size), a mechanism that checks whether the size allocated was exceeded (by comparing the buffer_rowCounter with the size allotted by the user), was introduced. If there is an overflow and the condition is met, the buffer will automatically flush itself and start saving again from the beginning.

```
void bufferWriter(char outputBuffers []){
    int counter_i = 0; //will count the outputBuffers index
    int counter_j = 0; //will count the columns of buffer

    if(buffer_rowCounter>=userSize) {
        for (int i = 0; i < userSize; i++) {
            arraycleaner(bufferRows[i].columns);
        }
        bufferReader();//clears screen
        buffer_rowCounter = 0;//resets index of buffer to 0
    }
    while (counter_i < strlen(outputBuffers)) {
        if (counter_i > COLUMNS || outputBuffers[counter_i] == '\n') {
            // if horizontal buffer size is exceeded or a new line character is encountered,
            // it will continue to print in a new line
            counter_j = 0;
            buffer_rowCounter++;
            counter_i++;
        } else {
            bufferRows[buffer_rowCounter].columns[counter_j] = outputBuffers[counter_i];
            counter_i++;
            counter_j++;
        }
    }
    buffer_rowCounter++;
}
```

bufferReader()

Whilst bufferWriter() was used to save all the output in a saving structure, bufferReader() is used to display the output in the output panel. To be able to do this, the panel is cleared with each method call using the standard function werase(), and the buffer is printed.

To print the buffer contents on the panel, a for loop was initialised, where it loops through each structure in the array of structure and displays its contents up until the current position of the buffer, using buffer_rowCounter as a restriction (since the rest would be empty or unreachable).

```
void bufferReader(){
    int localrowcounter = 1;//determine where buffer reader will print
    int localcolumncounter = 0;
    werase(Output_Panel);
    wprintw(Output_Panel, "Output Panel");
    for(int i = movValue; i <buffer_rowCounter; i++){
        mvwprintw(Output_Panel,localrowcounter,localcolumncounter,bufferRows[i].columns);
        localrowcounter++;
    }
}
```

In-Built Commands

In order for this shell to function, the customer required to have 7 in-built commands. These were commands tailor made for the custom shell in order to achieve certain specifications which the customer required. (Note that for all the following functions to work, they need to be followed by a space and then the necessary input.)

CHDIR()

The first in-built function was that of chdir(). This function would need to get the current working directory and change it to the one following the command (stored in the filepath array), which the user inputted:

chdir change current directory, followed by the desired directory
(e.g. chdir /home/sherlock)

The following approach to tackle this function was taken:

1. The method checks whether the first character of the array storing the new path is an end of line character. If this returns to be true it means the user did not input any new path, thus he is prompted accordingly.
2. If the first character is not an end of line character, it means there is some sort of path stored. For this reason, the validity of this path is checked through the use of an if statement, where if the standard function call chdir() does not return zero, means that the path is not valid, thus the user is prompted accordingly, or else if it returns zero the current working directory is switched to the valid path using the previously mentioned standard function.

```
void chdirFunction(){
    if (filepath[0] == '\0'){
        wprintw(Output_Panel,"expected path : /home/franklyn07/...");
    }else {
        if(chdir(filepath)!=0){
            wprintw(Output_Panel,"%s: No such file or directory!",filepath);
        }
    }
}
```

SHDIR ()

The second in-built function, which is quite related to the previous one, is that of shdir(). This function displays the current working directory to the user, as per requirement.

shdir output the current directory
(e.g. shdir)

To be able to achieve this, the system call getcwd() was invoked, where it gets the current working directory and stores it in the array you give it as a parameter.

If no output redirection was detected by the parser, the output would be saved to the buffer using bufferWriter() and then displayed on screen through bufferReader().

However if output redirection is detected, a new text file with the name given by the user and stored in the array filename, is created, if it doesn't already exist. On the other hand if it already exists, the new output will be appended to the already existing contents using the function fopen(), with 'a+', as a second parameter, which enables the appending. Fwrite() is then used to save the contents of the array storing the path to the opened file. To be able to notify the user whether the saving to the file was successful or not, a new condition was implemented where the appropriate prompt, according to whether the output was saved successfully or not, was written to a temporary array, which was then passed to the bufferWriter() method as a parameter in order to write the prompt to the buffer and be able to display it. After the execution is completed, the array was cleared using the previously mentioned arraycleaner() and the file was closed by invoking the function call fclose().

```
else if ((strcmp(command, "shdir")) == 0) {
  if((strcmp(filepath, ">")) == 0){
      char tmp[240]; //temporary array which stores cwd for file to save
      FILE *file = fopen(filename,"a+");
      if(file == NULL){
          strcpy(tmp,"Error creating file!");
          bufferWriter(tmp);
          bufferReader();
          arraycleaner(tmp);
      }else {
          getcwd(tmp, sizeof(tmp));
          fwrite(tmp, 1, strlen(tmp), file);
          fwrite("\n", 1, 1, file);
          arraycleaner(tmp);
          strcpy(tmp, "Saved Successfully!");
          bufferWriter(tmp);
          bufferReader();
          arraycleaner(tmp);
      }
      fclose(file);
  }else {
      getcwd(filepath, sizeof(filepath));
      bufferWriter(filepath);
      bufferReader();
  }
```

PRINT()

The print() in built command is the command which requires from the software to print whatever is passed as a parameter to it.

print output the text that follows the command
(e.g. print This is a message!)

The same principles applied to shdir() are applied here. If there is no output redirection detected, bufferWriter() will take the output array as a parameter, where the parser has the message the user wants to output, stored. bufferReader() performs the usual printing to the output panel.

On the other hand if output redirection is detected, a file is opened using the method mentioned in the shdir() section, and the message the user passed as a parameter, now stored in the array output2, is written to the file which the user indicated, using the function fwrite(). The user is prompted appropriately, temporary array was cleared, and the file is closed. (For further detail on how this works, refer to shdir() section)

```
else if ((strcmp(command, "print")) == 0) {
  if((strcmp(filepath, ">")) == 0){
      char tmp[240]; //temporary array which stores user interaction output
      FILE *file = fopen(filename,"a+");
      if(file == NULL){
          strcpy(tmp,"Error creating file!");
          bufferWriter(tmp);
          bufferReader();
          arraycleaner(tmp);
      }else {
          fwrite(output2, 1, strlen(output2), file);
          fwrite("\n", 1, 1, file);
          arraycleaner(tmp);
          strcpy(tmp, "Saved Successfully!");
          bufferWriter(tmp);
          bufferReader();
          arraycleaner(tmp);
      }
      fclose(file);
  }else{
      bufferWriter(output);
      bufferReader();
  }
```

PRINTVAR()

This in-built command was required to output the environmental variables which the user desires (ex. Home, path, etc.), which the shell in operation, uses.

printvar outputs the value of the specified shell variable
(e.g. printvar path)

In order to compile such a command the system call getenv() had to be used. However this system call takes its parameters in capital form. In order to ensure that the user does not forget to input such a parameters in the correct form, a local array was created and called local_buff. A while loop was also constructed and used in order to get the user inputted parameters, stored in output2, copy them to local_buff, by using the standard function strcpy(), and then transformed character by character, into capital form using the standard function toupper(). Then local_buff is passed as a parameter to the system call getenv(), where the returned string is either written to the buffer and displayed in the output panel or written to a file, indicated by the user, depending on whether output redirection command is detected or not.

```
void printvarFunct(){
    int i = 0;
    char local_buff[256];
    arraycleaner(local_buff);

    if((strcmp(filepath, ">")) == 0){
        strcpy(local_buff,output2);
        //the function only takes variables in capslock so we convert each character toupper
        while(i<strlen(local_buff)){
            local_buff[i] = (char) toupper(local_buff[i]);
            i++;
        };
        char tmp[240]; //temporary array which stores user interaction output
        FILE *file = fopen(filename,"a+");
        if(file == NULL){
            strcpy(tmp,"Error creating file!");
            bufferWriter(tmp);
            bufferReader();
            arraycleaner(tmp);
        }else {
            if(getenv(local_buff)!=NULL){
                fwrite(getenv(local_buff), 1, strlen(getenv(local_buff)), file);
                fwrite("\n", 1, 1, file);
                arraycleaner(tmp);
                strcpy(tmp, "Saved Successfully!");
                bufferWriter(tmp);
                bufferReader();
                arraycleaner(tmp);
            }else{
                arraycleaner(local_buff);
                //clears local buffer to be able to write errror prompt in it
                strcpy(local_buff,"Error occurred!");
                bufferWriter(local_buff);
            };
            fclose(file);
        }
```

```
    }else{
        strcpy(local_buff,filepath);
        //the function only takes variables in capslock so we convert each character toupper
        while(i<strlen(local_buff)){
            local_buff[i] = (char) toupper(local_buff[i]);
            i++;
        };
        if(getenv(local_buff)!=NULL){
            bufferWriter(getenv(local_buff));
        }else{
            arraycleaner(local_buff);//clears local buffer to be able to write errror prompt in it
            strcpy(local_buff,"Error occurred!");
            bufferWriter(local_buff);
        };
        bufferReader();
    }
}
```

SET()

The fourth in built command, is set(), and it doesn't work on its one. With it, it has another 4 sub commands, which the client specified. These commands in one way or another are all able to customise parts of the shell, however unlike the previous in builts, use this specific syntax; setcommand set-sucbommand=value. The four sub commands are:

- Prompt – changes the default ok into a string specified by the user
- Path – changes the path shell variable, into the one specified by the user
- Refresh – changes the rate with which the date – panel is refreshed (shall be discussed in the task 3 section)
- Buffer – changes the size of the buffer which is accessible by the user (default being 256 by 80)

The `set` command

The `set` built-in command is special in that it allows customisation of parts of the shell. The command syntax is `set variable=value`, and the following variables need to be supported:

`prompt` changes the prompt string; the default prompt string is `OK>`.

`path` string the system uses to locate and launch external binaries; each entry (a path) is separated by `:` from the next, e.g.: `/usr:/usr/bin:`

`refresh` changes the refresh rate of the clock in the `date-time panel`. The default value is `1` second.

`buffer` changes the size of the output buffer capacity in terms of characters. The default value is `80 by 256` characters.

Prompt

In order to change the default prompt from "ok>" to the prompt supplied by the user, firstly the array cleaner is invoked on the array prompt which as the name implies, holds prompt. Next using standard method stcrpy(), the user driven prompt which is currently stored in the array setinput (by the parser), is copied into the prompt array. Finally the ">" is concatenated to the end of the prompt using the standard method strcat().

Path

To change the current path of the shell, the system call setenv() was invoked. This system call takes the environmental variable which needs to be changed as a first parameter; where in our case it was hard-coded to PATH, since that is the concerned variable. The second parameter is the array which holds the new path, where in our case the parser will put such value, in the array setinput. Whilst the final parameter is that of whether we want the environmental variable to be overwritten or not if the name already exists, in our case we wanted it to be overwritten.

Refresh

Will be discussed in the section for task 3.

Buffer

Setting the buffer size requires the use of the atoi() standard function. This function will be used in order to return the integer value of the numbers stored as characters in the array setinput. Such an integer is then used to overwrite the default userSize global variable, which restricts the accessible size of the buffer structure. The new size alongside with the appropriate prompt, are concatenated in the setinput array, whose contents are then written to the buffer and displayed on the output panel.

```
else if ((strcmp(command, "set")) == 0) {
  if ((strcmp(setcommand, "prompt")) == 0) {
      arraycleaner(&prompt[0]);//clears previous input
      strcpy(prompt, strcat(setinput, ">"));
  } else if ((strcmp(setcommand, "path")) == 0) {
      setenv("PATH",setinput, 1);
  } else if ((strcmp(setcommand, "refresh")) == 0) {
      sleeper[0]=atoi(setinput);
  } else if ((strcmp(setcommand, "buffer")) == 0) {
      userSize = atoi(setinput);
      strcat(setinput,":New buffer size!");
      bufferWriter(setinput);
      bufferReader();
  } else {
      strcpy(setcommand,"Unkown command!");
      bufferWriter(setcommand);
      bufferReader();
  }
```

Finally as on can notice from the previous code, if no set command matches, the appropriate prompt will be written to the buffer and displayed on the output panel.

MOVE()

In the move built in function, it was required to be able to shift the buffer display around, since the buffer size could be greater than the available display. The following is the requirement given;

> **move** scroll through the output buffer by the specified number of lines (argument is an offset, in number of lines; negative values move backwards in history, positive values move forwards) (e.g. `move -5`)

In order to implement this a global variable movValue was initialised having the default number set to 0. If one goes back to the bufferReader() method, one can notice that the for loop which begins reading from the array of structures, uses moveValue as its initial index. This would enable the user to change from which line he wants the buffer to be displayed. The value by which the user wanted the buffer to move was achieved from the array filepath, by applying the previously mentioned standard function atoi(). Such integer will be incremented to the original movValue, anytime the move function is invoked. Note that if the movValue will be less than 0 (which in terms of buffer size doesn't make sense) or larger than userSize (which is an inaccessible area of the buffer), the buffer will print an error to the output panel.

```
} else if ((strcmp(command, "move")) == 0) {
    movValue += atoi(filepath);
    if(movValue < 0 || movValue>userSize){
        strcat(setinput,"Error!");//just found an empty buffer and used it as a temporary storage
        movValue -= atoi(filepath);//revert mathematical operation
        bufferWriter(setinput);
        bufferReader();
    } else {
        bufferReader();
    }
```

EXIT ()

As the name of the requirement in itself implies, this in built command is used to stop all the ongoing processes, perform a clean-up of whatever the shell was making use of and quit.

exit close the shell and quit
(e.g. `exit`)

The following approach to tackle this command was taken:

1. Since the shell needed to ask the user at least once for input before stopping, a do while loop was used to keep getting the user input and perform the required operations.
2. However, there needed to be some sort of blocking system which would stop such loop, when the exit command was received.
3. In order to achieve this an integer variable (closeShell) was initialised to 0. The loop would keep on looping while closeShell's value was zero.
4. When the exit command was inputted, this variable would be incremented, thus the do while loop would break, and the clean-up operations would commence (finalCleanup()).

External Commands

If the command entered does not match to any of the built in functions, the external commands section comes in place. In such an occurrence the shell will traverse the string inputted form left to right and will try to match it to an external command.

External commands

The shell should be able to run external commands using a simple input mechanism: if the entered command is not recognised as a built-in function, it is assumed to be an external command. The system traverses the path string from left to right, one entry at a time, and tries launching the external command using the current path substring. If no such command can be found, the user should be alerted by printing a message in the `output panel`. Example: typing `ls -la` will execute the Unix list command and redirect its output to the `output panel`.

To achieve this, the contents of the commandBuffer are passed as a parameter to the system call popen(), which will open a pipeline to the system, where you either read or write (which would be the second parameter of the function). In our case it will be read. If the command will fail, a NULL value will be returned and the user will be prompted appropriately. On the other hand if the external command is found, its output is written to a text file. To be able to change that file content to buffer content method fil_to_array() was compiled. This method takes a pointer to a file as a parameter and uses a while loop to traverse the file towards which such pointer is pointing. The contents of the text file are copied into an array called filestorage, using the standard method fgetc(). Then the contents of such an array are written directly to the buffer, using the usual bufferWriter().

```
else {
   FILE *f;
   if((f = popen(commandBuffer, "r")) == NULL) {
       fputs("External command not found!",testing);
   }else{
       file_to_array(f);
       pclose(f);
       bufferWriter(filestorage);
       bufferReader();
   }
```

```
void file_to_array(FILE* file){
    int i = 0;
    while((filestorage[i] = (char)fgetc(file))!=EOF){
        i++;
    }
    filestorage[i] = '\0';
    for(int j = 0; j<strlen(filestorage);j++){
        if(filestorage[j] == '\n'){
            filestorage[j] = 32;
        }
    }
}
```

Extra Functionalities

The following are methods which were not directly required by the client, but were implemented due to their usage.

FINALCLEANUP()

This method basically takes care of the final clean up, just as the name implies, of whenever the main do while loop exits. Its job is to delete all the panels and end the screen holding everything.

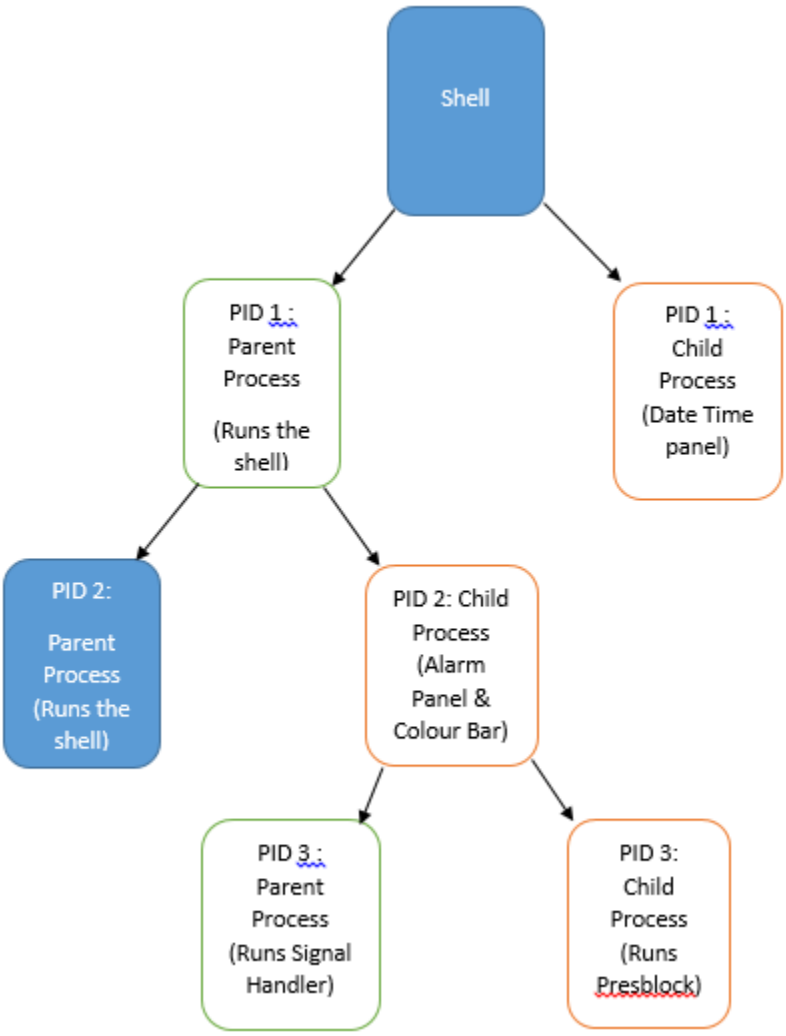
```
void finalCleanup(){
    delwin(Prompt_Panel);
    delwin(Alarm_Panel);
    delwin(Date_Panel);
    delwin(Output_Panel);
    delwin(Main);
    endwin();
    refresh();
}
```

Clear

Such in built function is used to clear the current buffer and display the new empty buffer on the output panel. This is done by triggering the exceeding userSize condition in bufferWriter(), and thus enabling the total buffer clean up.

```
} else if ((strcmp(command, "clear")) == 0){
    buffer_rowCounter = userSize;
    //this will activate condition in buffer writer, that clears the buffer
    bufferReader();
```

TASK 2 – ALARM PANEL



Shared Memory Segments:	Processing making use of them
Refresh Counter	Pid1 Parent & Child, Pid2 Parent
Cursor Postion	All except Pid3 Child
Exit Status	All except Pid3 Child

Following the previous two diagrams one can notice that system call fork() was used twice in this section (the one used for the date-time panel will be explained in the next section). The Pid2 child, in this section was used to initialised and display the alarm panel, and the colour bar, as well as to attach the process child2 to the already constructed shared memory segments (once again will be explained in the next section). However note that in such processes only the cursor position and exit status memory segments were useful, since the refresh counter memory segments will only be used in the next task. Such memory segments were attached two in this child in order, for the parent process pid3 to also have access to them.

Note that in child pid2, the system call kill() was used. Such system call was used to delete the child process whenever the user opted to exit the terminal; changing the status of the shared memory process.

```
}else if (pid2 == 0){
    pid_t pid3;

    //child process
    sleep(0.5); //gives enough time to date and time panel to print the time (w/o moving cursor)

    //looking for shm
    fputs("Child2 process:",testing);
    fputs("Consumer2 started",testing);


    if ((shmid2 = shmget(key2,1,0666)) < 0) {
      fputs("shmid1 parent",testing);
      exit(1);
    }

    if ((shmid3 = shmget(key3,1,0666)) < 0) {
      fputs("shmid1 parent",testing);
      exit(1);
    }


    // Attach to the segment.

    if ((status_1 = shmat(shmid2, NULL, 0)) == (int *) -1){
      fputs("status attachment in parent",testing);
      exit(1);
    }

    if ((column_1 = shmat(shmid3, NULL, 0)) == (int *) -1){
      fputs("column attachment in parent",testing);
      exit(1);
    }
```

```
status = (int*)status_1;
column = (int*)column_1;

Alarm_Panel = newwin(sizeSmallPanels_y,2*(sizeSmallPanels_x/3),0,sizeSmallPanels_x);
Colour_Panel = newwin(sizeSmallPanels_y,(sizeSmallPanels_x/3),0,sizeSmallPanels_x+(2*(sizeSmallPanels_x/3)));

wprintw(Alarm_Panel,"Alarm Panel\n");

init_pair(3,COLOR_BLACK,COLOR_BLUE);
init_pair(8,COLOR_BLACK,COLOR_WHITE);
init_pair(5,COLOR_BLACK,COLOR_RED);
init_pair(6,COLOR_BLACK,COLOR_GREEN);
init_pair(7,COLOR_BLACK,COLOR_YELLOW);

wbkgd(Colour_Panel,COLOR_PAIR(8));
wbkgd(Alarm_Panel,COLOR_PAIR(1));
wrefresh(Colour_Panel);
wrefresh(Alarm_Panel);
```

In the child process pid3, another system call called execvp() was called. This system call, is used to replace the image of the process with the one it is invoking in its path, where in our case was presblock.c. Such process would randomly generate different signals at different time.

```c
if(pid3 == 0){
    //child prcess

    //call pressbloc
    sleep(0.5); // to wait for parent process to start before it
    int pid_curr = getppid();
    char buff [10];
    snprintf(buff,sizeof(buff),"%d",pid_curr);
    char * const currentpid_c [] ={"presblock",buff, NULL} ;

    execvp("./presblock",currentpid_c);
```

```c
//Pressblock.c
int main(int argc, char** argv)
{
    FILE *trouble = fopen("presfile","a+");
  if (argc < 2) {
    fprintf(trouble, "Incorrect number of arguments; usage: presblock [pid]\n");
    fclose(trouble);
    exit(EXIT_FAILURE);
  }

  // Convert text pid to int
  int proc_pid = atoi(argv[1]);
  fprintf(trouble,"Process ID: %d",proc_pid);

  // Check if pid is a sane value
  if (proc_pid > 0) {

    while(proc_pid == getppid()) {
      // AGENT Smith:
      // rand() is the worst PRNG ever; FFS use something else!
      int delay = rand() % MAX_INTERVAL;
      fprintf(trouble, "Delay: %d\n", delay);
      sleep(delay);
      kill(proc_pid, SIGALRM);
    }
  } else {
    fprintf(trouble, "Invalid pid! Exiting...\n");
    fclose(trouble);
    exit(EXIT_FAILURE);
  }

  // Unreachable code
  exit(EXIT_SUCCESS);
}
```

The job of the parent process on the other hand, was to be able to catch these signals using the sytem call signal(). When the signal was caught, the signal handler was invoked, and thus this would invoke the update function, which would update the alarm panel and the colour bar, according to the duration between one signal and another. However this would only go on up until the user changed the shared memory segment holding the status, in such case, the parent would detach itself from the shared memory segments, making them available for clean-up.

```
}else if (pid3>0){
    //paretn process

    signal(SIGALRM, signalhandler);
    while(status[0]==0){
        sleep(1);
        timer++;
    }

    shmdt(status_1);
    shmdt(column_1);

}
```

```
void signalhandler(int signal){
    if(signal == SIGALRM){
        wmove(Alarm_Panel,1,0);
        mvwprintw(Alarm_Panel,1,0,"%d seconds from last signal!",timer);
        wrefresh(Alarm_Panel);
        datepanelupdate();
        timer = 0; //if signal is caught we will reset timer to 0.
    }
}

void datepanelupdate(){
    wmove(Colour_Panel,0,0);
    if(timer>20){
        wbkgd(Colour_Panel,COLOR_PAIR(3));
        wrefresh(Colour_Panel);
    }else if(timer <= 20 && timer >= 16){
        wbkgd(Colour_Panel,COLOR_PAIR(6));
        wrefresh(Colour_Panel);
    }else if(timer <= 15 && timer >= 11){
        wbkgd(Colour_Panel,COLOR_PAIR(7));
        wrefresh(Colour_Panel);
    }else if(timer <= 9 && timer >= 5){
        wbkgd(Colour_Panel,COLOR_PAIR(5));
        wrefresh(Colour_Panel);
    }else{
        wbkgd(Colour_Panel,COLOR_PAIR(8));
        wrefresh(Colour_Panel);
    }
    wmove(Prompt_Panel,row,column[0]);
    wrefresh(Prompt_Panel);
}
```

(Note that the method should have been named alarmpanelupdate())

TASK 3 - DATE-TIME PANEL

In task 3 it was required of the programmer, to write a date-time panel which would keep continuous updates of the current time in three different locations, with the locations being :

- Whitehouse
- Malta
- Tokyo

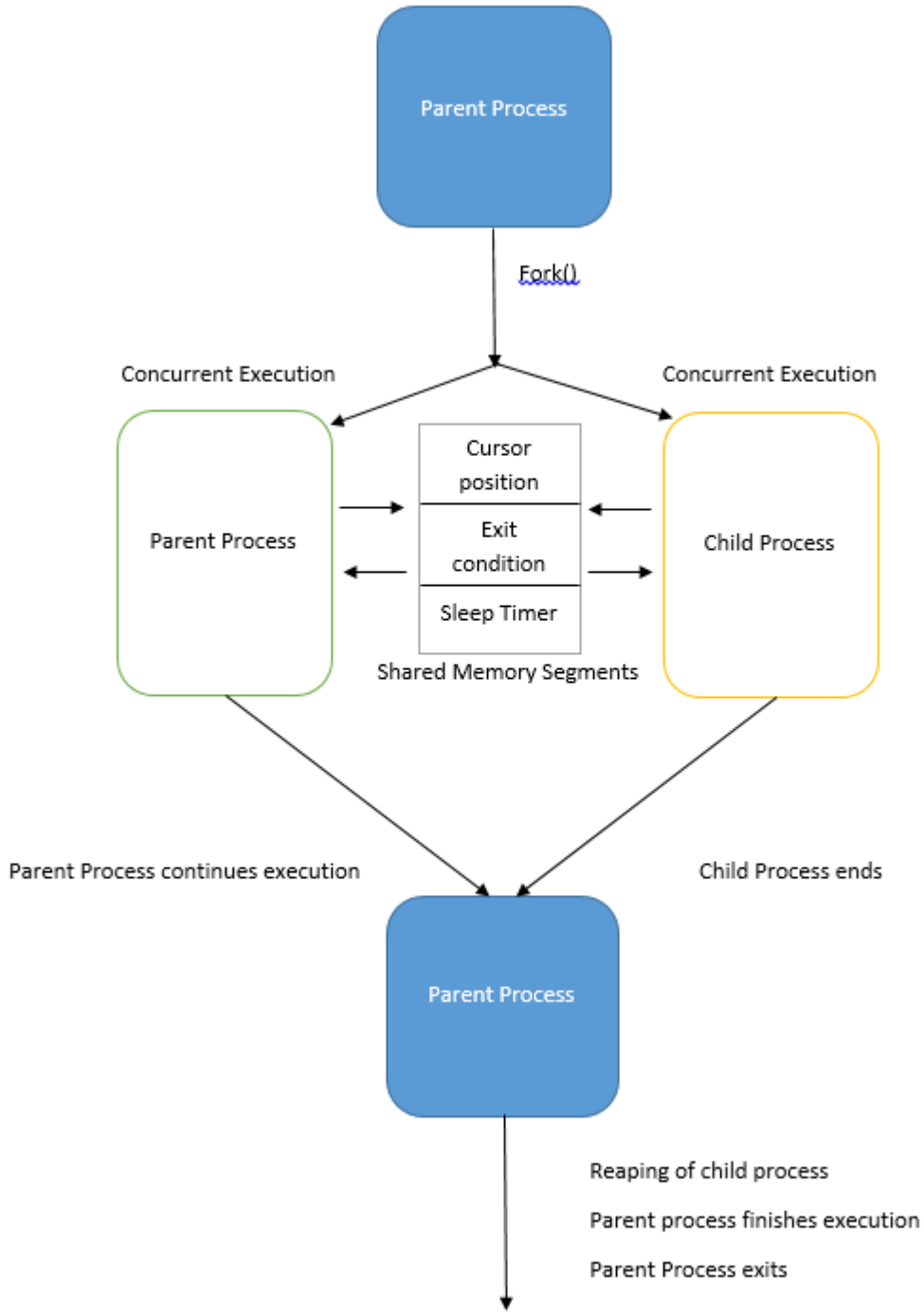
The `date-time panel` keeps an up-to-date display of the current time and date for the following time zones:

1. White House [Central Time Zone UTC-06:00]
2. Malta [Central European Time GMT+1]
3. Tokyo [Japan Time Zone UTC+09:00][2]

An independent child process (forked off the shell parent) is responsible for updating a shared memory segment (shared with the parent) with date and time information. The shared segment is the same size, in characters, of the screen area allotted to the panel. Thus, the panel can decide the visual layout of the three clocks.

Note: There is no need to factor in daylight saving time.

In order to achieve this, child processes and shared memory segments had to be used. The following is a diagram which represents the architecture constructed to be able to obtain the desired results:



In the main method which was found in the parent process, a child process was forked in order to have two processes running concurrently. This was done using they system call fork(), which returns twice, once it returns a 0 if it is in the child process, or else any other number greater than 0 if it is in the parent process. However this as done after invoking the initialiser() method, which was described in Task 1, in order for the two processes to inherit the memory allocation for the different panels.

Child Process

Keeping in mind the return values, 3 shared memory segments were created and attached to in the child. This was done through the use of the system calls shmget(), and shmat(). Shmget() takes the memory segments from memory, and will make them available for its process. It takes three parameters, the first one being the key which is the memory segment identifier, the second one being whether we want to find a new memory segment if the one with the key used is already taken, and the third parameter is the protocol with which the memory segment will be created. On the other hand system call shmat() will take the available memory procured by shmget() and will attach a pointer to it, which will be used to write directly to it.

As the diagram suggest, the first memory segment created was that for the cursor position. This shared memory segment was necessary such that the two processes know where the cursor is supposed to be when trying to print on the screen using the mvwprintw() functions, which takes the cursor position as a parameter. The second shared memory segment created was that of the sleep timer. This was of use since as described in task 1, the user had the possibility to alter the refresh time of the clock, through the in-built function; set refresh. For such reasons the two processes needed to be able to communicate this value between each other. This value would then be passed to the system call sleep() found in the child process, which would delay the clock before refreshing by, the amount of time stipulated by the user.

Note that in order for the date to be displayed in a proper way, the function dateconverter() was written. This was due to the fact, that the structure used returned integers, rather than strings, thus instead of Sunday a 0 would be returned for the day of the week attribute. To revolve around this two parameters were passed to the mentioned function, where the first one would indicate whether the value received would be a day of the week or month, and the second would be the attributed integer. Then two arrays, one containing the days of the week and the other containing the months were initialised. The second parameter would be passed to such array according to whether it was supposed to be a day of the week or month (depending on the first parameter), and the appropriate string would be returned.

```
char * dateconverter(int boolean, int toconvert) {
    // 1 will mean its a day of the week
    // 0 will mean its a month

    static char *months[] = {"January", "February", "March", "April", "May", "June", "July", "August", "September",
                             "October", "November", "December"};
    static char *days[] = {"Sunday", "Monday", "Teusday", "Wednesday", "Thursday", "Friday", "Saturday"};

    if (boolean == 1) {
        return days[toconvert];
    } else if (boolean == 0) {
        return months[toconvert];
    }
}
```

The third and final shared memory segment created was that of the status. This was the only mean available through which the child process would know that it should terminate. To activate this exit state, the parent process would send a variable of 1 (whenever the user entered exit as command), to the child process which would break the infinite loop that was going on. Such infinite loop was initialised in the child process in order for the time and date to be continuously updated in the date and time panel.

After such infinite while loop in the child process would be broken, a cleaning up process would begin, where firstly the pointers were detached from the shared memory segments, using the system call shmdt(). Then these memory segments would be deleted using the system call shmctl(), which as a first parameter, it takes which memory segment would be deleted (through its generated id) and the protocol with which it would be deleted.

```
//shared memory segment variables
int shmid1,shmid2,shmid3;
int *sleeptimer_1;
int *status_1;
int *column_1;

int *column;//declared global such that method get input can access it ()
int *sleeper;
int *status;

key_t key1 = 0x1234; //memory segment identifier
key_t key2 = 0x5678;
key_t key3 = 0x2468;
```

```
    pid_t pid1;
    pid1 = fork();

    if(pid1<0){
        perror("Fork error!");
    }else if (pid1 == 0){
    //child process
        fputs("Child process:",testing);
        char tmp2[256];

        fputs("Producer Started\n",testing);
        // Create the segments.
        if ((shmid1 = shmget(key1, 1, IPC_CREAT|0666)) < 0){
            fputs("shmid1 child",testing);
            exit(1);
        }

        if ((shmid2 = shmget(key2, 1, IPC_CREAT|0666)) < 0) {
            fputs("shmid2 child",testing);
            exit(1);
        }

        if ((shmid3 = shmget(key3, 1, IPC_CREAT|0666)) < 0) {
            fputs("shmid3 child",testing);
            exit(1);
        }
```

```
// Attach to the segment.
if ((sleeptimer_1= shmat(shmid1, NULL, 0)) == (int *) -1) {
    fputs("sleep timer attachment in child",testing);
    exit(1);
}

if ((status_1 = shmat(shmid2, NULL, 0)) == (int *) -1) {
    fputs("status attachment in child",testing);
    exit(1);
}

if ((column_1 = shmat(shmid3, NULL, 0)) == (int *) -1) {
    fputs("column position attachment in child",testing);
    exit(1);
}


sleeper = (int*)sleeptimer_1;
status = (int* )status_1;
column = (int* )column_1;

status[0] = 0;
sleeper[0] = 1;
column[0] = 0;
```

```
while(status[0] == 0)
{
    int WhiteHouse = -5;
    int Malta = 2;
    int Japan = 9;

    time_t rawtime;
    struct tm *info;

    time(&rawtime);
    info = gmtime(&rawtime);

    mvwprintw(Date_Panel,2,0,"WHITE HOUSE [USA]: %2d:%02d:%02d -> %s,%d %s,%d",
              (info->tm_hour+WhiteHouse)%24, info->tm_min, info->tm_sec, dateconverter(1,info->tm_wday),info->tm_mday,dateconverter(0,info->tm_mon),info->tm_year+1900);
    mvwprintw(Date_Panel,3,0,"MSIDA [MALTA]: %2d:%02d:%02d -> %s,%d %s,%d",
              (info->tm_hour+Malta)%24, info->tm_min, info->tm_sec, dateconverter(1,info->tm_wday),info->tm_mday,dateconverter(0,info->tm_mon),info->tm_year+1900);
    mvwprintw(Date_Panel,4,0,"TOKYO [JAPAN]: %2d:%02d:%02d -> %s,%d %s,%d",
              (info->tm_hour+Japan)%24, info->tm_min, info->tm_sec, dateconverter(1,info->tm_wday),info->tm_mday,dateconverter(0,info->tm_mon),info->tm_year+1900);

    wmove(Prompt_Panel,row,column[0]);
    wrefresh(Date_Panel);
    wrefresh(Prompt_Panel);
    sleep(sleeper[0]);
}

fputs("Producer Ended\n",testing);
```

```
//clearance
shmdt(sleeptimer_1);
shmdt(status_1);
shmdt(column_1);
if(shmctl(shmid1, IPC_RMID ,NULL) == -1){
    fputs("deleting status shm",testing);
    exit(1);
}

if(shmctl(shmid2, IPC_RMID ,NULL) == -1){
    fputs("deleting sleeptimer shm",testing);
    exit(1);
}

if(shmctl(shmid3, IPC_RMID ,NULL) == -1){
    fputs("deleting column shm",testing);
    exit(1);
}
```

Parent Process

Having only the child memory attached to the shared memory segments is not enough. The parent process had also to be attached. Using the same key identifiers as the child process, the parent process, would try to find such memory segments using system call shmget(), however this time with a different protocol, since the memory segment already existed and we were looking for it.

Once the shared memory segments were found, the parent also had to attach to them and have a pointer point to them. This enabled the parent process to write in them the necessary variables.

The first variable to write was that of the sleep timer variable invoked in the set refresh command.

```
else if ((strcmp(setcommand, "refresh")) == 0) {
  sleeper[0]=atoi(setinput);
```

The second one was the interrupting value of 1 which would be sent to the child process as soon as the user wanted to exit from the shell. Note that besides sending the interrupting value of 1, the exit command would also detach the pointers from the shared memory segments, in order to enable the child process to delete them. This was of utmost importance since only the process that created them can delete them, and this cannot be done if some other process is still attached to them.

```
else if (strcmp(command, "exit") == 0) {
  closeShell++;

  status[0] = 1;
  fputs("Consumer Ended\n",testing);
  shmdt(sleeptimer_1);
  shmdt(status_1);
  shmdt(column_1);
```

Note that the code used to find the memory segment and attach to it will not be included here since it is similar to the once used in the child process, however this can be found in the following full code section.