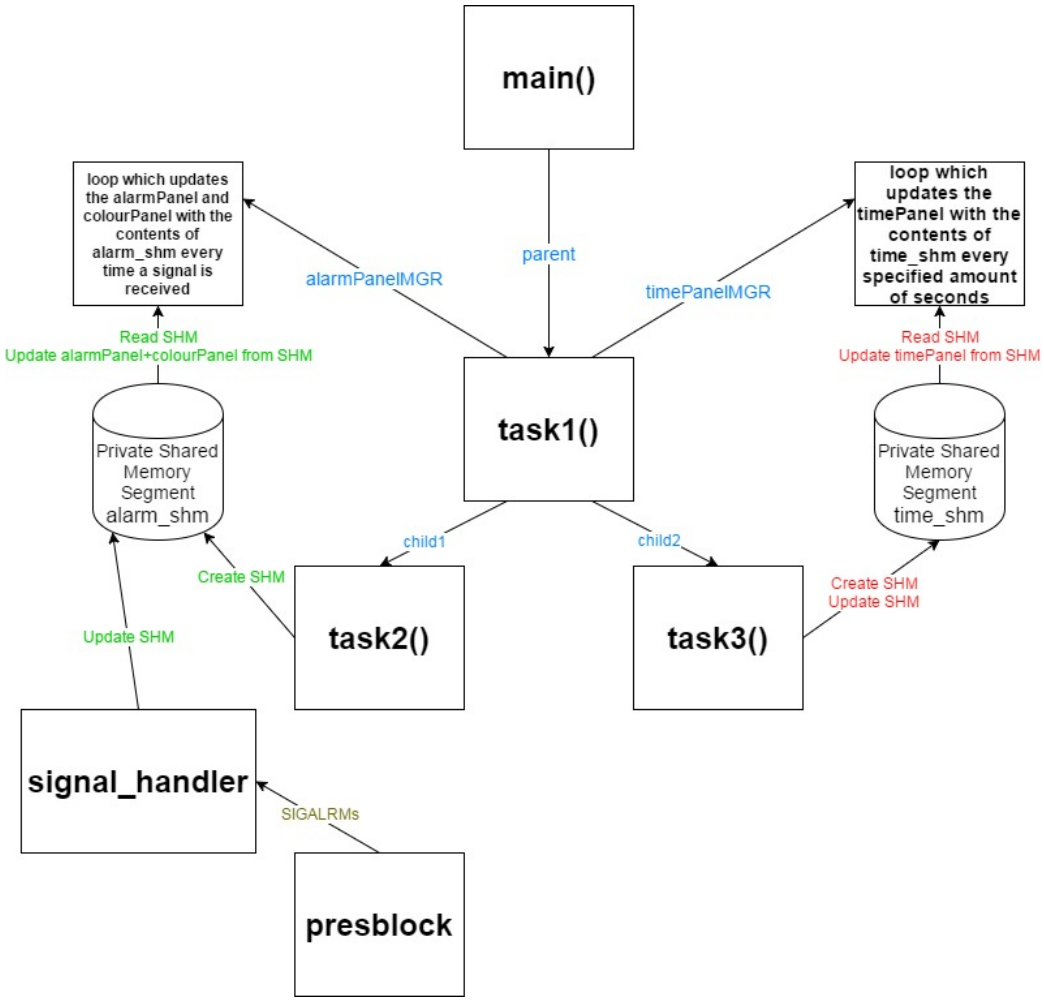
CPS1012 - Assignment

Daniel Magro
484497M

May 2017

Structure and Explanation



The above diagram is a high-level view of all the different methods, processes and shared memory segments.

The words in **blue** represent all the different processes that are working together concurrently. When the program is launched, it forks into the **parent** process which runs task1(), and two child processes, **child1** and **child2**, which run task2() and task3() respectively.

The words in **green** represent the interactions that occur with the alarm Shared Memory Segment. Similarly, the words in **red** represent the interactions that occur with the time Shared Memory Segment.

The word in **yellow** represents the signals being sent from the presblock daemon to the signal_handler.

The shared memory segments are declared as structs of the following type:

alarm_shm is declared as the **alarmInfo** struct. This is made up of:
int colour - int storing the colour of the colour panel
char Message [32] - string (array of chars) storing the time at which the notification was received
int alarmLC - The line counter which indicates at which line (y co-ordinate) in the alarm panel the notification will be printed

time_shm is declared as the **timeInfo** struct. This includes 3 strings (character arrays), each of which will store the data to be printed for each time zone's clock

When the program is launched, after all the libraries are included and global variables declared, the mutex lock **printLock** is initialised and the method **signal_handler** is declared to handle SIGALRMs. The process ID is also displayed for 5 seconds so that the user can input that to the presblock daemon. Once this is all done, the program forks into 2 children, whereby the **parent** will mainly handle **task1** and **ncurses**, **child1** will handle **task2()** and **child2** will handle **task3()**.

task1() first declares all the windows (panels) which will be used to form the basic interface and initialise them as subwindows of the mainpanel, giving them a size and position. The size of the main window will fill up the size of the terminal (it is dynamic) and all subwindow sizes and positions are calculated as percentages of those dimensions.
Next it attaches to the **alarm_shm** (alarm shared memory segment) and **time_shm** (time shared memory segment).
Colours are then started, the RGB definition of Yellow is changed to the values of Orange, and each colour pair is initialised.

Another fork is executed, **alarmPanelMGR**, which first sets the LineCounter inside the shm to 1. The following code is looped until runLoop is set to zero (on exit). The child process: sleeps for a second (to reduce constant CPU load), locks the mutex lock (to avoid its cursor being moved), prints an Alarm Received notification inside the alarm panel along with the time at which the alarm was received (from the shm), changes the colour of the colour panel (according to what is stored inside the shm), prints an Alarm Handled notification, refreshes the alarm and colour panels and unlocks the mutex lock.

Another fork is executed, **timePanelMGR**, which operates similarly to the alarmPanelMGR. The following code is looped until runLoop is set to zero on exit. The child process: sleeps for the specified refresh time (internal shell variable), locks the mutex lock (to avoid its cursor being moved), and prints all the different timezones on 3 different lines from the shm, refreshes the time panel, and unlocks the mutex lock.

Next, the user input is obtained by accepting his input character by character, using the 'getch' function, and then printing (echoing) every character as it is inputted, using 'mvwaddch'. Backspace is handled by deleting the last inputted letter from the array storing the string, and clearing that letter from the prompt panel. Letters are accepted until the user presses 'Enter'.

Then the input is divided into command and argument using the 'sscanf' function.

The command to be executed is chosen based on a long if, else if, else if, ... where every condition is an 'strcmp'. The mutex lock is locked before beginning this if - else if tree and unlocked at the end, since outputs to the output panel occur for every command. At the end, the output panel is refreshed and the Line Counters are adjusted accordingly. This repeats itself until the user inputs "exit", at which point the loop is exitted, the file pointer is closed, the shared memory segments detached, and the 2 child processes (alarmPanelMGR and timePanelMGR) are killed.

In **task2**, the Private Shared Memory Segment (alarm_shm) is created. Then the 'timespec' **time2** is set to the current time (CLOCK_MONOTONIC since this clock will be used to calculate time intervals) and a loop takes place which pauses execution of this method/child process in order not to Destroy the Shared Memory Segment, up until runLoop is set to zero.

The **signal_handler** method handles all the SIGALRMs that are received. First it attaches to the Private Shared Memory Segment, alarm_shm. Then it checks if the signal received was a SIGALRM (by design of the presblock, this step may be considered redundant as the signal sent is always a SIGALRM), if so the following code is executed: time1 is set to time2, the value of time2 is updated with the current time from the monotonic clock, the timeDiff (interarrival time) is calculated by subtracting time1 from time2 (in seconds) and, based on the interarrival time, the colourpair of the colour panel is decided and stored inside alarm_shm. The current time is also stored inside alarm_shm so that it can be displayed alongside the "Alarm Received" notification, in a specific format using 'strftime'. The line counter of the alarm panel is also updated and stored in the shm. Finally the signal handler detaches from the shm.

In **task3**, the Private Shared Memory Segment (time_shm) is created. Three timeval structs are declared, one for each time zone. An loop is then started which runs every specified number of seconds (due to the 'sleep' for 'refreshTime' function) until runLoop is set to zero. The following is repeated for every time zone:
First 'gettimeofday' is run, to get the current epoch time (in seconds), the time zone is adjusted by adding/subtracting the number of hours * (60 min-

utes * 60 seconds) and the time is printed inside the time_shm using 'sprintf' and 'ctime' in order to format the epoch time as readable time. When the loop exits, the shared memory segment is destroyed.

Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     // for strcmp, strlen, strcpy, strcat, ...

// Imports for Forks
#include <unistd.h>
#include <signal.h>     // for kill (killing child processes)

// Imports for Shared Memory Segment
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

// Imports for ncurses functionality
#include <ncurses.h>
#include <sys/ioctl.h>  // for max window size
#include <pthread.h>    // for mutex locks

// Imports for Alarm and Time Panel
#include <time.h>
#include <sys/time.h>

int task1();
int task2(); void signal_handler(int sig);
int task3();

// Shared Memory Segment structs:
struct alarmInfo{
    int colour;
    char message[32];
    int alarmLC;
};

struct timeZones{
    char USAtime[64];
    char MALTAtime[64];
```

```
37      char TOKYOtime[64];
38  };
39
40  // Global Variables:
41
42  // Internal Shell variable, time between every Time Panel
        refresh
43  unsigned int refreshTime = 1;
44
45  // Two timespec structs which are used to calculate the
        difference between 2 consecutive SIGALRMs
46  struct timespec time1;
47  struct timespec time2;
48
49  // The y-size (height) of the Alarm Panel
50  int alarmY;
51
52  // Boolean value (stored as int) which terminates all
        infinite loops when exiting the program, thus allowing
53  // the methods to complete the clean up tasks after the end
        of their loops
54  int runLoop = 1;
55
56  // Declaring the printLock Mutex Lock as a global variable
57  pthread_mutex_t printLock;
58
59  int main(void){
60      pthread_mutex_init(&printLock, NULL);
61
62      // Declaring the signal handler that will be receiving &
        handling SIGALRM calls from the presblock daemon
63      signal(SIGALRM, (__sighandler_t) signal_handler);
64      // Displaying the Process ID that presblock needs to be
        provided
65      printf("Please enter this PID inside presblock: %d",
        getpid());
66      fflush(stdout);
67      sleep(5);
68
69      // Forking the Parent Process into 2 child processes
70      pid_t child1, child2;
71
72      if (!(child1 = fork())) {
73          // first child - Alarm Panel
74          task2();
75          _exit(0);
76      } else if (!(child2 = fork())) {
77          // second child - Time Panel
78          task3();
```

```
79          _exit(0);
80      } else {
81          // parent
82          task1();
83      }
84
85      // By setting runLoop to 0, all 'infinite' loops will
    terminate
86      runLoop = 0;
87      // Give the methods some time to execute the code after
    the 'infinite' loops (clean up) before killing the
    processes
88      sleep(2);
89
90      // Killing off the child processes - (SIGTERM can be used
     to let processes clean up)
91      kill(child1, SIGKILL);
92      kill(child2, SIGKILL);
93
94      return 0;
95  }
96
97  int task1(){
98      WINDOW * mainwin;
99      WINDOW * promptPanel, * outputPanel;
100     WINDOW * alarmPanel, * colourPanel;
101     WINDOW * timePanel;
102
103     // Getting maximum dimensions of terminal
104     //getmaxyx(mainwin, mainwinY, mainwinX);
105     struct winsize mainwinSize;
106     ioctl(STDOUT_FILENO, TIOCGWINSZ, &mainwinSize);
107
108     // Setting sizes of every window
109     int mainwinY = mainwinSize.ws_row, mainwinX = mainwinSize
    .ws_col;
110
111     int promptY  = mainwinY/4, promptX = mainwinX;      // Y
    =24 or 43 - X=80 or 143 (windowed 24*80)(Fullscreen
    43*143)
112     int outputY = mainwinY/2, outputX = mainwinX;
113     //int outputY = 80, outputX = 256;
114
115     alarmY = mainwinY/4; int alarmX = mainwinX*3/8;
116     int colourY = mainwinY/4, colourX = mainwinX/8;
117
118     int timeY = mainwinY/4, timeX = mainwinX/2;
119
120     // Initialize ncurses
```

```
121     if ( (mainwin = initscr()) == NULL ) {
122         fprintf(stderr, "Error initialising ncurses.\n");
123         exit(EXIT_FAILURE);
124     }
125 
126     // Switch off echoing
127     //noecho();
128     // Switch off cursor
129     curs_set(0);
130 
131     // subwin(WINDOW, sizeY, sizeX, locationY, locationX);
132     // Initializing the prompt panel
133     promptPanel = subwin(mainwin, promptY, promptX, (mainwinY
    *3/4), 0);
134     // add a border to the prompt panel
135     box(promptPanel, 0, 0);
136     wrefresh(promptPanel);
137     // Initializing the output panel
138     outputPanel = subwin(mainwin, outputY, outputX, (mainwinY
    *1/4), 0);
139     box(outputPanel, 0, 0);
140     wrefresh(outputPanel);
141 
142     // Initializing the alarm panel
143     alarmPanel = subwin(mainwin, alarmY, alarmX, 0, timeX);
144     box(alarmPanel, 0, 0);
145     wrefresh(alarmPanel);
146     // Initializing the colour panel
147     colourPanel = subwin(mainwin, colourY, colourX, 0, (timeX
    +alarmX));
148     box(colourPanel, 0, 0);
149     wrefresh(colourPanel);
150 
151     // Initializing the time panel
152     timePanel = subwin(mainwin, timeY, timeX, 0, 0);
153     box(timePanel, 0, 0);
154     wrefresh(timePanel);
155 
156 
157 
158     // Private Shared Memory Segments
159 
160     // Alarm Panel Private Shared Memory Segment identifier
161     key_t alarmKey = 0x0001;
162     size_t alarmSize = sizeof(struct alarmInfo);   // alarmY*
    alarmX (+colourY)
163     int alarm_shmid = shmget(alarmKey, alarmSize, 0666);
164     if (alarm_shmid < 0) {
165         perror("shmget");
```

```
166         exit(1);
167     }
168     // Attaching to the Private Shared Memory Segment which
    is storing the Alarm Panel
169     struct alarmInfo * alarm_shm = (struct alarmInfo *) shmat
    (alarm_shmid, NULL, 0);
170     if (alarm_shm == (struct alarmInfo *) -1) {
171         perror("shmat");
172         exit(1);
173     }
174
175     // Time Panel Private Shared Memory Segment identifier
176     key_t timeKey = 0x0002;
177     size_t timeSize = sizeof(struct timeZones);   // timeY*
    timeX
178     int time_shmid = shmget(timeKey, timeSize, 0666);
179     if (time_shmid < 0) {
180         perror("shmget");
181         exit(1);
182     }
183     // Attaching to the Private Shared Memory Segment which
    is storing the Time Panel
184     struct timeZones * time_shm = (struct timeZones *) shmat(
    time_shmid, NULL, 0);
185     if (time_shm == (struct timeZones *) -1) {
186         perror("shmat");
187         exit(1);
188     }
189
190
191     // Starting Colours in ncurses
192     start_color();
193
194     // Change the RGB values of the colour YELLOW to those of
     the colour orange
195     init_color(COLOR_YELLOW, 1000, 647, 0);
196
197     // Defining the colour pairs which will be used for the
    alarm colour bar
198     init_pair(1, COLOR_BLACK, COLOR_WHITE);
199     init_pair(2, COLOR_BLACK, COLOR_RED);
200     init_pair(3, COLOR_BLACK, COLOR_YELLOW);
201     init_pair(4, COLOR_BLACK, COLOR_GREEN);
202     init_pair(5, COLOR_BLACK, COLOR_BLUE);
203
204     // Alarm Panel Updater - Reads from Alarm Shared Memory
    Segment and outputs to Alarm Panel
205     pid_t alarmPanelMGR = fork();
206     if (alarmPanelMGR < 0){
```

```
207         mvwprintw(alarmPanel, 1, 1, "Fork Failed");
208     } else if (alarmPanelMGR == 0){
209         alarm_shm->alarmLC = 1;
210         while(runLoop == 1) {
211             sleep(1);
212 
213             pthread_mutex_lock(&printLock);
214             // Printing the time from the shared memory
    segment + Alarm Received
215             mvwprintw(alarmPanel, alarm_shm->alarmLC, 1, "[%s
    ] Alarm Received",alarm_shm->message);
216             // Changing the colour of the alarm panel
217             wbkgd(colourPanel, COLOR_PAIR(alarm_shm->colour))
    ;
218             // Printing that the alarm has been handled
219             mvwprintw(alarmPanel, (alarm_shm->alarmLC + 1),
    1, "[%s] Alarm Handled",alarm_shm->message);
220             wrefresh(colourPanel);
221             wrefresh(alarmPanel);
222             pthread_mutex_unlock(&printLock);
223         }
224     }
225 
226 
227     // Time Panel Updater - Reads from Time Shared Memory
    Segment and outputs to Time Panel
228     pid_t timePanelMGR = fork();
229     if (timePanelMGR < 0){
230         mvwprintw(timePanel, 1, 1, "Fork Failed");
231     } else if (timePanelMGR == 0){
232         while(runLoop == 1) {
233             sleep(refreshTime);
234 
235             pthread_mutex_lock(&printLock);
236             mvwprintw(timePanel, 1, 1, time_shm->USAtime);
237             mvwprintw(timePanel, 2, 1, time_shm->MALTAtime);
238             mvwprintw(timePanel, 3, 1, time_shm->TOKYOtime);
239             wrefresh(timePanel);
240             pthread_mutex_unlock(&printLock);
241         }
242     }
243 
244 
245     // Char which stores the character inputted by the user
246     char inputChar;
247     // Array of characters which will store the command
    entered by the user
248     char command[20];
249     // Array of characters which will store the argument
```

```
    entered by the user
250  char argument[200];
251
252  // internal shell variables
253  char prompt[32];
254  char path[256];
255  //unsigned int refreshTime;
256  char buffer[16];
257  int buffery;
258  int bufferx;
259
260  // default values of shell internal variables (set)
261  strcpy(prompt,"OK");
262  //refreshTime = 1;
263  strcpy(buffer, "80x256");
264  sscanf(buffer, "%dx%d",&buffery,&bufferx);  // buffery
    =80;bufferx=256;
265
266  // accessing a file to store output in
267  FILE * outputFP;
268  outputFP = fopen("output", "w");
269
270  FILE * systemFP;
271
272  // counter which stores which line the program is on in
    the Prompt Panel(for cursor)
273  int promptLC = 1;
274  int outputLC = 1;
275  int i, j;   // counters
276  char var[10];   // will store which internal variable
    will be changed
277  char temp[256];   // used as a temporary character array
278  do{
279      // Outputting the prompt (eg: OK>)
280      pthread_mutex_lock(&printLock);
281      // clear the line you will start writing to
282      wmove(promptPanel, promptLC, 1); wclrtoeol(
    promptPanel); box(promptPanel, 0, 0);
283      mvwprintw(promptPanel, promptLC, 1, "%s>",prompt);
284      wrefresh(promptPanel);
285      pthread_mutex_unlock(&printLock);
286      //wnoutrefresh(promptPanel);
287      //doupdate();
288
289      // Getting user input
290      pthread_mutex_lock(&printLock);
291      //mvwscanw(promptPanel, promptLC, (int) strlen(prompt
    )+2, "%s %180[^\n]s",command,argument);
292      //pthread_mutex_unlock(&printLock);
```

```
293         i=0;
294         // Getting input character by character
295         do{
296             // Get the user inputted character and store it
297             inputChar = (char) getch();
298             if (inputChar == 127 && i != 0) {   //
    KEY_BACKSPACE
299                 i--;
300                 temp[i] = '\0';
301                 wmove(promptPanel, promptLC, (int) (strlen(
    prompt)+2+i+1)); wclrtoeol(promptPanel); box(promptPanel,
    0, 0);
302                 wrefresh(promptPanel);
303             } else if (inputChar == 13){
304                 temp[i] = '\0';
305                 break;
306             } else {
307                 temp[i] = inputChar;
308                 i++;
309                 // echo the user's input in the prompt panel
310                 mvwaddch(promptPanel, promptLC, (int) (strlen
    (prompt)+2+i), inputChar);
311                 wrefresh(promptPanel);
312             }
313         } while(inputChar != '\n'); // do this until the user
     presses 'Enter'
314         temp[i] = '\0';
315         sscanf(temp, "%s %180[^\n]s",command,argument);
316         wrefresh(promptPanel);
317         pthread_mutex_unlock(&printLock);
318
319
320
321         // Handling the user's chosen command
322         pthread_mutex_lock(&printLock);
323         if (strcmp(command, "chdir") == 0) {
324             strcpy(temp, getcwd(0,0));
325             chdir(argument);
326             mvwprintw(outputPanel, outputLC, 1, "Directory
    changed from: %s to: %s",temp,getcwd(0,0));
327             fprintf(outputFP, "Directory changed from: %s to:
     %s\n",temp,getcwd(0,0));
328         } else if (strcmp(command, "shdir") == 0){
329             mvwprintw(outputPanel, outputLC, 1, "Current
    Directory: %s",getcwd(0,0));
330             fprintf(outputFP, "Current Directory: %s\n",
    getcwd(0,0));
331         } else if (strcmp(command, "print") == 0){
332             //mvwaddstr(outputPanel, outputLC, 1, argument);
```

```
333             mvwprintw(outputPanel, outputLC, 1, "%s",argument
    );
334             fprintf(outputFP, "%s\n",argument);
335         } else if (strcmp(command, "printvar") == 0){
336             if (strcmp(argument, "prompt") == 0){
337                 mvwprintw(outputPanel, outputLC, 1, "prompt:
    %s",prompt);
338                 fprintf(outputFP, "prompt: %s\n",prompt);
339             } else if (strcmp(argument, "path") == 0){
340                 mvwprintw(outputPanel, outputLC, 1, "path: %s
    ",path);
341                 fprintf(outputFP, "path: %s\n",path);
342             } else if (strcmp(argument, "refresh") == 0){
343                 mvwprintw(outputPanel, outputLC, 1, "refresh:
     %u",refreshTime);
344                 fprintf(outputFP, "refresh: %u\n",refreshTime
    );
345             } else if (strcmp(argument, "buffer") == 0){
346                 mvwprintw(outputPanel, outputLC, 1, "buffer:
    %dx%d",buffery,bufferx);
347                 fprintf(outputFP, "buffer: %dx%d\n",buffery,
    bufferx);
348             }
349         } else if (strcmp(command, "set") == 0){
350             // finding out which variable will be set and
    what value it will be set to
351             i = 0, j = 0;
352             while(argument[i] != '=') {
353                 var[j] = argument[i];
354                 i++;
355                 j++;
356             }
357             var[j] = '\0';
358             i++;    // for = sign
359             j = 0;
360             while(argument[i] != '\0') {
361                 temp[j] = argument[i];
362                 i++;
363                 j++;
364             }
365             temp[j] = '\0';
366 
367             //sscanf(argument, "%16[^=]s=%180[^\n]s",var,temp
    );
368 
369             // var now holds the variable to be set
370             // temp holds the value of the variable
371             if (strcmp(var, "prompt") == 0){
372                 strcpy(prompt, temp);
```

```
373                 mvwprintw(outputPanel, outputLC, 1, "prompt
    was set to: %s",prompt);
374                 fprintf(outputFP, "prompt was set to: %s\n",
    prompt);
375             } else if (strcmp(var, "path") == 0){
376                 strcpy(path, temp);
377                 mvwprintw(outputPanel, outputLC, 1, "path was
     set to: %s",path);
378                 fprintf(outputFP, "path was set to: %s\n",
    path);
379             } else if (strcmp(var, "refresh") == 0){
380                 refreshTime = atoi(temp);
381                 mvwprintw(outputPanel, outputLC, 1, "refresh
    was set to: %u",refreshTime);
382                 fprintf(outputFP, "refresh was set to: %u\n",
    refreshTime);
383             } else if (strcmp(var, "buffer") == 0){
384                 sscanf(temp, "%dx%d",&buffery,&bufferx);
385                 wresize(outputPanel, buffery, bufferx);
    wrefresh(outputPanel);
386                 mvwprintw(outputPanel, outputLC, 1, "buffer
    was set to: %dx%d",buffery,bufferx);
387                 fprintf(outputFP, "buffer was set to: %dx%d\n
    ",buffery,bufferx);
388             }
389         } else if (strcmp(command, "move") == 0){
390             mvwprintw(outputPanel, outputLC, 1, "Window was
    moved by %d",atoi(argument));
391             fprintf(outputFP, "Window was moved by %d\n",atoi
    (argument));
392         } else if (strcmp(command, "exit") == 0){
393             mvwprintw(outputPanel, outputLC, 1, "Orange Wave
    will now exit");
394             fprintf(outputFP, "Orange Wave will now exit");
395         } else{     // external command
396             strcpy(temp, command); strcat(temp, " "); strcat(
    temp, argument);
397             mvwprintw(outputPanel, outputLC, 1, "%s was not
    found as a built-in function, trying to run as an external
     command",temp);
398             fprintf(outputFP, "%s was not found as a built-in
     function, trying to run as an external command\n",temp);
399             // redirecting the output of the system call to
    the tempOut File
400             strcat(temp, " > tempOut");
401             // emptying tempOut File
402             systemFP = fopen("tempOut", "w");
403             fclose(systemFP);
404             // Making the system call with the user's command
```

```
 + output redirection
405             system(temp);
406             // reading from the text file and outputting the
    result
407             systemFP = fopen("tempOut", "r");
408             strcpy(temp, "");
409             fgets(temp, 250, systemFP);
410             fclose(systemFP);
411             mvwprintw(outputPanel, ++outputLC, 1, "%s",temp);
412             fprintf(outputFP, "%s\n",temp);
413         }
414         wrefresh(outputPanel);
415         pthread_mutex_unlock(&printLock);
416 
417         // if the Line Counter for the Prompt Panel has
    reached the end, then start from the beginning/top again
418         if(promptLC < (promptY-2)){
419             promptLC++;
420         } else{
421             promptLC = 1;
422         }
423 
424         if(outputLC < (outputY-2)){
425             outputLC++;
426         } else{
427             outputLC = 1;
428         }
429 
430     }while(strcmp(command, "exit") != 0);   // ==0 means they
    are equal, so != will loop until exit is entered
431 
432     // Clean up after ourselves
433     delwin(promptPanel);
434     delwin(outputPanel);
435     delwin(alarmPanel);
436     delwin(colourPanel);
437     delwin(timePanel);
438     delwin(mainwin);
439     endwin();
440     refresh();
441 
442     // Close the File
443     fclose(outputFP);
444 
445     // Detach the Shared Memory segments
446     shmdt(alarm_shm);
447     shmdt(time_shm);
448 
449     // killing off the child processes
```

```
450     kill(alarmPanelMGR, SIGKILL);
451     kill(timePanelMGR, SIGKILL);
452 
453     return 0;
454 }
455 
456 int task2(){
457 
458     // Shared memory segment
459 
460     // Alarm Panel Private Shared Memory Segment identifier
461     key_t alarmKey = 0x0001;
462     size_t alarmSize = sizeof(struct alarmInfo);   //alarmY*
    alarmX
463     // Create the Private Shared Memory Segment
464     int alarm_shmid = shmget(alarmKey, alarmSize, IPC_CREAT |
    0666);
465     if (alarm_shmid < 0) {
466         perror("shmget");
467         exit(1);
468     }
469     // Attach to the segment
470     struct alarmInfo * alarm_shm = (struct alarmInfo *) shmat
    (alarm_shmid, NULL, 0);
471     if (alarm_shm == (struct alarmInfo *) -1) {
472         perror("shmat");
473         exit(1);
474     }
475 
476     // Set the current time for the presblock
477     clock_gettime(CLOCK_MONOTONIC, &time2);
478 
479     // Does not allow the Shared Memory Segment to be
    destroyed until the program is ready to exit
480     while(runLoop == 1){
481         pause();
482     }
483 
484     // Destroying the Shared Memory Segment when we are done
485     if(shmctl(alarm_shmid, IPC_RMID ,NULL) == -1) {
486         perror("shmctl");
487         exit(1);
488     }
489 
490     return 0;
491 }
492 // Signal Handling method for the presblock daemon
493 void signal_handler(int sig){
494 
```

```
495     // Attaching to the Alarm Shared Memory Segment
496     key_t alarmKey = 0x0001;
497     size_t alarmSize = sizeof(struct alarmInfo);   // alarmY*
    alarmX (+colourY?)
498     int alarm_shmid = shmget(alarmKey, alarmSize, 0666);
499     if (alarm_shmid < 0) {
500         perror("shmget");
501         exit(1);
502     }
503     // Attaching to the Private Shared Memory Segment which
    is storing the Alarm Panel
504     struct alarmInfo * alarm_shm = (struct alarmInfo *) shmat
    (alarm_shmid, NULL, 0);
505     if (alarm_shm == (struct alarmInfo *) -1) {
506         perror("shmat");
507         exit(1);
508     }
509
510
511     if (sig == SIGALRM){
512         time1 = time2;
513         clock_gettime(CLOCK_MONOTONIC, &time2);
514
515         int timeDiff = time2.tv_sec - time1.tv_sec;
516
517         // Decide which colour pair to display based on the
    interarrival time and store it in the Shared Memory
    Segment
518         if (timeDiff < 5){
519             // white
520             alarm_shm->colour = 1;
521         } else if (timeDiff>=5 && timeDiff<10){
522             // red
523             alarm_shm->colour = 2;
524         } else if (timeDiff>=10 && timeDiff<15){
525             // orange
526             alarm_shm->colour = 3;
527         } else if (timeDiff>=15 && timeDiff<=20){
528             // green
529             alarm_shm->colour = 4;
530         } else {
531             // blue
532             alarm_shm->colour = 5;
533         }
534
535         // Store the time at which the alarm was received in
    the Shared Memory Segment
536         strftime(alarm_shm->message, 31, "%H:%M:%S", gmtime(&
    time2.tv_sec));
```

```
537
538         // Change the y-coordinate at which the alarm prompts
        will be printed inside tha alarm panel
539         if(alarm_shm->alarmLC < (alarmY-4)){
540             alarm_shm->alarmLC += 2;
541         } else{
542             alarm_shm->alarmLC = 1;
543         }
544
545     } else {
546         perror("Unexpected Signal Received");
547     }
548
549     // Detach from the shared memory segment
550     shmdt(alarm_shm);
551 }
552
553 int task3(){
554     // Shared memory segment
555
556     // Time Panel Private Shared Memory Segment identifier
557     key_t timeKey = 0x0002;
558     size_t timeSize = sizeof(struct timeZones);   //timeY*
        timeX
559     // Create the Private Shared Memory Segment
560     int time_shmid = shmget(timeKey, timeSize, IPC_CREAT |
        0666);
561     if (time_shmid < 0) {
562         perror("shmget");
563         exit(1);
564     }
565     // Attach to the segment
566     struct timeZones * time_shm = (struct timeZones *) shmat(
        time_shmid, NULL, 0);
567     if (time_shm == (struct timeZones *) -1) {
568         perror("shmat");
569         exit(1);
570     }
571
572     struct timeval USAtime;
573     struct timeval MALTAtime;
574     struct timeval TOKYOtime;
575
576     // while global is not 1
577     while (runLoop == 1) {
578         sleep(refreshTime);
579
580         // Getting the current epoch time storing it in
        USAtime
```

```
581         gettimeofday(&USAtime, NULL);
582         // Reducing 6 Hours worth of seconds from the epoch
    time to adjust for the Time Zone
583         USAtime.tv_sec -= 6*(60*60);
584         // Storing the formatted time in the Shared Memory
    Segment
585         sprintf(time_shm->USAtime, "WHITE HOUSE [USA]: %s",
    ctime((const time_t *) &USAtime.tv_sec));
586
587         gettimeofday(&MALTAtime, NULL);
588         MALTAtime.tv_sec += 1*(60*60);
589         sprintf(time_shm->MALTAtime, "MALTA [MSIDA]: %s",
    ctime((const time_t *) &MALTAtime.tv_sec));
590
591         gettimeofday(&TOKYOtime, NULL);
592         TOKYOtime.tv_sec += 9*(60*60);
593         sprintf(time_shm->TOKYOtime, "JAPAN [TOKYO]: %s",
    ctime((const time_t *) &TOKYOtime.tv_sec));
594     }
595
596     // Destroying the Shared Memory Segment when we are done
597     if(shmctl(time_shmid, IPC_RMID ,NULL) == -1) {
598         perror("shmctl");
599         exit(1);
600     }
601
602     return 0;
603 }
```