Table of Contents

Source Code

lift_sim_A.c

```
1. /* File: lift_sim_A.c
2.  * Author: Adil Khokhar (19182405)
3.  * Purpose: Simulate lift requests using Posix Threads
4.  * Date Modified: 16/05/2020
5.  */
6.
7. #include <stdio.h>
8. #include <stdlib.h>
9. #include <pthread.h>
10. #include <string.h>
11. #include <unistd.h>
12.
13. #include "lift_sim_A.h"
14. #include "structs.h"
15. #include "fileIO.h"
16.
17. pthread_mutex_t lock;
18. pthread_cond_t full;
19. pthread_cond_t empty;
20.
21. buffer liftRequests[10];
22. lifts liftArray[3];
23.
24. int bufferSize;
25. int sleepTime;
26. int finished;
27. int finishLift;
28. int in;
29. int out;
30. int isFull;
31. int isEmpty;
32.
33. int requestNo;
34.
35. int main(int argc, char *argv[])
36. {
37.     pthread_t lift_R;
38.     pthread_t lift_1;
39.     pthread_t lift_2;
40.     pthread_t lift_3;
41.
42.     /* Allocate some memory to int pointers to pass to threads */
43.     int *arg1 = (int*)malloc(sizeof(int));
44.     int *arg2 = (int*)malloc(sizeof(int));
```

```
45.     int *arg3 = (int*)malloc(sizeof(int));
46.
47.     if( argc == 3 )
48.     {
49.         if(atoi(argv[1]) < 1)
50.         {
51.             printf("Error! Buffer Size has to be greater than or equal to 1!\n");
52.         }
53.
54.         else if(atoi(argv[2]) < 0)
55.         {
56.             printf("Error! Time has to be greater than or equal to 0!\n");
57.         }
58.
59.         else
60.         {
61.             bufferSize = atoi(argv[1]) + 1;
62.             sleepTime = atoi(argv[2]);
63.
64.             /* Run function initalise (lift_sim_A.c) */
65.             initialise();
66.             /* Run function openFiles (fileIO.c) */
67.             openFiles();
68.
69.             /* Create POSIX threads and pass specific indexes (tells lift function)
70.                which lift is running */
71.             if(pthread_create(&lift_R, NULL, request, NULL) == -1)
72.             {
73.                 printf("Can't create Lift R\n");
74.             }
75.
76.             else
77.             {
78.                 printf("Lift-R Thread Created and Running ...\n");
79.             }
80.
81.             *arg1 = 0;
82.             if(pthread_create(&lift_1, NULL, lift, arg1) == -1)
83.             {
84.                 printf("Can't create Lift 1\n");
85.             }
86.
87.             else
88.             {
89.                 printf("Lift-1 Thread Created and Running ...\n");
90.             }
91.
92.             *arg2 = 1;
```

```
93.             if(pthread_create(&lift_2, NULL, lift, arg2) == -1)
94.             {
95.                 printf("Can't create Lift 2\n");
96.             }
97.
98.             else
99.             {
100.                    printf("Lift-2 Thread Created and Running ...\n");
101.                }
102.
103.                *arg3 = 2;
104.                if(pthread_create(&lift_3, NULL, lift, arg3) == -1)
105.                {
106.                    printf("Can't create Lift 3\n");
107.                }
108.
109.                else
110.                {
111.                    printf("Lift-3 Thread Created and Running ...\n");
112.                }
113.
114.                /* Wait for the threads to terminate */
115.                pthread_join(lift_R,NULL);
116.                pthread_join(lift_1,NULL);
117.                pthread_join(lift_2,NULL);
118.                pthread_join(lift_3,NULL);
119.
120.                /* Destroy mutexes and condition variables */
121.                pthread_mutex_destroy(&lock);
122.                pthread_cond_destroy(&full);
123.                pthread_cond_destroy(&empty);
124.
125.                /* Run function writeResult (fileIO.c) */
126.
    writeResult((liftArray[0].totalMovement+liftArray[1].totalMovement+liftArray[2].tot
    alMovement), requestNo);
127.                /* Run function closeFiles (fileIO.c) */
128.                closeFiles();
129.            }
130.        }
131.
132.        else
133.        {
134.            printf("Error! Incorrect Number of Arguments!\n");
135.        }
136.
137.        /* Free dynamically allocated memory */
138.        free(arg1);
```

```
139.          free(arg2);
140.          free(arg3);
141.
142.          return 0;
143.      }
144.
145.      /*
146.       * Function: initalise
147.       * Purpose: Initalise some specific variables for beginning of lift
    simulator
148.       */
149.      void initialise()
150.      {
151.          int jj;
152.
153.          /* Initialise mutex lock and condition variables */
154.          if (pthread_mutex_init(&lock, NULL) != 0)
155.          {
156.              printf("\n mutex init has failed\n");
157.          }
158.
159.          if (pthread_cond_init(&full, NULL) != 0)
160.          {
161.              printf("\n full init has failed\n");
162.          }
163.
164.          if (pthread_cond_init(&empty, NULL) != 0)
165.          {
166.              printf("\n empty init has failed\n");
167.          }
168.
169.          /* Add Lift Names to the Structs */
170.          strcpy(liftArray[0].name, "Lift-1");
171.          strcpy(liftArray[1].name, "Lift-2");
172.          strcpy(liftArray[2].name, "Lift-3");
173.
174.          /* Set initial lift values */
175.          for(jj = 0; jj < 3; jj++)
176.          {
177.              liftArray[jj].prevRequest = 1;
178.              liftArray[jj].totalMovement = 0;
179.              liftArray[jj].totalRequests = 0;
180.          }
181.
182.          in = 0;
183.          out = 0;
184.          finished = 0;
185.          requestNo = 0;
```

```
186.          isFull = 0;
187.          isEmpty = 0;
188.          finishLift = 0;
189.      }
190.
191.      /*
192.       * Function: request
193.       * Purpose: Producer which adds lift requests to buffer
194.       */
195.      void *request(void *param)
196.      {
197.          int reading[2];
198.          int* readPointer;
199.
200.          while(finished == 0)
201.          {
202.              /* Lock Mutex */
203.              pthread_mutex_lock(&lock);
204.
205.              /* Check if mutex is currently full and wait if it is */
206.              while(((in+1)%bufferSize) == out)
207.              {
208.                  pthread_cond_wait(&full, &lock);
209.              }
210.
211.              /* Obtain next request from input file using readNextValue
    (fileIO.c) */
212.              readPointer = readNextValue(reading);
213.
214.              /* When requests are exhausted, readNextValue returns a value of 66,
215.                 which ordinarily is not a possible request */
216.              if(readPointer[0] == 66)
217.              {
218.                  finished = 1;
219.              }
220.
221.              else
222.              {
223.                  /* Add new request to buffer and change 'in' which points to
    current
224.                     request index (for FIFO queue) */
225.                  liftRequests[in].source = readPointer[0];
226.                  liftRequests[in].destination = readPointer[1];
227.                  requestNo++;
228.                  /* Use writeBuffer (fileIO.c) to write to output file */
229.                  writeBuffer(readPointer[0],readPointer[1], requestNo);
230.                  in = (in+1)%bufferSize;
231.              }
```

```
232.
233.             /* Signal that buffer is not empty and unlock mutex */
234.             pthread_cond_signal(&empty);
235.             pthread_mutex_unlock(&lock);
236.         }
237.
238.         printf("Lift-R Thread Finished ...\n");
239.
240.         return NULL;
241.     }
242.
243.     /*
244.      * Function: lift
245.      * Purpose: Consumer which serves lift request from buffer
246.      */
247.     void *lift(void *param)
248.     {
249.         /* i is used to obtain specific lift from lifts array */
250.         int i = *((int *) param);
251.
252.         while(finishLift == 0)
253.         {
254.             /* Lock Mutex */
255.             pthread_mutex_lock(&lock);
256.
257.             /* Check if buffer is empty and requests are still available, if so,
    wait */
258.             while((in == out) && (finished != 1))
259.             {
260.                 pthread_cond_wait(&empty, &lock);
261.             }
262.
263.             /* Run if buffer is not empty */
264.             if(in != out)
265.             {
266.                 /* Serve request from buffer */
267.                 liftArray[i].source = liftRequests[out].source;
268.                 liftArray[i].destination = liftRequests[out].destination;
269.                 liftArray[i].movement = abs(liftArray[i].prevRequest -
    liftArray[i].source) + abs(liftArray[i].destination - liftArray[i].source);
270.                 liftArray[i].totalMovement += liftArray[i].movement;
271.                 liftArray[i].totalRequests++;
272.
273.                 /* Use writeLift (fileIO.c) to write lift request to output file
    */
274.                 writeLift(&liftArray[i]);
275.
```

```
276.                 /* Make lifts previous request the destination that was just
    served */
277.                 liftArray[i].prevRequest = liftArray[i].destination;
278.
279.                 /* Change buffer index */
280.                 out = (out+1)%bufferSize;
281.
282.                 /* Signal that buffer is no longer full */
283.                 pthread_cond_signal(&full);
284.             }
285.
286.             /* Run if buffer is empty and if no more requests are available.
    This will
287.                break the while loop */
288.             if((in == out) && (finished == 1))
289.             {
290.                 finishLift = 1;
291.             }
292.
293.             /* Unlock the mutex */
294.             pthread_mutex_unlock(&lock);
295.
296.             /* Allow user define time to pass for lift request */
297.             sleep(sleepTime);
298.         }
299.
300.         printf("%s Thread Finished ...\n", liftArray[i].name);
301.
302.         return NULL;
303.     }
```

lift_sim_A.h

```
1. #ifndef LA_H
2. #define LA_H
3.
4. void initialise();
5. void *request(void *param);
6. void *lift(void *param);
7.
8. #endif
```

lift_sim_B.c

```
1. /* File: lift_sim_B.c
2.  * Author: Adil Khokhar (19182405)
3.  * Purpose: Simulate lift requests using Processes, POSIX Semaphores and POSIX
    Shared Memory
```

```
4.   * Date Modified: 16/05/2020
5.   */
6.
7.  #include <stdio.h>
8.  #include <stdlib.h>
9.  #include <string.h>
10. #include <fcntl.h>
11. #include <unistd.h>
12. #include <sys/shm.h>
13. #include <sys/wait.h>
14. #include <sys/stat.h>
15. #include <sys/types.h>
16. #include <sys/mman.h>
17. #include <semaphore.h>
18.
19. #include "lift_sim_B.h"
20. #include "structs.h"
21. #include "fileIO.h"
22.
23. /* Have to manually define ftruncate function otherwise doesn't work. */
24. int ftruncate(int f, off_t length);
25.
26. sem_t *mutex;
27. sem_t *empty;
28. sem_t *full;
29.
30. sem_t *fileOut;
31.
32. buffer* liftBuffer;
33. lifts* liftArray;
34. int* in;
35. int* out;
36.
37. int main(int argc, char *argv[])
38. {
39.     int bufferFd;
40.     int arrayFd;
41.     int inFd;
42.     int outFd;
43.     int jj;
44.     int bufferSize;
45.     int sleepTime;
46.     pid_t lift_R;
47.     pid_t lift_processes[3];
48.
49.     if(argc == 3)
50.     {
51.         if(atoi(argv[1]) < 1)
```

```
52.         {
53.             printf("Error! Buffer Size has to be greater than or equal to 1!\n");
54.         }
55.
56.         else if(atoi(argv[2]) < 0)
57.         {
58.             printf("Error! Time has to be greater than or equal to 0!\n");
59.         }
60.
61.         else
62.         {
63.             bufferSize = atoi(argv[1]) + 1;
64.             sleepTime = atoi(argv[2]);
65.
66.             /*
67.             * Create Shared Memory for Buffer, Lift Array,
68.             * and In and Out (Keep Track of Buffer Element)
69.             */
70.             bufferFd = shm_open("/liftbuffer", O_CREAT | O_RDWR, 0666);
71.             ftruncate(bufferFd, bufferSize*sizeof(buffer));
72.             liftBuffer = (buffer*)mmap(0, bufferSize*sizeof(buffer), PROT_READ |
    PROT_WRITE, MAP_SHARED, bufferFd, 0);
73.
74.             arrayFd = shm_open("/liftarray", O_CREAT | O_RDWR, 0666);
75.             ftruncate(arrayFd, 3*sizeof(lifts));
76.             liftArray = (lifts*)mmap(0, 3*sizeof(lifts), PROT_READ | PROT_WRITE,
    MAP_SHARED, arrayFd, 0);
77.
78.             inFd = shm_open("/in", O_CREAT | O_RDWR, 0666);
79.             ftruncate(inFd, sizeof(int));
80.             in = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
    inFd, 0);
81.
82.             outFd = shm_open("/out", O_CREAT | O_RDWR, 0666);
83.             ftruncate(outFd, sizeof(int));
84.             out = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
    outFd, 0);
85.
86.             /* Run function initalise (lift_sim_B.c) */
87.             initialise();
88.             /* Run function openFiles (fileIO.c) */
89.             openFiles();
90.
91.
92.             /* Create Buffer Producer Child Process */
93.             lift_R = fork();
94.
95.             if(lift_R == 0)
```

```
96.             {
97.                 printf("Lift-R Process Created and Running ...\n");
98.                 request(bufferSize);
99.                 exit(0);
100.                }
101.
102.                /* Create Consumer Child Processes */
103.                for(jj = 0; jj < 3; jj++)
104.                {
105.                    lift_processes[jj] = fork();
106.
107.                    if(lift_processes[jj] == 0)
108.                    {
109.                        printf("%s Process Created and Running ...\n",
    liftArray[jj].name);
110.                        lift(jj, bufferSize, sleepTime);
111.                        exit(0);
112.                    }
113.                }
114.
115.                /* Wait for all child processes to finish */
116.                waitpid(lift_R, NULL, 0);
117.
118.                for(jj = 0; jj<3; jj++)
119.                {
120.                    waitpid(lift_processes[jj], NULL, 0);
121.                }
122.
123.                /* Run function writeResult (fileIO.c) */
124.
    writeResult((liftArray[0].totalMovement+liftArray[1].totalMovement+liftArray[2].tot
    alMovement),
    (liftArray[0].totalRequests+liftArray[1].totalRequests+liftArray[2].totalRequests))
    ;
125.                /* Run function closeFiles (fileIO.c) */
126.                closeFiles();
127.
128.                /* Unmap and Close Shared Memory */
129.                munmap(liftBuffer, 3*sizeof(buffer));
130.                close(bufferFd);
131.
132.                munmap(liftArray, bufferSize*sizeof(lifts));
133.                close(arrayFd);
134.
135.                munmap(in, sizeof(int));
136.                close(inFd);
137.
138.                munmap(out, sizeof(int));
```

```
139.                 close(outFd);
140.
141.                 shm_unlink("/liftbuffer");
142.                 shm_unlink("/liftarray");
143.                 shm_unlink("/int");
144.                 shm_unlink("/out");
145.
146.                 /* Close Shared Semaphores */
147.                 sem_close(mutex);
148.                 sem_close(full);
149.                 sem_close(empty);
150.                 sem_close(fileOut);
151.             }
152.         }
153.
154.         else
155.         {
156.             printf("Error! Incorrect Number of Arguments!\n");
157.         }
158.
159.         return 0;
160.     }
161.
162.     /*
163.      * Function: initalise
164.      * Purpose: Initalise some specific variables for beginning of lift
    simulator
165.      */
166.     void initialise()
167.     {
168.         int jj;
169.
170.         /* Create Shared Named Semaphores */
171.         mutex = sem_open("/mutex", O_CREAT | O_EXCL, 0, 1);
172.         full = sem_open("/full", O_CREAT | O_EXCL, 0, 0);
173.         empty = sem_open("/empty", O_CREAT | O_EXCL, 0, 10);
174.         fileOut = sem_open("/fileOut", O_CREAT | O_EXCL, 0, 1);
175.
176.         /* Unlink Shared Named Semaphores */
177.         sem_unlink("/mutex");
178.         sem_unlink("/full");
179.         sem_unlink("/empty");
180.         sem_unlink("/fileOut");
181.
182.         /* Add Lift Names to the Structs */
183.         strcpy(liftArray[0].name, "Lift-1");
184.         strcpy(liftArray[1].name, "Lift-2");
185.         strcpy(liftArray[2].name, "Lift-3");
```

```
186.
187.          /* Set initial lift values */
188.          for(jj = 0; jj < 3; jj++)
189.          {
190.              liftArray[jj].prevRequest = 1;
191.              liftArray[jj].totalMovement = 0;
192.              liftArray[jj].totalRequests = 0;
193.              liftArray[jj].finished = 0;
194.          }
195.
196.          /* Set initial buffer position */
197.          *in = 0;
198.          *out = 0;
199.      }
200.
201.      /*
202.       * Function: request
203.       * Purpose: Producer which adds lift requests to a shared buffer
204.       */
205.      void request(int bufferSize)
206.      {
207.          int reading[2];
208.          int* readPointer;
209.          int finished;
210.          int requestNo;
211.
212.          /* finished is used to indicate when producer should finish once no more
213.             requests are available */
214.          finished = 0;
215.
216.          /* Keeps track of number of requests added to buffer */
217.          requestNo = 0;
218.
219.          while(finished == 0)
220.          {
221.              /* Obtain next request from input file using readNextValue
    (fileIO.c) */
222.              readPointer = readNextValue(reading);
223.
224.              /* When requests are exhausted, readNextValue returns a value of 66,
225.                 which ordinarily is not a possible request */
226.              if(readPointer[0] == 66)
227.              {
228.                  /* Breaks while loop and ends producer */
229.                  finished = 1;
230.
231.                  /* Tells each consumer that no more requests are available */
232.                  liftArray[0].finished = 1;
```

```
233.                 liftArray[1].finished = 1;
234.                 liftArray[2].finished = 1;
235.
236.                 /* Used as a redundancy. Sometimes, once requests are finished,
237.                    the consumer processes may still be waiting for a signal from
    the
238.                    producer for more requests. Manually signalling them to
    continue
239.                    will avoid this almost-deadlock (Need 3 for each consumer) */
240.                 sem_post(full);
241.                 sem_post(full);
242.                 sem_post(full);
243.             }
244.
245.             else
246.             {
247.                 /* Lock mutex and stop if buffer is currently full */
248.                 sem_wait(empty);
249.                 sem_wait(mutex);
250.
251.                 /* Add new request to buffer and change *in which points to
    current
252.                    request index (for FIFO queue) */
253.                 liftBuffer[*in].source = readPointer[0];
254.                 liftBuffer[*in].destination = readPointer[1];
255.                 requestNo++;
256.                 *in = (*in+1)%bufferSize;
257.
258.                 /* Unlock mutex and signal waiting consumers */
259.                 sem_post(mutex);
260.                 sem_post(full);
261.
262.                 /* Lock file mutex and use writeBuffer (fileIO.c) to write new
    buffer
263.                    to output file */
264.                 sem_wait(fileOut);
265.                 writeBuffer(readPointer[0], readPointer[1], requestNo);
266.                 sem_post(fileOut);
267.             }
268.         }
269.
270.         printf("Lift-R Process Finished ...\n");
271.
272.         /* Close all semaphores */
273.         sem_close(mutex);
274.         sem_close(full);
275.         sem_close(empty);
276.         sem_close(fileOut);
```

```
277.      }
278.
279.      /*
280.       * Function: lift
281.       * Purpose: Consumer which serves lift request from shared buffer
282.       */
283.      void lift(int i, int bufferSize, int sleepTime)
284.      {
285.          int finishLift;
286.
287.          /* Variable to indicate when consumer should finish */
288.          finishLift = 0;
289.
290.          while(finishLift == 0)
291.          {
292.              /* Lock mutex and wait if buffer is empty */
293.              sem_wait(full);
294.              sem_wait(mutex);
295.
296.              /* Check if requests from input file have finished and if buffer is
297.                 currently empty (Producer may have signalled that buffer is not
    empty
298.                 to avoid infinite loop even though buffer is empty */
299.              if((liftArray[i].finished == 0) || (*in != *out))
300.              {
301.                  /* Remove request from buffer and server the request */
302.                  liftArray[i].source = liftBuffer[*out].source;
303.                  liftArray[i].destination = liftBuffer[*out].destination;
304.                  liftArray[i].movement = abs(liftArray[i].prevRequest -
    liftArray[i].source) + abs(liftArray[i].destination - liftArray[i].source);
305.                  liftArray[i].totalMovement += liftArray[i].movement;
306.                  liftArray[i].totalRequests++;
307.
308.                  /* Lock file mutex and write lift request to output using
    writeLift (fileIO.c) */
309.                  sem_wait(fileOut);
310.                  writeLift(&liftArray[i]);
311.                  sem_post(fileOut);
312.
313.                  /* Make lifts previous request the destination that was just
    served */
314.                  liftArray[i].prevRequest = liftArray[i].destination;
315.
316.                  /* Change current buffer index */
317.                  *out = (*out+1)%bufferSize;
318.              }
319.
320.              else
```

```
321.             {
322.                 /* Change finishLift to indicate that consumer should end loop
    */
323.                 finishLift = 1;
324.             }
325.
326.             /* Unlock mutex and signal waiting producer */
327.             sem_post(mutex);
328.             sem_post(empty);
329.
330.             /* Allow user define time to pass for lift request */
331.             sleep(sleepTime);
332.         }
333.
334.         printf("%s Process Finished ...\n", liftArray[i].name);
335.
336.         /* Close all semaphores */
337.         sem_close(mutex);
338.         sem_close(full);
339.         sem_close(empty);
340.         sem_close(fileOut);
341.     }
```

lift_sim_B.h

```
1. #ifndef LB_H
2. #define LB_H
3.
4. void initialise();
5. void request(int bufferSize);
6. void lift(int i, int bufferSize, int sleepTime);
7.
8. #endif
```

structs.h

```
1. #ifndef S_H
2. #define S_H
3.
4. typedef struct {
5.     int source;
6.     int destination;
7. } buffer;
8.
9. typedef struct {
10.     int source;
11.     int destination;
12.     int prevRequest;
```

```
13.     int movement;
14.     int totalMovement;
15.     int totalRequests;
16.     int finished;
17.     char name[6];
18. } lifts;
19.
20. #endif
```

fileIO.c

```
1.  /* File: fileIO.c
2.   * Author: Adil Khokhar (19182405)
3.   * Purpose: Responible for all File Input/Output for Lift Simulator
4.   * Date Modified: 16/05/2020
5.   */
6.
7.  #include <stdio.h>
8.  #include <stdlib.h>
9.
10. #include "structs.h"
11. #include "fileIO.h"
12.
13. FILE *fileInput;
14. FILE *fileOutput;
15.
16. /*
17.  * Function: openFiles
18.  * Purpose: Open input and output files for lift simulator
19.  */
20. void openFiles()
21. {
22.     fileInput = fopen("sim_input", "r+");
23.
24.     if (fileInput == NULL)
25.     {
26.         printf("Could not open file\n");
27.     }
28.
29.     fileOutput = fopen("sim_output", "w+");
30.
31.     if (fileInput == NULL)
32.     {
33.         printf("Could not open file\n");
34.     }
35.
36.     fclose(fileOutput);
37. }
```

```
38.
39. /*
40.  * Function: readNextValue
41.  * Purpose: Read next set of values from input file. If file has been fully
42.  * read, return the number 66 (not a normally accepted parameter) to
43.  * indicate file is finished
44.  */
45. int* readNextValue(int* reading)
46. {
47.     if(fscanf(fileInput, "%d %d", &reading[0], &reading[1]) != EOF)
48.     {
49.     }
50.
51.     else
52.     {
53.         reading[0] = 66;
54.     }
55.
56.     return reading;
57. }
58.
59. /*
60.  * Function: writeBuffer
61.  * Purpose: Print source, destination and the current request number to output
62.  * file when new item gets added to lift buffer
63.  */
64. void writeBuffer(int source, int destination, int requestNo)
65. {
66.     fileOutput = fopen("sim_output", "a");
67.     fprintf(fileOutput, "--------------------------------\n");
68.     fprintf(fileOutput, "New Lift Request From Floor %d to %d\n", source,
    destination);
69.     fprintf(fileOutput, "Request No: %d\n", requestNo);
70.     fprintf(fileOutput, "--------------------------------\n");
71.     fprintf(fileOutput, "\n");
72.     fclose(fileOutput);
73. }
74.
75. /*
76.  * Function: writeLift
77.  * Purpose: Write the lift movement summary when lift serves request from buffer
78.  * (accepts lifts struct pointer and uses structs.h)
79.  */
80. void writeLift(lifts* lift)
81. {
82.     fileOutput = fopen("sim_output", "a");
83.     fprintf(fileOutput, "%s Operation\n", lift->name);
84.     fprintf(fileOutput, "Previous Position: Floor %d\n", lift->prevRequest);
```

```
85.     fprintf(fileOutput, "Request: Floor %d to %d\n", lift->source, lift-
    >destination);
86.     fprintf(fileOutput, "Detail Operations\n");
87.     fprintf(fileOutput, "   Go from Floor %d to Floor %d\n", lift->prevRequest,
    lift->source);
88.     fprintf(fileOutput, "   Go from Floor %d to Floor %d\n", lift->source, lift-
    >destination);
89.     fprintf(fileOutput, "   #movement for this request: %d\n", lift->movement);
90.     fprintf(fileOutput, "   #request: %d\n", lift->totalRequests);
91.     fprintf(fileOutput, "   Total #movement: %d\n", lift->totalMovement);
92.     fprintf(fileOutput, "Current position: Floor %d\n", lift->destination);
93.     fprintf(fileOutput, "\n");
94.     fclose(fileOutput);
95. }
96.
97. /*
98.  * Function: writeResult
99.  * Purpose: Write total movement and total requests at end of output file
100.       * when lift simulator has concluded
101.       */
102.      void writeResult(int movement, int requests)
103.      {
104.          fileOutput = fopen("sim_output", "a");
105.          fprintf(fileOutput, "--------------------------------\n");
106.          fprintf(fileOutput, "Total Number of Requests: %d\n", requests);
107.          fprintf(fileOutput, "Total Number of Movements: %d\n", movement);
108.          fprintf(fileOutput, "--------------------------------\n");
109.          fclose(fileOutput);
110.      }
111.
112.      /*
113.       * Function: closeFiles
114.       * Purpose: Close input file once simulator has finished
115.       */
116.      void closeFiles()
117.      {
118.          fclose(fileInput);
119.      }
```

fileIO.h

```
1. #ifndef F_H
2. #define F_H
3.
4. void openFiles();
5. int* readNextValue();
6. void writeBuffer(int source, int destination, int requestNo);
7. void writeLift(lifts* lift);
```

```
8.  void writeResult(int movement, int requests);
9.  void closeFiles();
10. 
11. #endif
```

gen_sim_input.c

```
1.  /* File: gen_sim_input.c
2.   * Author: Adil Khokhar (19182405)
3.   * Purpose: Generate random input file for lift simulator
4.   * Date Modified: 16/05/2020
5.   */
6.  
7.  #include <stdio.h>
8.  #include <stdlib.h>
9.  #include <time.h>
10. 
11. int main(void)
12. {
13.     FILE *fp;
14.     int randomNum1, randomNum2;
15.     int numRequests;
16.     int ii;
17. 
18.     /* Open input file to write */
19.     fp = fopen("sim_input", "w+");
20. 
21.     if (fp == NULL)
22.     {
23.         printf("Could not open file\n");
24.     }
25. 
26.     else
27.     {
28.         /* Set starting point for random integers */
29.         srand(time(0));
30. 
31.         /* Generates random number from 50 to 100 (Number of Requests) */
32.         numRequests = (rand() % (100 - 50 + 1)) + 50;
33. 
34.         printf("Num requests = %d\n", numRequests);
35. 
36.         /* Picks two random numbers between 1 and 20 and prints on new lines each
    time */
37.         for(ii = 0; ii < numRequests; ii++)
38.         {
39.             randomNum1 = (rand() % (20 - 1 + 1)) + 1;
40.             randomNum2 = (rand() % (20 - 1 + 1)) + 1;
```

```
41.
42.             while(randomNum1 == randomNum2)
43.             {
44.               randomNum1 = (rand() % (20 - 1 + 1)) + 1;
45.               randomNum2 = (rand() % (20 - 1 + 1)) + 1;
46.             }
47.
48.             fprintf(fp,"%d %d\n",randomNum1, randomNum2);
49.         }
50.
51.         fclose(fp);
52.     }
53.
54.     return 0;
55. }
```

Read-Me File

Lift Simulator

Introduction

This application simulates three lifts serving requests from an input file. There are two different executables, lift_sim_A and lift_sim_B. The first executable uses POSIX threads to run this simulation while the second uses Processes, POSIX Semaphores and POSIX shared memory.

There are 20 floors that can be served and there can be anywhere from 50 to 100 requests. The requests themselves are placed into a buffer (size is user defined) during the simulation and there is user-defined delay between each lift request (to simulate the time it takes for a lift to complete a request).

Compiling Program

This program comes with a Makefile which simplifies compiling each file. To compile all the files, run the command:

```
$ make
```

To remove all the executable and .o files, run:

```
$ make clean
```

Using Program

Input File

To initially start the program, an input file must be generated which contains 50-100 lift requests in the correct format. To simplify things, an executable called 'gen_sim_input' is compiled along with the rest of the executables. This program generates a random input file in the correct format.

To run this program, run the command:

```
$ ./gen_sim_input
```

The console should print a line like this:

```
Num requests = 50
```

50 will be replaced by the number of requests generated in the input file (sim_input).

Running Simulation Using Threads

To simulate the lifts using threads, the executable called 'lift_sim_A' must be used. To run the program, use the command:

```
$ ./lift_sim_A ${buffer_size} ${sleep_time}
```

Replace `${buffer_size}` with the size of the buffer that stores the lift requests during simulation (any integer more than or equal to 1). Also, replace `${sleep_time}` with the time that the lifts will take to serve a request (any integer more than or equal to 0).

Running Simulation Using Processes

To simulate the lifts using processes, the executable called 'lift_sim_B' must be used. To run the program, use the command:

```
$ ./lift_sim_B ${buffer_size} ${sleep_time}
```

Replace `${buffer_size}` with the size of the buffer that stores the lift requests during simulation (any integer more than or equal to 1). Also, replace `${sleep_time}` with the time that the lifts will take to serve a request (any integer more than or equal to 0).

Output File

All of the actions of the lift simulator for both executables are stored in a file called 'sim_output'.

The structure of the 'sim_output' are as follows (Numbers replaced as necessary):

- Lift Request added to buffer

```
--------------------------------------------
New Lift Request From Floor 10 to Floor 2
Request No: 21
--------------------------------------------
```

- Lift serves request from buffer

```
Lift-1 Operation
Previous position: Floor 8
Request: Floor 2 to Floor 9
Detail operations:
 Go from Floor 8 to Floor 2
 Go from Floor 2 to Floor 9
#movement for this request: 13
#request: 5
Total #movement: 70
Current position: Floor 9
```

- Result when simulator ends

```
--------------------------------------------
Total number of requests: 100
Total number of movements: 5520
--------------------------------------------
```

Mutual Exclusion/Shared Resources

lift_sim_A (Threads)

In the threads part of this assignment, mutual exclusion is achieved through the use of mutex locks. When any thread enters its critical section, a mutex lock is used to make sure only that thread is using any shared variables/resources. As well as this, condition variables are also used which come into play when the lift buffer is empty or full. The condition variable will make the thread 'go to sleep' and unlock the mutex it was using. It will then wait for a signal for another condition variable when the buffer is changed. The thread will then 'wake up', lock the mutex again and enter its critical section.

The two shared resources in this program are the lift buffer and the output file. No shared memory was needed to be allocated since threads by design share all the resources that their parent process has access to.

lift_sim_B (Processes)

In the processes part of this assignment, semaphores are used to achieve mutual exclusion. When any child process enters its critical section and tries to access the lift buffer, it decrements a semaphore called mutex (making its value 0). When any other child process tries to enter its critical section, it will also try to decrement the mutex. However, since the mutex is 0, it will fail and ultimately 'go to sleep'. The initial process (which decremented the semaphore initially) will increment the value of the semaphore back to 1 when it is done accessing the buffer. This will allow another child process to enter its critical section.

A similar semaphore called fileOut is used in the exact same manner as mutex when a child process tries to access the output file. Both of these semaphores are known as binary semaphores.

Two counting semaphores are also used (full and empty) to indicate when the buffer is full/empty. Their minimum value is 0 and their maximum value is the buffer size. They work exactly the same way as the binary semaphores and are contained within the mutex semaphore.

There are four shared resources in the processes that had to be allocated dynamic shared memory. These are the lift buffer, the array of three lifts and an in and out integer pointer. The lift buffer is used to access the shared buffer. The array of three lifts contains the current value of all three lifts (source, destination, movement etc.). The in and out integer pointers point to the current buffer index which makes sure the buffer operates as a FIFO queue.

Imperfections

As far as the required program specifications, the program works as expected. During the programming segment of this assignment, there were many print statements in each critical section of the threads/processes. This was for debugging purposed as you could see exactly what the code was doing. This includes if the buffer was operating correctly (and if it was being filled to its capacity), if the mutual exclusion was working correctly and if the shared resources were being closed/exited properly.

The only imperfection that could be analysed further would be a potential memory leak. When you use valgrind to run the program, it says there is some memory still reachable. This isn't bad since no memory is lost. However, it could be due to the way the semaphores work with their shared memory. This would have to be further researched.

Sample Input/Output

```
adil@programmer:~/Lift-Simulator$ ./gen_sim_input
Num requests = 58
```

Figure 1: Generating Input File

```
adil@programmer:~/Lift-Simulator$ cat sim_input
1 13
16 1
17 16
17 4
11 8
3 10
16 17
4 5
6 9
7 2
12 14
8 7
16 8
4 6
7 4
10 3
4 7
10 13
10 13
20 4
2 15
20 17
11 15
13 16
```

Figure 2: Input File

```
adil@programmer:~/Lift-Simulator$ ./lift_sim_A 10 1
Lift-R Thread Created and Running ...
Lift-1 Thread Created and Running ...
Lift-2 Thread Created and Running ...
Lift-3 Thread Created and Running ...
Lift-R Thread Finished ...
Lift-2 Thread Finished ...
Lift-3 Thread Finished ...
Lift-1 Thread Finished ...
```

Figure 3: Running Thread Lift Simulator

```
--------------------------------
New Lift Request From Floor 9 to 2
Request No: 57
--------------------------------

--------------------------------
New Lift Request From Floor 5 to 9
Request No: 58
--------------------------------

Lift-1 Operation
Previous Position: Floor 17
Request: Floor 16 to 1
Detail Operations
   Go from Floor 17 to Floor 16
   Go from Floor 16 to Floor 1
   #movement for this request: 16
   #request: 17
   Total #movement: 208
Current position: Floor 1

Lift-2 Operation
Previous Position: Floor 4
Request: Floor 17 to 13
Detail Operations
   Go from Floor 4 to Floor 17
   Go from Floor 17 to Floor 13
   #movement for this request: 17
   #request: 17
   Total #movement: 226
Current position: Floor 13

Lift-3 Operation
Previous Position: Floor 12
Request: Floor 18 to 5
Detail Operations
   Go from Floor 12 to Floor 18
   Go from Floor 18 to Floor 5
   #movement for this request: 19
   #request: 17
   Total #movement: 194
Current position: Floor 5
```

Figure 4: Output from thread program

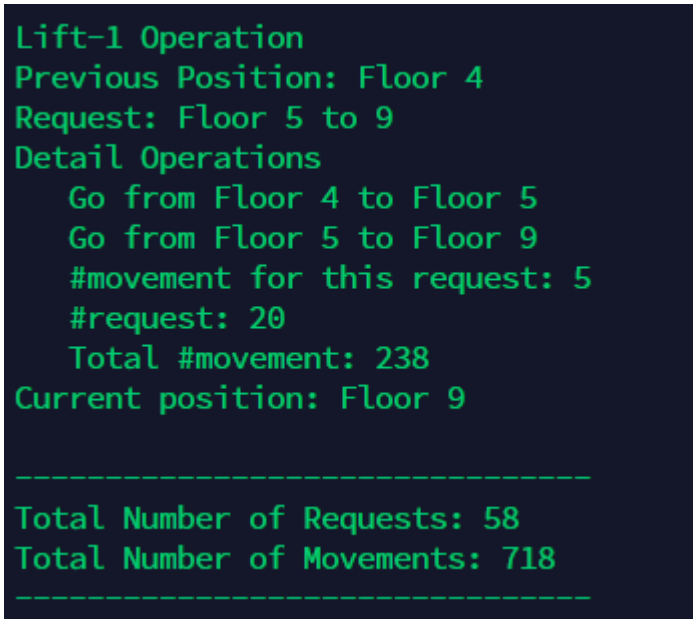

Figure 5: Output from thread program

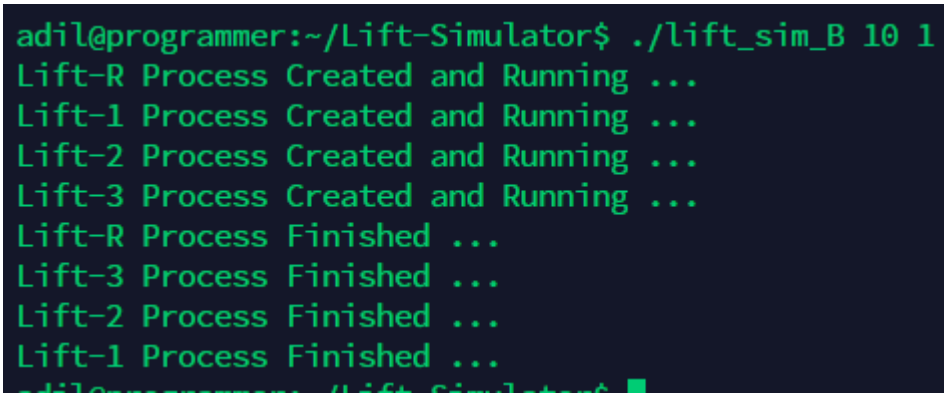

Figure 6: Running Process Lift Simulator

```
------------------------------
New Lift Request From Floor 5 to 9
Request No: 58
------------------------------

Lift-1 Operation
Previous Position: Floor 17
Request: Floor 16 to 1
Detail Operations
   Go from Floor 17 to Floor 16
   Go from Floor 16 to Floor 1
   #movement for this request: 16
   #request: 17
   Total #movement: 208
Current position: Floor 1
```

Figure 3: Output from process simulator

```
Lift-2 Operation
Previous Position: Floor 18
Request: Floor 9 to 2
Detail Operations
   Go from Floor 18 to Floor 9
   Go from Floor 9 to Floor 2
   #movement for this request: 16
   #request: 19
   Total #movement: 223
Current position: Floor 2

Lift-1 Operation
Previous Position: Floor 4
Request: Floor 5 to 9
Detail Operations
   Go from Floor 4 to Floor 5
   Go from Floor 5 to Floor 9
   #movement for this request: 5
   #request: 20
   Total #movement: 238
Current position: Floor 9

------------------------------
Total Number of Requests: 58
Total Number of Movements: 718
------------------------------
```

Figure 8: Output from process simulator