Done by:

Yuguang Huang

Shreyas Bhivandkar

Do Nhat Anh

Sadman Sakib Hasan

EECS 3221 – Assignment 3

1. **Introduction**

The main purpose of this assignment was programming with POSIX thread. We had to code the program with using POSIX unnamed semaphores. Like the previous assignment we had to complete the code given by the instructor and generate a makefile that would run our program and give us outputs – all which was mentioned in the README file. We had to test multiple outputs to make sure the program was working and in the end complete a fine tuned report which explained every part of the assignment in much depth.

For the first part of the assignment we had to implement two types of alarm requests. These had to be recognized and handled by our program New\_Alarm\_Cond.c, as follows:

First type of Alarm request, which included a message and a message number as the second parameter and the second type of alarm request was prefixed with the key word ***Cancel*.** We had to make sure if the user typed in something other than one of the above two types of valid alarm requests, then an error message will be displayed, and the invalid request will be discarded.

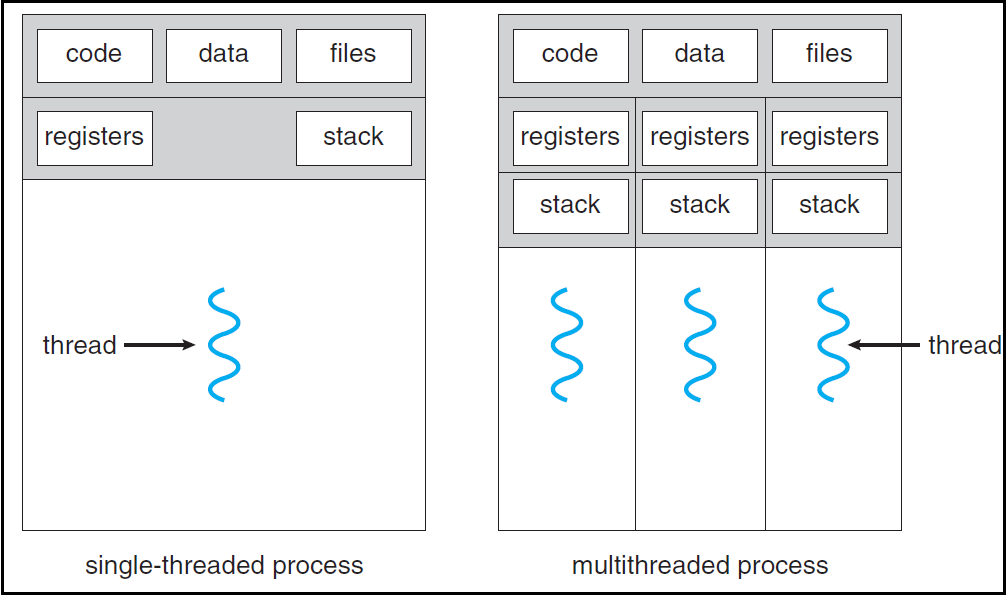
For the second part of the assignment we had to implement a main thread in New\_Alarm\_Cond.c which would first determine whether the format of each alarm request is consistent with one of the two formats mentioned above. If the Message string exceeds 128 characters, it will be truncated to 128 characters. The main thread first checks whether the format of each alarm request is consistent with one of the two formats above, otherwise the alarm request will be rejected with an error message. We had to make sure that if there does not exist any alarm request with the same message number in an alarm list, then the main thread will insert the alarm request that was just received into the alarm list, and if an alarm request exist with the same message number then we replace it with the most recent one.

For the third part of the assignment we had to make sure that the alarm\_thread in New\_Alarm\_Cond.c checks alarm requests in the alarm list whenever the alarm list has been changed. If the alarm request is NOT prefixed with the keyword ***Cancel*** in the alarm list, then the alarm\_thread will immediately create a periodic\_display\_thread. Also one thing to keep in mind was that once we found an alarm request prefixed with the keyword ***Cancel*** in the alarm list, the alarm\_thread will remove the all the data corresponding to the alarm request with the corresponding message number from the alarm list.

For the fourth part of the assignment was linked to the third part because we had to implement periodic\_display\_thread in New\_Alarm\_Cond.c. The purpose of this was that each periodic\_display\_thread was responsible for periodically looking up an alarm request with a specific message number in the alarm list, then printing, every Time seconds, where Time is the first parameter in an alarm request. We had to print the thread accordingly as to the message number.

The last part of the assignment was the Synchronization of thread accesses to the alarm list, by treating the threads as readers and writers in New\_Alarm\_Cond.c. We were required to synchronize thread accesses to the shared data – the alarm list, by treating the threads as readers and writers, and implementing a solution to the ***Readers-Writers*** problem as we have learned in class and as mentioned in the book. To distinguish between Reader-Writer; a­­­­­ny thread that needs to modify the alarm list, should be treated as a writer process - only one writer process should be able to modify the alarm list at a time, while any thread that only needs to read information from the alarm list, should be treated as a reader process - any number of reader processes should be able to read information from the alarm list simultaneously.

1. **Description**

This assignment gives an illustration in utilizing a semaphore as a synchronization tool. It’s a deep insight into the usage of semaphores.   
  
  
Figure 1: Concepts of Single-Threaded vs Multi-Threaded Process  
  
Semaphores  
  
In software engineering, especially in working frameworks, a semaphore is a variable or unique information sort that is utilized for controlling access, by various methods, to a typical asset in a parallel programming or a multi-client environment. Semaphores are furnished with two operations, verifiably meant as V (otherwise Signal) and P (or Wait). Operation V augments the semaphore S, and operation P decrements it.

The estimation of the semaphore S is the quantity of units of the asset that are presently accessible. The P operation squanders time or rests until an asset secured by the semaphore gets to be accessible, at which time the asset is quickly asserted. The V operation is the reverse: it makes an asset accessible again after the procedure has completed the process of utilizing it. One essential property of semaphore S is that its esteem can't be changed aside from by utilizing the V and P operations.

A straightforward approach to comprehend hold up and signal operations is:

* Wait: If the estimation of semaphore variable is not invalid, decrements it by 1. Something else, the procedure executing holds up is blocked (i.e., added to the semaphore's line) until the worth is more noteworthy or equivalent to 1.

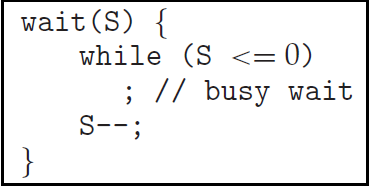
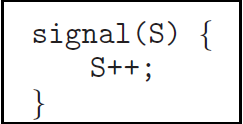


Figure 2: Wait or P Procedure

* Signal: Increments the estimation of semaphore variable by 1. After the augmentation, if the pre increment worth was negative (significance there are techniques sitting tight for an asset), it exchanges a blocked methodology from the semaphore's holding up line to the prepared l

  
Figure 3: Signal or V Procedure

**3. Requirements**

***A3.1 Two types of “Alarm requests” in “New\_Alarm\_Cond.c”***

(a) Alarm requests which include, as a second parameter, a Message Number, with

the following format:

The first requirement from this assignment is to increase the type of "Alarm request" the program can handle compare to the previous assignment. Our group have no trouble with this part of the assignment, it is solved by simple adding another sscanf in the main function. In this part we also decide to implement a new void function: alarm\_cancel (alarm\_t \*alarm) used to handle the second type of "Alarm request". We believe this is the better design instead of using void function: alarm\_insert (alarm\_t \*alarm).

(b) Alarm requests prefixed with the key word “Cancel:”, with the following format:

The second requirement from this assignment was to implement the other type of input command “Cancel”. The cancel command would cancel the command with the specified message number.

Our group had no problems figuring out the Cancel command because we just had to change the sscanf and only let the user input input the message number. Implementing the cancel command was somewhat challenging because we had to remove the alarm completely from the list. But our team were able to fix it. We solved by searching through the list for the message number inputted by the user and if there was an alarm with that message number we just pulled it off the list.

***A3.2. The main thread in “New\_Alarm\_Cond.c”***

For the next part of the assignment: Handling the main thread in New\_Alarm\_Cond. We first make the choice in alarm\_insert function which is simply replacing any alarm with the same message number existing in the list regardless of its expired time. However, soon we realize that the timing which alarm message expired is incorrect. The reason is due to after replacing the alarm message (seconds, message) the list become unorganized.

For example, with the following input:

100 Message(1) 100

200 Message(2) 200

300 Message(3) 300

120 Message(3) 120

The order of alarm expire should be 1,3,2 but instead 2,3 expire on the same time after 200 seconds.

We first try to solve this problem by searching through the entire list after done replacing the alarm (including the alarm in the waiting list) and move it to the correct position (the linked list is arrange based on the user input time from lowest to highest). This function will handle any change make to the alarm, increasing or decreasing in time. This solution works fine but there is a much better way of doing it, simply by canceling the alarm that is being replaced first and then add a new alarm to the list. This way the current alarm\_insert function is sufficient to handle all input.

The alarm\_cancel function was implemented very similar to alarm\_insert, instead of adding alarm to the linked list, alarm\_cancel will drop it from the linked list while maintaining other alarms. We also decide to add a few new condition in the main thread to cancel the alarm that is running in the waiting list: cancelled condition to signal the thread when to drop the waiting alarm and take in the next one.

***A3.3. The alarm thread in “New\_Alarm\_Cond.c”***

The alarm\_thread in “New\_Alarm\_Cond.c” checks alarm requests in the alarm list whenever the alarm list has changed.

On finding a new alarm request that is NOT prefixed with the keyword “Cancel:” in the alarm list, the alarm\_thread will immediately create a periodic\_display\_thread specified in the next section. (described next in 3.4)

On finding an alarm request prefixed with the keyword “Cancel:” in the alarm list, the alarm\_thread will remove the all the data corresponding to the alarm request with the corresponding Message Number from the alarm list, including both the alarm request with the format specified in A3.1 (a), and the alarm request with the format specified in A3.1 (b).

Our group did not have any problems solving this section.

***A3.4. The “periodic\_display\_threads” in “New\_Alarm\_Cond.c”***

As requested in the assignment, we implemented a new thread responsible for printing alarm and any change has been made to it the moment those change applied. Each alarm will have its own periodic\_display\_thread and all these threads will run separately, only terminate under two conditions: the alarm expired or cancelled. We do not encounter any trouble in this part because we used the same intuition of assignment 2 here.

1. **Programming**
   1. **Source Code**
2. /\*
3. \* alarm\_cond.c
4. \*
5. \* This is an enhancement to the alarm\_mutex.c program, which
6. \* used only a mutex to synchronize access to the shared alarm
7. \* list. This version adds a condition variable. The alarm
8. \* thread waits on this condition variable, with a timeout that
9. \* corresponds to the earliest timer request. If the main thread
10. \* enters an earlier timeout, it signals the condition variable
11. \* so that the alarm thread will wake up and process the earlier
12. \* timeout first, requeueing the later request.
13. \*/
14. #include <pthread.h>
15. #include <time.h>
16. #include "errors.h"
17. #include <strings.h>
18. #define DEBUG
19. /\*
20. \* The "alarm" structure now contains the time\_t (time since the
21. \* Epoch, in seconds) for each alarm, so that they can be
22. \* sorted. Storing the requested number of seconds would not be
23. \* enough, since the "alarm thread" cannot tell how long it has
24. \* been on the list.
25. \*/
26. typedef struct alarm\_tag {
27. struct alarm\_tag \*link;
28. int seconds;
29. int number;
30. time\_t time; /\* seconds from EPOCH \*/
31. char message[128];
32. } alarm\_t;
33. typedef struct display\_thread {
34. int number;
35. int seconds;
36. char message[128];
37. pthread\_t thread;
38. } thread\_struct;
39. pthread\_mutex\_t alarm\_mutex = PTHREAD\_MUTEX\_INITIALIZER;
40. pthread\_cond\_t alarm\_cond = PTHREAD\_COND\_INITIALIZER;
41. pthread\_cond\_t nalarm\_cond = PTHREAD\_COND\_INITIALIZER;
42. alarm\_t \*alarm\_list = NULL, \*alarm\_display = NULL;
43. time\_t current\_alarm = 0;
44. int current\_number = 0;
45. int current\_seconds = 0;
46. int cancelled = 0;
47. int counter = 0;
48. thread\_struct thread\_list[100];
49. void alarm\_cancel (alarm\_t \*alarm);
50. void alarm\_insert (alarm\_t \*alarm);
51. void \*alarm\_thread (void \*arg);
52. void \*pd\_thread (alarm\_t \*alarm);
53. void \*pd\_thread1 (void \*arg) ;
54. /\*
55. \* Insert alarm entry on list, in order.
56. \*/
57. void alarm\_insert (alarm\_t \*alarm)
58. {
59. int status;
60. alarm\_t \*\*last, \*next, \*temp, \* temp2;
61. temp2 = alarm;
62. pthread\_t thread2;
63. /\*
64. \* LOCKING PROTOCOL:
65. \*
66. \* This routine requires that the caller have locked the
67. \* alarm\_mutex!
68. \*/
69. last = &alarm\_list;
70. next = \*last;
71. alarm\_display = alarm;
72. if (alarm->number == current\_number) {
73. alarm\_cancel(alarm);
74. }
76. while (next != NULL) {
77. if (next->number == alarm->number) {
78. alarm\_cancel(next);
79. /\* strcpy(next->message,alarm->message);
80. next->number = alarm->number;
81. next->time = alarm->time;
82. next->seconds = alarm->seconds;
83. while (next->link != NULL){
84. if( next->time >= next->link->time) {
86. (\*last) = (\*last)->link;
87. temp = next;
88. next = next->link;
89. temp->link = next->link;
91. next->link = temp;
93. // printf("Last: %d, Next: %d, Next->link: %d\n", (\*last)->number, next->number, next->link->number);
94. }
95. else {
96. last = &next->link;
97. next = next->link;
98. }
99. }
100. if (next == NULL) {
101. \*last = alarm;
102. alarm->link = NULL;
103. }
104. break;\*/
105. }
106. if (next->time >= alarm->time) {
107. alarm->link = next;
108. \*last = alarm;
109. break;
110. }
111. last = &next->link;
112. next = next->link;
113. }
115. /\*
116. \* If we reached the end of the list, insert the new alarm
117. \* there. ("next" is NULL, and "last" points to the link
118. \* field of the last item, or to the list header.)
119. \*/
120. if (next == NULL) {
121. \*last = alarm;
122. alarm->link = NULL;
123. }
125. pthread\_create (
126. &thread2, NULL, pd\_thread1, NULL);
127. thread\_list[counter].number = alarm->number;
128. thread\_list[counter].thread = thread2;
129. thread\_list[counter].seconds = alarm->seconds;
130. strcpy(thread\_list[counter].message,alarm->message);
131. // printf("in the thread: %d, %d\n", thread2, alarm->number);
132. counter++;
133. // printf("Crosses Create\n");
135. #ifdef DEBUG
136. printf ("[list\_insert: ");
137. for (next = alarm\_list; next != NULL; next = next->link)
138. printf ("%d(%d) number = %d [\"%s\"] ", next->time,
139. next->time - time (NULL), next->number, next->message);
140. printf ("]\n");
141. #endif
142. /\*
143. \* Wake the alarm thread if it is not busy (that is, if
144. \* current\_alarm is 0, signifying that it's waiting for
145. \* work), or if the new alarm comes before the one on
146. \* which the alarm thread is waiting.
147. \*/

150. if(current\_alarm == 0 || alarm->time < current\_alarm)
151. current\_alarm = alarm->time;
152. status = pthread\_cond\_signal (&alarm\_cond);
153. if (status != 0)
154. err\_abort (status, "Signal cond");

157. }
158. void alarm\_cancel (alarm\_t \*alarm)
159. {
160. int status;
161. alarm\_t \*\*last, \*next, \*temp, \*temp2;
163. /\*
164. \* LOCKING PROTOCOL:
165. \*
166. \* This routine requires that the caller have locked the
167. \* alarm\_mutex!
168. \*/
170. last = &alarm\_list;
171. next = \*last;
172. temp = next;
174. int j;
175. for( j = 0; j < counter; j++){
176. if (thread\_list[j].number == alarm->number){
177. //printf("threadid: %d\n", thread\_list[j].thread);
178. pthread\_cancel(thread\_list[j].thread);
179. thread\_list[j].number = 0;
180. printf("Display thread exiting at %d: %d Message(%d) %s\n", time(NULL), thread\_list[j].seconds, alarm->number, thread\_list[j].message);
182. break;
183. }
184. }

187. while (next != NULL) {
188. if (next->number == alarm->number){
189. if(next->link != NULL) {
190. temp->link = next->link;
191. next = NULL;
192. \*last = temp->link;
193. }
194. else if (temp->number != alarm->number){
195. next = NULL;
196. temp->link = NULL;
197. }
198. else {
199. alarm\_list = NULL;
200. }
201. break;
202. }
203. last = &next->link;
204. temp = next;
205. next = next->link;
206. }
208. if(current\_number == alarm->number){
209. //current\_alarm = alarm\_list->time;
210. //current\_number = alarm\_list->number;
211. //current\_seconds = alarm\_list->seconds;
212. // printf("about to exit\n");
213. cancelled = 1;
214. status = pthread\_cond\_signal (&alarm\_cond);
215. if (status != 0)
216. err\_abort (status, "Signal cond");
217. }


221. #ifdef DEBUG
223. printf ("[list1: ");
224. for (next = alarm\_list; next != NULL; next = next->link)
225. printf ("%d(%d) number = %d [\"%s\"] ", next->time,
226. next->time - time (NULL), next->number, next->message);
227. printf ("]\n");
228. #endif
229. /\*
230. \* Wake the alarm thread if it is not busy (that is, if
231. \* current\_alarm is 0, signifying that it's waiting for
232. \* work), or if the new alarm comes before the one on
233. \* which the alarm thread is waiting.
234. \*/


238. }
239. /\*
240. \* The alarm thread's start routine.
241. \*/
242. void \*pd\_thread1 (void \*arg)
243. {
244. alarm\_t \*display\_copy = (alarm\_t\*) malloc(sizeof(alarm\_t\*));
245. display\_copy->seconds = alarm\_display->seconds;
246. strcpy(display\_copy->message, alarm\_display->message);
247. display\_copy->number = alarm\_display->number;
248. while(1) {
250. sleep(display\_copy->seconds);
251. printf("Alarm displayed at %d: %d Message(%d) %s\n", time(NULL), display\_copy->seconds, display\_copy->number, display\_copy->message);
252. }
253. free(display\_copy);
254. //printf("Display thread exiting at %d: %d Message(%d) %s\n", time(NULL), display\_copy->seconds, display\_copy->number, display\_copy->message);
256. return NULL;
257. }
258. void \*alarm\_thread (void \*arg)
259. {
260. alarm\_t \*alarm;
261. struct timespec cond\_time;
262. time\_t now;
263. int status;
264. int expired;
266. /\*
267. \* Loop forever, processing commands. The alarm thread will
268. \* be disintegrated when the process exits. Lock the mutex
269. \* at the start -- it will be unlocked during condition
270. \* waits, so the main thread can insert alarms.
271. \*/
272. status = pthread\_mutex\_lock (&alarm\_mutex);
273. if (status != 0)
274. err\_abort (status, "Lock mutex");
275. while (1) {
276. /\*
277. \* If the alarm list is empty, wait until an alarm is
278. \* added. Setting current\_alarm to 0 informs the insert
279. \* routine that the thread is not busy.
280. \*/
281. current\_alarm = 0;
282. while (alarm\_list == NULL) {
283. status = pthread\_cond\_wait (&alarm\_cond, &alarm\_mutex);
284. if (status != 0)
285. err\_abort (status, "Wait on cond");
286. }
287. alarm = alarm\_list;
288. alarm\_list = alarm->link;
289. now = time (NULL);
290. expired = 0;
291. if (alarm->time > now) {
292. #ifdef DEBUG
293. printf ("[waiting: %d(%d)\"%s\"]\n", alarm->time,
294. alarm->time - time (NULL), alarm->message);
295. #endif
296. cond\_time.tv\_sec = alarm->time;
297. cond\_time.tv\_nsec = 0;
298. current\_alarm = alarm->time;
299. current\_number = alarm->number;
300. current\_seconds = alarm->seconds;
301. printf ("\"Alarm Processed at %d: %d %s\"\n", now, alarm->seconds, alarm->message);
302. while (current\_alarm == alarm->time) {
303. status = pthread\_cond\_timedwait (
304. &alarm\_cond, &alarm\_mutex, &cond\_time);
305. // printf("Came back after waiting\n");
306. if (status == ETIMEDOUT) {
307. expired = 1;
308. break;
309. }
310. if (status != 0)
311. err\_abort (status, "Cond timedwait");
312. }
314. if (!expired)
315. alarm\_insert (alarm);
317. if(cancelled == 1){
318. alarm->time = now;
319. }
321. } else
322. expired = 1;
323. if (expired && cancelled != 1) {
324. printf ("(%d) %s\n", alarm->seconds, alarm->message);
325. free (alarm);
326. }
327. else {
328. cancelled = 0;
329. }
330. }
331. }
332. int main (int argc, char \*argv[])
333. {
334. int status;
335. char line[128];
336. alarm\_t \*alarm;
337. pthread\_t thread, thread2;
339. status = pthread\_create (
340. &thread, NULL, alarm\_thread, NULL);
341. if (status != 0)
342. err\_abort (status, "Create alarm thread");
343. while (1) {
344. printf ("Alarm> ");
345. if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
346. if (strlen (line) <= 1) continue;
347. alarm = (alarm\_t\*)malloc (sizeof (alarm\_t));
348. if (alarm == NULL)
349. errno\_abort ("Allocate alarm");
351. /\*
352. \* Parse input line into seconds (%d) and a message
353. \* (%128[^\n]), consisting of up to 128 characters
354. \* separated from the seconds by whitespace.
355. \*/
356. if (sscanf (line, "%d Message(%d) %128[^\n]",
357. &alarm->seconds, &alarm->number, &alarm->message) == 3) {
358. status = pthread\_mutex\_lock (&alarm\_mutex);
359. if (status != 0)
360. err\_abort (status, "Lock mutex");
361. alarm->time = time (NULL) + alarm->seconds;
363. printf ("\"Alarm Recieved at %d: %d %s\"\n", time(NULL), alarm -> seconds, alarm->message);
364. /\*
365. \* Insert the new alarm into the list of alarms,
366. \* sorted by expiration time.
367. \*/
368. alarm\_insert (alarm);

371. status = pthread\_mutex\_unlock (&alarm\_mutex);
372. if (status != 0)
373. err\_abort (status, "Unlock mutex");
375. }
376. else if (sscanf (line, "Cancel Message(%d)",
377. &alarm->number) == 1) {
378. status = pthread\_mutex\_lock (&alarm\_mutex);
379. if (status != 0)
380. err\_abort (status, "Lock mutex");
382. alarm->time = time (NULL) + alarm->seconds;
383. alarm\_cancel (alarm);
385. status = pthread\_mutex\_unlock (&alarm\_mutex);
386. if (status != 0)
387. err\_abort (status, "Unlock mutex");
388. }
389. else {
390. fprintf (stderr, "Bad command\n");
391. free (alarm);
392. }
393. }
394. }
395. **Test Cases**
396. red 40 % cc New\_Alarm\_Cond.c -D\_POSIX\_PTHREAD\_SEMANTICS -lpthread
397. red 41 % a.out
398. Alarm> 10 Message(1) 10
399. "Alarm Recieved at 1449548506: 10 10"
400. Alarm> "Alarm Processed at 1449548506: 10 10"
401. 7 Message(1) 7Alarm displayed at 1449548516: 10 Message(1) 10
402. "Alarm Recieved at 1449548517: 7 7"
403. Display thread exiting at 1449548517: 10 Message(1) 10
404. Alarm> "Alarm Processed at 1449548517: 7 7"
405. Alarm displayed at 1449548524: 7 Message(1) 7
406. 9 Alarm displayed at 1449548531: 7 Message(1) 7
407. Alarm> 9 Message(2) 9Alarm displayed at 1449548538: 7 Message(1) 7
408. "Alarm Recieved at 1449548539: 9 9"
409. Alarm> "Alarm Processed at 1449548539: 9 9"
410. Alarm displayed at 1449548545: 7 Message(1) 7
411. Alarm displayed at 1449548548: 9 Message(2) 9
412. 13 Alarm displayed at 1449548552: 7 Message(1) 7
413. Message(3) 13
414. "Alarm Recieved at 1449548555: 13 13"
415. Alarm> "Alarm Processed at 1449548555: 13 13"
416. Alarm displayed at 1449548557: 9 Message(2) 9
417. Alarm displayed at 1449548559: 7 Message(1) 7
418. Alarm displayed at 1449548566: 9 Message(2) 9
419. Alarm displayed at 1449548566: 7 Message(1) 7
420. Alarm displayed at 1449548568: 13 Message(3) 13
421. Cancel Message(1)Alarm displayed at 1449548573: 7 Message(1) 7
422. Display thread exiting at 1449548574: 7 Message(1) 7
423. Alarm> Alarm displayed at 1449548575: 9 Message(2) 9
424. Alarm displayed at 1449548581: 13 Message(3) 13
425. CaAlarm displayed at 1449548584: 9 Message(2) 9
426. ncel Message(3)
427. Display thread exiting at 1449548587: 13 Message(3) 13
428. Alarm> Alarm displayed at 1449548593: 9 Message(2) 9
429. Alarm displayed at 1449548602: 9 Message(2) 9
430. **Makefile**

.PHONY : clean

alarm\_mutex : New\_Alarm\_Cond.c

cc -o alarm\_mutex New\_Alarm\_Cond.c -D\_POSIX\_PTHREAD\_SEMANTICS -lpthread

./alarm\_mutex

.PHONY : clean

clean :

-rm \*.o $(objects)

**d. Errors.h file**

#ifndef \_\_errors\_h

#define \_\_errors\_h

#include <unistd.h>

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\*

\* Define a macro that can be used for diagnostic output from

\* examples. When compiled -DDEBUG, it results in calling printf

\* with the specified argument list. When DEBUG is not defined, it

\* expands to nothing.

\*/

#ifdef DEBUG

# define DPRINTF(arg) printf arg

#else

# define DPRINTF(arg)

#endif

/\*

\* NOTE: the "do {" ... "} while (0);" bracketing around the macros

\* allows the err\_abort and errno\_abort macros to be used as if they

\* were function calls, even in contexts where a trailing ";" would

\* generate a null statement. For example,

\*

\* if (status != 0)

\* err\_abort (status, "message");

\* else

\* return status;

\*

\* will not compile if err\_abort is a macro ending with "}", because

\* C does not expect a ";" to follow the "}". Because C does expect

\* a ";" following the ")" in the do...while construct, err\_abort and

\* errno\_abort can be used as if they were function calls.

\*/

#define err\_abort(code,text) do { \

fprintf (stderr, "%s at \"%s\":%d: %s\n", \

text, \_\_FILE\_\_, \_\_LINE\_\_, strerror (code)); \

abort (); \

} while (0)

#define errno\_abort(text) do { \

fprintf (stderr, "%s at \"%s\":%d: %s\n", \

text, \_\_FILE\_\_, \_\_LINE\_\_, strerror (errno)); \

abort (); \

} while (0)

#endif

1. **README**

1. You can compile and run the program either manually or automatically using the Make file we have included for you.

1-A) If you would like to compile the program "New\_Alarm\_Cond.c" manually, use the following command:

a. cc New\_Alarm\_Cond.c -D\_POSIX\_PTHREAD\_SEMANTICS -lpthread

b. Then type "a.out" to run the executable code.

1-B) If you would like to compile the program Automatically using our Make file, use the following command:

a. Make sure the makefile file is in assignment 3 directory with all other files.

b. Open terminal at assignment 3 directory.

c. Type make and hit enter.

d. To terminate make sure to use CTRL + C and not CTRL + D.

2. At the prompt "ALARM>", type in the number of seconds at which

the alarm should expire, followed by the text of the message.

For example:

Alarm> 5 Message(2) Buy groceries at Metro

OR

Cancel: Message(2)

(To exit from the program, type Ctrls-s)

1. **References**
   * Chapter 5, pages 237-238 of the course textbook, “Operating System Concepts” for POSIX unnamed semaphores
   * Chapter 5, pages 220-222 of the course textbook, “Operating System Concepts” for Readers-Writers problem