```c
/*
* New_Alarm_Cond.c

   AUTHORS **
  Francis Okoyo             213811336
  Tyler Noble               212270369
  Adham El-Shafie           212951018
  Lindan Thillanayagam      213742176


* This is an enhancement to the alarm_cond.c program, which
* used only a mutex and condition variables to synchronize acces
* to a list of alarms. This versions does the job by only using
* semaphores
*/
#include <pthread.h>
#include <time.h>
#include "errors.h"
#include <semaphore.h>

/*
* The "alarm" structure now contains the time_t (time since the
* Epoch, in seconds) for each alarm, so that they can be
* sorted. Storing the requested number of seconds would not be
* enough, since the "alarm thread" cannot tell how long it has
* been on the list.
*/
typedef struct alarm_tag {
  struct alarm_tag    *link;
  int                 seconds;
  time_t              time;   /* seconds from EPOCH */
  char                message[128];

  /******* new additions to the alarm_tag structure ********/
  int               type; //identifies the message type ( type >= 1 )
  int               prev_type; // previous message type
  int               is_new; // 1 == "new" and 0 == "old/not new"
  int               number; /* Message Number */
  int               request_type; // TypeA == 1 TypeB == 2 TypeC == 3
  int               expo; // signifies that the type change was confirmed
  int               first;
  /********************end new additions***************/
} alarm_t;

/*
*
* Thread structure used to keep a linked list of thread id's which will be
* organized by their message type.
*
* This is a replacement of the sparce matrix used in the previous project
* "New_Alarm_Mutex.c". This is a lot more efficient as it does not Allocate
* an unneccesary amount of space. we do loose the O(1) acces time though.
*/
typedef struct thread_tag { // NEW STRUCT
  struct thread_tag     *link;
  pthread_t             thread_id;
  int                   type;
```

```
58   int                     number;
59
60 } thread_t;
61
62 sem_t rw_sem, sem_mutex;
63 int read_count = 0; // number o readers using the list
64 int writing = 0; //flag to notify that there is a writer writing to the list
65 int ready = 0; // flag to notify readers that a writer is about to write
66
67 alarm_t *alarm_list = NULL;
68 time_t current_alarm = 0;
69 thread_t *thread_list = NULL;  // List of Thread id's
70
71 const int TYPE_A = 1; // Constants to specify alarm request type
72 const int TYPE_B = 2;
73 const int TYPE_C = 3;
74
75 int insert_flag; //1 if a new alarm has been inserted. set to 0 after processing
76
77 int debug_flag;
78
79 /*************************HELPER CODE*************************/////////////
80 /*
81 * prints out contents of the thread list as well as the contents of the alarm
82 * list for debugging
83 */
84 void display_lists(){
85   thread_t **last, *next;
86   alarm_t **alast, *anext;
87
88   last = &thread_list;
89   next = *last;
90
91   alast = &alarm_list;
92   anext = *alast;
93
94   printf ("\n[Thread List: ");
95   for (next = thread_list; next != NULL; next = next->link)
96     printf ("{message type = %d thread_id = <%lu>} ",next->type,next->thread_id);
97   printf ("]\n");
98
99   printf ("[Alarm List: ");
100    for (anext = alarm_list; anext != NULL; anext = anext->link)
101      printf (" {Request Type = %d Alarm # = %d message type = %d} ",
102              anext->request_type, anext->number, anext->type);
103  printf ("]\n");
104 }
105
106
107 /*
108 * checks if an alarm's type has changed
109 *
110 * returns 1 if so and 0 otherwise
111 */
112 int check_prev(alarm_t *a){
113
114  if(a->type != a->prev_type)
```

```
115     return 1;
116
117   return 0;
118 }
119
120
121 /*
122 * Check the alarm list to see if a Type A alarm of this type number exists.
123 *
124 * return 1 if so and 0 otherwise.
125 *
126 */
127 int check_type_a_exists(int type){
128   int     status;
129   alarm_t *next, **last;
130
131   last = &alarm_list;
132   next = *last;
133   while (next != NULL) {
134     if(next->type == type && next->request_type == TYPE_A){
135
136       return 1;
137     }
138
139     last = &next->link;
140     next = next->link;
141   }
142
143   return 0;
144 }
145
146 /*
147 * Check the alarm list to see if a Type A alarm of this message number exists.
148 *
149 * return 1 if so and 0 otherwise.
150 *
151 */
152 int check_number_a_exists(int num){
153   int     status;
154   alarm_t *next, **last;
155
156   last = &alarm_list;
157   next = *last;
158   while (next != NULL) {
159     if(next->number == num && next->request_type == TYPE_A){
160       return 1;
161     }
162
163     last = &next->link;
164     next = next->link;
165   }
166
167   return 0;
168 }
169
170 /*
171 * Check the alarm list to see if an alarm with this type already exists.
```

```
172 * Takes the message type and request type as parameters
173 *
174 * return 1 if so and 0 otherwise.
175 */
176 int check_dup(int type, int req){
177   int     status;
178   alarm_t *next, **last;
179 
180   last = &alarm_list;
181   next = *last;
182   while (next != NULL) {
183     if(next->type == type && next->request_type == req){
184       return 1; // it exists already
185     }
186 
187     last = &next->link;
188     next = next->link;
189   }
190   return 0; // It doesn't exist.
191 }
192 
193 /*
194 * Check the alarm list to see if an alarm with this number already exists.
195 * Takes the message number and request type as parameters
196 *
197 * return 1 if so and 0 otherwise.
198 *
199 */
200 int check_dup_2(int num, int req){
201   int     status;
202   alarm_t *next, **last;
203 
204   last = &alarm_list;
205   next = *last;
206   while (next != NULL) {
207     if(next->number == num && next->request_type == req){
208 
209       return 1; // it exists already
210     }
211 
212     last = &next->link;
213     next = next->link;
214   }
215   return 0; // It doesn't exist.
216 }
217 
218 /*
219 * Removes a Type A alarm of the specified message number from the alarm list
220 *
221 * Returns the message type of the alarm that was just removed from alarm list
222 *
223 * Requires Mutex for alarm list to prevent writing while readers are reading
224 * Mutex is needed because this method removes from (writes to) the alarm list
225 */
226 int remove_alarm(int number){
227   alarm_t **last, *next;
228   int val = 0;
```

```
229
230   /*
231   * LOCKING PROTOCOL:
232   *
233   * This routine requires that the caller have locked the
234   * alarm_mutex!
235   */
236   last = &alarm_list;
237   next = *last;
238
239   /*
240   * If list is empty, return 0;
241   */
242   if (next == NULL)
243     return val;
244
245   while (next != NULL){
246
247     /*
248     * if we find the alarm within the list, delete it.
249     */
250     if (next->number == number && next->request_type == TYPE_A){
251       val = next->type;
252       *last = next->link;
253       free(next);
254       break; // remove the thread the Alarm.
255     }
256
257     last = &next->link;
258     next = next->link;
259   }
260
261   return val;
262
263 }
264
265 /*
266 * Removes a type B alarm request responsible for type A alarms with the
267 * specified type
268 *
269 * Requires Mutex for alarm list to prevent writing while readers are reading
270 * Mutex is needed because this method removes from (writes to) the alarm list
271 */
272 void remove_alarm_B(int type){
273   alarm_t **last, *next;
274   /*
275   * LOCKING PROTOCOL:
276   *
277   * This routine requires that the caller have locked the
278   * alarm_mutex!
279   */
280   last = &alarm_list;
281   next = *last;
282
283   while (next != NULL){
284     /*
285     * if we find the alarm within the list, delete it.
```

```
286     */
287     if (next->request_type == TYPE_B && next->type == type){
288       *last = next->link;
289       free(next);
290       break; // remove the thread the Alarm.
291     }
292
293     last = &next->link;
294     next = next->link;
295   }// End while
296 }
297
298
299 /*
300 * Removes a type C alarm request responsible for cancelling alarms with the
301 * specified alarm number type
302 *
303 * Requires Mutex for alarm list to prevent writing while readers are reading
304 * Mutex is needed because this method removes from (writes to) the alarm list
305 */
306 void remove_alarm_C(int number){
307   alarm_t **last, *next;
308   /*
309   * LOCKING PROTOCOL:
310   *
311   * This routine requires that the caller have locked the
312   * alarm_mutex!
313   */
314   last = &alarm_list;
315   next = *last;
316
317   while (next != NULL){
318     /*
319     * if we find the alarm within the list, delete it.
320     */
321     if (next->request_type == TYPE_C && next->number == number){
322       *last = next->link;
323       free(next);
324       break; // remove the thread the Alarm.
325     }
326
327     last = &next->link;
328     next = next->link;
329   }// End while
330 }
331
332 /*
333 * Insert alarm entry on list, in order of message number.
334 *
335 * Requires Mutex for alarm list to prevent writing while readers are reading
336 * Mutex is needed because this method removes from (writes to) the alarm list
337 */
338 void alarm_insert (alarm_t *alarm){
339   int status;
340   alarm_t **last, *next;
341
342   /*
```

```
343   * LOCKING PROTOCOL:
344   *
345   * This routine requires that the caller have locked the
346   * alarm_mutex!
347   */
348   last = &alarm_list;
349   next = *last;
350   while (next != NULL) {
351
352     /*
353     * Replace existing alarm or insert the new alarm arranged by message number.
354     *
355     * If the alarm is a type B request, it will be inserted in the front as it
356     * has a Message Number of 0.
357     * If the alarm is a type C request, it will be inserted along Type A's.
358     */
359     if (next->number == alarm->number && alarm->request_type == TYPE_A){//A.3.2.2
360
361       // swap the nodes (Replacement)
362       alarm->link = next->link;
363       alarm->prev_type = next->type;
364       *last = alarm;
365       free(next);
366       printf("Type A Replacement Alarm Request With Message Number (%d) "
367       "Received at <%d>: <A>\n", alarm->number, (int)time(NULL));
368       break; // Add the Alarm.
369
370     }else if (next->number > alarm->number){
371
372       alarm->link = next;
373       *last = alarm;
374       break; // Add the Alarm.
375
376      }
377
378     last = &next->link;
379     next = next->link;
380   }
381   /*
382   * If we reached the end of the list, insert the new alarm
383   * there.  ("next" is NULL, and "last" points to the link
384   * field of the last item, or to the list header.)
385   */
386   if (next == NULL) {
387     *last = alarm;
388     alarm->link = NULL;
389   }
390 }
391
392 ///THREAD STUFF
393
394 /*
395 * insert thread id into the thread list in order of Message Type
396 *
397 */
398 void insert_thread(thread_t *thread){
399
```

```
400   thread_t **last, *next;
401
402   last = &thread_list;
403   next = *last;
404   while (next != NULL) {
405
406     /*
407     * insert the thread id into the thread list
408     */
409     if (next->type > thread->type){
410
411       thread->link = next;
412       *last = thread;
413       break; // Add the Alarm.
414
415      }
416
417     last = &next->link;
418     next = next->link;
419   }
420   /*
421   * If we reached the end of the list, insert the new thread
422   * there.  ("next" is NULL, and "last" points to the link
423   * field of the last item, or to the list header.)
424   */
425   if (next == NULL) {
426     *last = thread;
427     thread->link = NULL;
428   }
429 }
430
431 /*
432 * ittereate through the thread list and terminate threads
433 * of MessageType(Type)
434 * also removes it from the thread list
435 *
436 * Note that every thread is allowed to complete its routine before terminatied
437 * this is to avoid the mutex being locked and not having a way to unlock it
438 *
439 */
440 void terminate_thread(int type){
441   thread_t **last, *next;
442   last = &thread_list;
443   next = *last;
444
445   while (next != NULL){
446
447     /*
448     * if we find the thread within the list, delete it.
449     */
450     if (next->type == type){
451
452       int success = pthread_cancel(next->thread_id); //terminate that thread
453       if(success != 0) // checks if the thread was successfuly terminated
454         err_abort (success, "thread was not canceled");
455
456       *last = next->link;
```

```
457       free(next);
458       break; // remove the thread.
459
460     }
461     last = &next->link;
462     next = next->link;
463   }// End while
464 }
465
466 /*
467 *
468 * Check the thread list to see if there are any useless threads in the list.
469 * A thread is considered useless if there are no Type A alarms of its message
470 * type available to be printed
471 *
472 * terminate the thread if such thread exists and return 1. If no such thread
473 * exists, return 0
474 *
475 */
476 int check_useless_thread(){
477   thread_t **last, *next;
478
479   last = &thread_list;
480   next = *last;
481
482   /*
483   * loop throught the thread list and check the alarm list for Type A alarms
484   * that have the same message type as the thread. if at least 1 exists, return
485   * 0.
486   */
487   while(next != NULL){
488
489     if(check_type_a_exists(next->type) == 0){
490       terminate_thread(next->type);
491       return 1;
492     }
493     next = next->link;
494   }
495
496   return 0;
497 }
498
499 /*
500 * When debug mode is activated, prints out the contents of the alarm list as
501 * well as the thread list. Also prints out the values for the semaphore
502 * variables (during the time debug is called) used for mutual exclusion.
503 *
504 */
505 void debug(){
506
507   if (debug_flag){
508     display_lists();
509     printf("Ready = %d read_count = %d writing = %d\n\n", ready,
510     read_count, writing );
511   }
512
513 }
```

```
514 /*************************END HELPER CODE*************************/////////
515
516
517 /* READER
518 *
519 * TYPE B CREATED THREAD (periodic display thread).
520 * responsible for periodically looking up a Type A alarm request with a
521 * Message Type in the alarm list, then printing, every Time seconds.
522 *
523 * A3.4
524 */
525 void *periodic_display_thread(void *arg){
526   alarm_t *alarm = alarm_list;
527   int status, flag;
528
529   int *arg_pointer = arg;
530   int type = *arg_pointer; // parameter passed by the create thread call
531
532   /*
533   * Loop forever, processing Type A alarms of specified message type.
534   * The alarm thread will be disintegrated when the process exits.
535   */
536
537   while (1){
538
539     while(ready > 0){
540       // wrtiter is ready to trite so don't do anything
541     }
542
543     pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); //disable cancellation
544     while (alarm_list == NULL){
545       ///// ACHTUNG! /////
546       pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL); //enable cancellation
547       pthread_testcancel();
548     }
549
550     /////
551     if (flag == 1){
552       alarm = alarm_list; // go back to the beginning
553       flag = 0;
554     }
555     if (alarm->link == NULL){
556       flag = 1; // go back to the beginning of the list
557     }
558     /////
559     sem_wait(&sem_mutex);
560     read_count++;
561     sem_post(&sem_mutex);
562
563     if(alarm->type != type && alarm->request_type == TYPE_A){ //A.3.4.2
564       /*
565       * check if its type has changed. if its type has changed from a different
566       * one, notify the user that an alarm with the specified type which
567       * previously had a different type has been assigned.
568       */
569       if(check_prev(alarm) == 1 && alarm->prev_type == type){
570         if(alarm->expo == 0){ // check if alarm change has been acknowledged
```

```
571           printf("Alarm With Message Type (%d) Replaced at <%d>: "
572           "<Type A>\n", alarm->type, (int)time(NULL)); // A.3.4.2
573           alarm->expo = 1; // alarm exposed (change acknowledged)
574         }
575       }
576     }
577
578
579     if (alarm->type == type && alarm->request_type == TYPE_A && alarm->first == 1){
580         alarm->time = time (NULL) + alarm->seconds;
581       alarm->first = 0;
582     }
583
584     if(alarm->type == type && alarm->request_type == TYPE_A && time(NULL) >= alarm->time){
    //A.3.4.1
585       // PRINT MESSAGE // A.3.4.1
586       printf("Alarm With Message Type (%d) and Message Number"
587       " (%d) Displayed at <%d>: <Type A> : ",
588       alarm->type, alarm->number, (int)time(NULL) );
589       printf ("\"%s\"\n", alarm->message);
590       alarm->time = time (NULL) + alarm->seconds;
591     }
592       alarm = alarm->link; // go to the next node on the list
593
594     sem_wait(&sem_mutex);
595     read_count--;
596     sem_post(&sem_mutex);
597
598     /* used to avoid potential deadlock from thread termination
599     */
600     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL); //enable cancellation
601     pthread_testcancel(); // set a cancellation point
602   }// End While(1)
603 }
604
605 /*WRITER
606 *
607 * The alarm thread's start routine.
608 *
609 * An initial thread which is responsible for looping through the
610 * alarm list and checking the status of each type A alarm, as well as performing
611 * type B or C requests as they are inserted.
612 *
613 * A3.3
614 */
615 void *alarm_thread (void *arg){
616
617   alarm_t **last, *next;
618   int status;
619
620   /*
621   * Loop forever, processing commands. The alarm thread will
622   * be disintegrated when the process exits.
623   */
624
625   while (1){
626
```

```
627     /*
628     * If a new alarm hasnt been added, wait until a new alarm is
629     * added
630     */
631
632      while (insert_flag == 0) {
633        //busy wait
634      }
635
636     /*
637     * when a new alarm has been inserted, loop through the alarm list and find
638     * and process the alarm
639     */
640     last = &alarm_list;
641     next = *last;
642
643     while(next != NULL){
644
645       /*
646       * upon finding a new type A alarm, checks if there exists a useless
647       * periodic display thread and terminates it if such thread exists.
648       */
649       if(next->request_type == TYPE_A){ // A.3.3.1
650         if(next->is_new == 1){
651           next->is_new = 0; // alarm is no longer new
652
653           status = check_useless_thread(); // remove possible useless threads
654           if (status == 1){ // then remove it from the alarm list
655             ////////////////
656             ready++; // the writer is ready to use the alarm list
657             while(read_count > 0 || writing > 0){
658               // busy waits for readers to be done
659             }
660             status = sem_wait(&rw_sem);
661             if(status != 0)
662               err_abort(status, "rw_sem wait");
663             writing++; // writer has control of the data structure
664             /* critical section */
665
666             remove_alarm_B(next->prev_type); // remove it from the list
667             debug();
668             writing--;
669             status = sem_post(&rw_sem);
670             if(status != 0)
671               err_abort(status, "rw_sem post");
672             ready--;
673             ////////////////
674           }else{
675             debug();
676           }
677           break;
678         }
679       }// END TYPE A ***************************///////
680
681       /*
682       * upon finding a new type B alarm, creates a periodic display thread
683       * responsible for printing messages its specified type
```

```
684       */
685       else if(next->request_type == TYPE_B){ // A.3.3.2
686         if(next->is_new == 1){
687
688           next->is_new = 0; // alarm is no longer new
689           thread_t *thrd;
690           pthread_t thread;
691
692
693           thrd = (thread_t*)malloc (sizeof (thread_t)); //allocate thread struct
694           if (thrd == NULL)
695           errno_abort ("Allocate Thread");
696
697           /* create a thread for periodically printing messages
698           *  pass message type as an argument
699           */
700           status = pthread_create(&thread, NULL, periodic_display_thread,
701           &next->type);
702           if (status != 0)
703             err_abort (status, "Create alarm thread"); // A.3.3.2 (a)
704           thrd->type = next->type; // set the attributes for the thread struct
705           thrd->thread_id = thread;
706
707           insert_thread(thrd);
708
709           printf("Type B Alarm Request Processed at <%d>: New Periodic Dis"
710           "play Thread With Message Type (%d) Created.\n", (int)(time(NULL)),
711           next->type ); // A.3.3.2 (b)
712           debug();
713           break;
714         }
715       }// END TYPE B ***************************///////
716
717       /*
718       * upon finding a new type C alarm, removes the alarm of the message
719       * number specified by the Type C alarm from the alarm list.
720       *
721       * if there are no more alarm requests in the alarm list the same type as
722       * the one that was just removed, terminate the periodic display thread
723       * responsible for displaying those messages.
724       *
725       * This is the only part of the  alarm thread that writes to the alarm list
726       */
727       else if(next->request_type == TYPE_C){ //A.3.3.3
728         int val;
729         if(next->is_new == 1){
730           next->is_new == 0; // alarm is no longer new
731
732
733           ready++; // the writer is ready to use the alarm list
734           while(read_count > 0 || writing > 0){
735             // busy waits for readers to be done
736           }
737           status = sem_wait(&rw_sem);
738           if(status != 0)
739             err_abort(status, "rw_sem wait");
740           writing++; // writer has control of the data structure
```

```
741
742
743           val = remove_alarm(next->number); // A.3.3.3 (a)
744           if(val != 0){ // A.3.3.3 (c)
745
746             remove_alarm_C(next->number);// remove alarm from the alarm list
747             printf("Type C Alarm Request Processed at <%d>: Alarm Request"
748             " With Message Number (%d) Removed\n", (int)(time(NULL)),
749             next->number);
750           }
751
752           if(check_type_a_exists(val) == 0){ // A.3.3.3 (b)
753
754             terminate_thread(val); // terminate the thread
755             remove_alarm_B(val); // remove the B alarm from alarm list
756             remove_alarm_C(next->number);// remove alarm from the alarm list
757
758             printf("No More Alarm Requests With Message Type (%d):"
759             " Periodic Display Thread For Message Type (%d)"
760             " Terminated.\n", val, val); // A.3.3.3 (d)
761
762           }debug();
763
764           writing--;
765           status = sem_post(&rw_sem);
766           if(status != 0)
767             err_abort(status, "rw_sem post");
768           ready--;
769         }
770         break;
771       }// END TYPE C ************************///////
772       next = next->link; //go to the next node
773     } // End list loop
774     insert_flag = 0; // finished looping and processd new alarm
775   }
776 }
777
778 /*WRITER
779 * Parses inputs as specified in assaignment 3 outline
780 *
781 * Creates 3 different alarm requests (Type A - C) and inserts them into the
782 * alarm list. THe alarm thread then processes these alarm requests as they already
783 * inserted
784 */
785 int main (int argc, char *argv[]){
786   int status;
787   char line[128];
788   char deb[8];
789   alarm_t *alarm;
790   thread_t *thrd;
791   pthread_t thread;
792
793   status = sem_init(&rw_sem, 0, 1); // initialize reader writer Semaphore
794   if(status != 0)
795     err_abort(status, "Create READ-WRITE Semaphore");
796
797   status = sem_init(&sem_mutex, 0, 1); // initialize reader writer Semaphore
```

```
798   if(status != 0)
799     err_abort(status, "Create Mutex Semaphore");
800
801   /*
802   * Create the initial thread responsible for looping through the alarm list
803   * and performing operations depening on the request type
804   *
805   * leaving the argument "NULL" would also imply that the initial thread
806   */
807   status = pthread_create (&thread, NULL, alarm_thread, NULL);
808   if (status != 0) err_abort (status, "Create alarm thread");
809
810   while (1) {
811     printf ("alarm> ");
812     if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
813     if (strlen (line) <= 1) continue;
814     alarm = (alarm_t*)malloc (sizeof (alarm_t));
815     if (alarm == NULL) errno_abort ("Allocate alarm");
816
817     /*
818     * Parse input line into seconds (%d) and a message
819     * (%64[^\n]), consisting of up to 64 characters
820     * separated from the seconds by whitespace.
821     *
822     * Checks what type of alarm / message is being entered.
823     *
824     */
825     /************************TYPE A************************/
826     if (sscanf (line, "%d Message(%d, %d) %128[^\n]",
827     &alarm->seconds, &alarm->type, &alarm->number, alarm->message) == 4 &&
828     alarm->seconds > 0 && alarm->number > 0 && alarm->type > 0){ // A.3.2.1
829
830       alarm->time = time (NULL) + alarm->seconds;
831       alarm->request_type = TYPE_A;
832       alarm->is_new = 1;
833       alarm->prev_type = alarm->type;
834       alarm->first = 1;
835
836       ready++; // the writer is ready to use the alarm list
837       while(read_count > 0 || writing > 0){
838         // wait for readers or writers to finish
839       }
840       status = sem_wait(&rw_sem);
841       if(status != 0)
842         err_abort(status, "rw_sem wait");
843       writing++; // writer has control of the data structure
844       /*
845       * Insert the new alarm into the list of alarms, CRITICAL SECTION
846       */
847       alarm_insert (alarm);
848       printf("Type A Alarm Request With Message Number <%d> Received at"
849       " time <%d>: <Type A>\n", alarm->number, (int)time(NULL));
850
851       insert_flag = 1; // a new alarm has been inserted
852
853       writing--;
854       status = sem_post(&rw_sem);
```

```
855       if(status != 0)
856         err_abort(status, "rw_sem post");
857       ready--;
858
859     }
860     /*********************END TYPE A************************/
861     /************************TYPE B************************/
862     else if (sscanf(line,"Create_Thread: MessageType(%d)",&alarm->type) == 1
863     && alarm->type > 0){ // A.3.2.3 - A.3.2.5
864
865       int exists = check_type_a_exists(alarm->type);
866       int dup = check_dup(alarm->type, TYPE_B);
867       /*
868       * Creates a Type B alarm that is then inserted into the alarm list.
869       * Does not allow for duplicate type B alarms.
870       * Only creates one if there exists a type A alarm of type B's
871       * Message Type.
872       */
873
874       if(exists == 0){ // A.3.2.3
875
876         printf("Type B Alarm Request Error: No Alarm Request With Message Type"
877         "(%d)!\n", alarm->type);
878         free(alarm); // deallocate alarm that isn't used
879
880       }else if(dup == 1){ // A.3.2.4
881         // May need to fix as there is confusion between "Number" and "Type"
882         printf("Error: More Than One Type B Alarm Request With"
883           " Message Type (%d)!\n", alarm->type );
884         free(alarm); // deallocate alarm that isn't used
885
886       }else if(exists == 1 && dup == 0){ //A.3.2.5
887
888         alarm->request_type = TYPE_B;
889         alarm->is_new = 1;
890
891         ready++; //
892         while(read_count > 0 || writing > 0){
893           // busy wait
894         }
895         status = sem_wait(&rw_sem);
896         if(status != 0)
897           err_abort(status, "rw_sem wait");
898         writing++; // writer has control of the data structure
899
900         /*
901         * Insert the new alarm into the list of alarms
902         * Insert the new thread into the list of threads
903         */
904         alarm_insert (alarm);
905         printf("Type B Create Thread Alarm Request With Message Type (%d)"
906         " Inserted Into Alarm List at <%d>!\n", alarm->type, (int)time(NULL));
907         insert_flag = 1; // a new alarm has been inserted
908
909         writing--;
910         status = sem_post(&rw_sem);
911         if(status != 0)
```

```
912           err_abort(status, "rw_sem post");
913         ready--;
914       }
915     }
916     /*********************END TYPE B************************/
917     /*************************TYPE C************************/
918     else if (sscanf (line, "Cancel: Message(%d)", &alarm->number) == 1 &&
919     alarm->number > 0 ){ //
920
921       int exists = check_number_a_exists(alarm->number);
922       int dup2 = check_dup_2(alarm->number, TYPE_C);
923
924       /*
925       * Creates a Type C alarm that is then inserted into the alarm list.
926       * Does not allow for duplicate type alarms.
927       * Only creates one if there exists a type A alarm of type C's Message
928       * Type.
929       */
930
931       if (exists == 0){ // A.3.2.6
932
933         printf("Error: No Alarm Request With Message"
934           " Number (%d) to Cancel!\n", alarm->number );
935         free(alarm);
936
937       }else if (dup2 == 1){ // A.3.2.7
938
939         printf("Error: More Than One Request to Cancel Alarm Request With"
940           " Message Number (%d)!\n", alarm->number);
941         free(alarm);
942
943       }else if (exists == 1 && dup2 == 0 ){ // A.3.2.8
944
945         alarm->request_type = TYPE_C;
946         alarm->is_new = 1;
947
948
949         ready++;
950         while(read_count > 0 || writing > 0){
951           // busy wait
952         }
953         status = sem_wait(&rw_sem);
954         if(status != 0)
955           err_abort(status, "rw_sem wait");
956         writing++; // writer has control of the data structure
957
958         /*
959         * Insert the new alarm into the list of alarms.
960         */
961         alarm_insert (alarm);
962         printf("Type C Cancel Alarm Request With Message Number (%d)"
963           " Inserted Into Alarm List at <%d>: <Type C>\n", alarm->number,
964               (int)time(NULL));
965
966         insert_flag = 1; // a new alarm has been inserted
967
968         writing--;
```

```
969         status = sem_post(&rw_sem);
970         if(status != 0)
971           err_abort(status, "rw_sem post");
972         ready--;
973       }
974     }
975     /*********************END TYPE C************************/
976     else if (sscanf(line,"%s", deb) && strcmp("debug",deb) == 0){ // debugging
977       if (debug_flag == 0){
978         printf("**DEBUG MODE ENGAGED**\n");
979         debug_flag = 1;
980       }else{
981         printf("**DEBUG MODE DISENGAGED**\n");
982         debug_flag = 0;
983       }
984 
985     }
986     else{
987       fprintf (stderr, "Bad command\n");
988       free (alarm);
989     }
990   }// end while
991 }
992 
```