```c
1 /*
2 * New_Alarm_Cond.c
3
4    AUTHORS **
5   Francis Okoyo 213811336
6   Tyler Noble 212270369
7   Adham El-Shafie 212951018
8   Lindan Thillanayagam 213742176
9
10
11 * This is an enhancement to the alarm_cond.c program, which
12 * used only a mutex and condition variables to synchronize acces
13 * to a list of alarms. This versions does the job by only using
14 * semaphores
15 */
16 #include <pthread.h>
17 #include <time.h>
18 #include "errors.h"
19 #include <semaphore.h>
20
21 /*
22 * The "alarm" structure now contains the time_t (time since the
23 * Epoch, in seconds) for each alarm, so that they can be
24 * sorted. Storing the requested number of seconds would not be
25 * enough, since the "alarm thread" cannot tell how long it has
26 * been on the list.
27 */
28 typedef struct alarm_tag {
29   struct alarm_tag    *link;
30   int                 seconds;
31   time_t              time;   /* seconds from EPOCH */
32   char                message[128];
33
34   /******* new additions to the alarm_tag structure ********/
35   int                 type; //identifies the message type ( type >= 1 )
36   int                 prev_type; // previous message type
37   int                 is_new; // 1 == "new" and 0 == "old/not new"
38   int                 number; /* Message Number */
39   int                 request_type; // TypeA == 1 TypeB == 2 TypeC == 3
40   int                 expo; // signifies that the type change was confirmed
41   /******************end new additions***************/
42 } alarm_t;
43
44 /*
45 *
46 * Thread structure used to keep a linked list of thread id's which will be
47 * organized by their message type.
48 *
49 * This is a replacement of the sparce matrix used in the previous project
50 * "New_Alarm_Mutex.c". This is a lot more efficient as it does not Allocate
51 * an unneccesary amount of space. we do loose the O(1) acces time though.
52 */
53 typedef struct thread_tag { // NEW STRUCT
54   struct thread_tag   *link;
55   pthread_t           thread_id;
56   int                 type;
57   int                 number;
```

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

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

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

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

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

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

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

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

```
514 * variables (during the time debud is called) used for mutual exclusion.
515 *
516 */
517 void debug(){
518
519   if (debug_flag){
520     display_lists();
521     printf("Ready = %d read_count = %d writing = %d\n\n", ready, read_count, writing );
522   }
523
524 }
525 /************************END HELPER CODE****************************///////////
526
527
528 /* READER
529 *
530 * TYPE B CREATED THREAD (periodic display thread).
531 * responsible for periodically looking up a Type A alarm request with a
532 * Message Type in the alarm list, then printing, every Time seconds.
533 *
534 * A3.4
535 */
536 void *periodic_display_thread(void *arg){
537   alarm_t *alarm = alarm_list;
538   int status, flag;
539
540   int *arg_pointer = arg;
541   int type = *arg_pointer; // parameter passed by the create thread call
542
543   //data
544   char r_message[128]; int r_type, r_sec, r_num, r_req_type;
545   /////
546
547   /*
548   * Loop forever, processing Type A alarms of specified message type.
549   * The alarm thread will be disintegrated when the process exits.
550   */
551
552   while (1){
553
554     while(ready > 0){
555       // wrtiter is ready to trite so don't do anything
556     }
557
558     pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); //disable cancellation
559     while (alarm_list == NULL){
560       ///// ACHTUNG! /////
561       pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL); //enable cancellation
562       pthread_testcancel();
563     }
564
565     /////
566     if (flag == 1){
567       alarm = alarm_list; // go back to the beginning
568       flag = 0;
569     }
570     if (alarm->link == NULL){
```

```
571        flag = 1; // go back to the beginning of the list
572      }
573      /////
574
575      read_count++;
576      /* read all the importnat data */
577      r_type = alarm->type;
578      r_num = alarm->number;
579      r_sec = alarm->seconds;
580      r_req_type = alarm->request_type;
581      strcpy(r_message, alarm->message);
582      //////////////////////////////////
583      if(r_type != type && r_req_type == TYPE_A){ //A.3.4.2
584        /*
585         * check if its type has changed. if its type has changed from a different
586         * one, notify the user that an alarm with the specified type which
587         * previously had a different type has been assigned.
588         */
589        if(check_prev(alarm) == 1 && alarm->prev_type == type){
590          if(alarm->expo == 0){ // check if alarm change has been acknowledged
591            printf("Alarm With Message Type (%d) Replaced at <%d>: "
592            "<Type A>\n", r_type, (int)time(NULL)); // A.3.4.2
593            alarm->expo = 1; // alarm exposed (chanhe acknowledged)
594          }
595        }
596      }
597      alarm = alarm->link; // go to the next node on the list
598      read_count--;
599
600      /* used to avoid potential deadlock from thread termination
601       */
602      pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL); //enable cancellation
603      pthread_testcancel(); // set a cancellation point
604
605      if(r_type == type && r_req_type == TYPE_A){ //A.3.4.1
606        delay(r_sec);
607        // PRINT MESSAGE // A.3.4.1
608        printf("Alarm With Message Type (%d) and Message Number"
609        " (%d) Displayed at <%d>: <Type A> : ",
610        r_type, r_num, (int)time(NULL) );
611        printf ("\"%s\"\n", r_message);
612      }
613   }// End While(1)
614 }
615
616 /*WRITER
617  *
618  * The alarm thread's start routine.
619  *
620  * An initial thread which is responsible for looping through the
621  * alarm list and checking the status of each type A alarm, as well as performing
622  * type B or C requests as they are inserted.
623  *
624  * A3.3
625  */
626 void *alarm_thread (void *arg){
627
```

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

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

```
742            next->is_new == 0; // alarm is no longer new

743

744

745            ready++; // the writer is ready to use the alarm list
746            while(read_count > 0 || writing > 0){
747               // busy waits for readers to be done
748            }
749            status = sem_wait(&rw_sem);
750            if(status != 0)
751               err_abort(status, "rw_sem wait");
752            writing++; // writer has control of the data structure

753

754

755            val = remove_alarm(next->number); // A.3.3.3 (a)
756            if(val != 0){ // A.3.3.3 (c)

757

758               remove_alarm_C(next->number);// remove alarm from the alarm list
759               printf("Type C Alarm Request Processed at <%d>: Alarm Request"
760               " With Message Number (%d) Removed\n", (int)(time(NULL)),
761               next->number);
762            }

763

764            if(check_type_a_exists(val) == 0){ // A.3.3.3 (b)

765

766               terminate_thread(val); // terminate the thread
767               remove_alarm_B(val); // remove the B alarm from alarm list
768               remove_alarm_C(next->number);// remove alarm from the alarm list

769

770               printf("No More Alarm Requests With Message Type (%d):"
771               " Periodic Display Thread For Message Type (%d)"
772               " Terminated.\n", val, val); // A.3.3.3 (d)

773

774            }debug();

775

776            writing--;
777            status = sem_post(&rw_sem);
778            if(status != 0)
779               err_abort(status, "rw_sem post");
780            ready--;
781         }
782       break;
783      }// END TYPE C ***************************///////
784     next = next->link; //go to the next node
785   } // End list loop
786   insert_flag = 0; // finished looping and processd new alarm
787  }
788 }

789

790 /*WRITER
791 * Parses inputs as specified in assaignment 3 outline
792 *
793 * Creates 3 different alarm requests (Type A - C) and inserts them into the
794 * alarm list. THe alarm thread then processes these alarm requests as they already
795 * inserted
796 */
797 int main (int argc, char *argv[]){
798   int status;
```

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

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

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

```
970                    (int)time(NULL));
971
972          insert_flag = 1; // a new alarm has been inserted
973
974          writing--;
975          status = sem_post(&rw_sem);
976          if(status != 0)
977            err_abort(status, "rw_sem post");
978          ready--;
979        }
980      }
981    /*********************END TYPE C************************/
982    else if (sscanf(line,"%d", &status) == 1 && status == 15){ // debugging
983      if (debug_flag == 0){
984        printf("**DEBUG MODE ENGAGED**\n");
985        debug_flag = 1;
986      }else{
987        printf("**DEBUG MODE DISENGAGED**\n");
988        debug_flag = 0;
989      }
990
991    }
992    else{
993      fprintf (stderr, "Bad command\n");
994      free (alarm);
995    }
996  }// end while
997 }
998
```