EECS 3221: POSIX Threads & Semaphores Assignment Report

Dinesh Kalia
(213273420 - dinesh49)
&
Camillo John (CJ) D'Alimonte
(212754396 - cjdal34)
&
Alistair Scheuhammer
(213043609 - siggy)
&
Ben Zhou
(213383823 - bozon92)

December 7, 2015

Contents

1 Abstract

The use of threads to address parallelism and process synchronization are of paramount significance to the field of shared-resource multiprocessor architecture. A prior case study sought to use mutex locks in POSIX threads (or Pthreads) to program an alarm. The goal was to implement multi-threads that could process incoming alarm requests, as well as print a machine time and alarm message after certain time intervals. This case study yielded valuable insight into the way that mutexes in multi-threads restrict access to shared memory, but also revealed a rather serious limitation on their flexibility and reliability.

After becoming familiar with mutexes, it becomes apparent that they are binary constructs; they only allow one thread to have access to a set of shared resources at a time, while locking out access to the others. While this may protect those shared resources, it does not permit the possibility of simultaneous access to multiple threads, which is where the idea of a semaphore becomes relevant. A mutex gives access to one thread and locks out the others, but a semaphore can simultaneously grant access to as many threads as the programmer specifies. Whereas the previous case study explored the properties of mutexes, this study goes a step further within the same context and analyzes how semaphores can be implemented to achieve the functionalities of the specified alarm program.

2 Discussion

2.1 Design and Implementation

Throughout the course of implementing the assignment specifications, many design decisions had to be carefully considered. Several of these choices were made in order to ease the process of implementation, such as ignoring alarm requests with 0 or lower sleep time. Another such decision was to use the first letter of the user input to parse requests into “cancel” and “non-cancel” requests, as any cancel request would begin with a “C”.

Other decisions were issues of logic and intuition, like the choice to accept only requests with positive IDs, as it would not make much sense to accept 0-ID and negative-ID requests. Another common-sense judgment was to order the alarms in the list from lowest ID to highest, instead of the other way around.

There was also an effort to minimize foreseeable problems. For example, it was decided that when a cancel request is received, the alarm ID is set to the ID of the alarm to be removed; this reduces the overall work done by replacing the request to be removed with the corresponding cancel request. In order to be safe, the mutex for the alarm list was set to lock whenever a writer thread was executing. And because there can be any number of reader threads, it seemed unnecessary to lock the mutex for reader threads, which reduced the work even further. To add even another layer of caution, cancel requests were initialized with dummy values in order to prevent the program from possibly trying to access the values of a cancel request. This helped to avoid having to alter the debugging code listing the contents of the alarm list. Another preventative measure was to have the writer threads decrement the writer counter after unlocking the mutex. This avoided the possibility of another thread activating as the writer count decremented, with that thread then trying to interact with a locked list.

While there was ample consideration of the aforementioned issues, the bulk of the design decisions were relevant to how the program actually functioned. To this end, the main thread (writer), periodic display threads (readers), and the process of cancelling a request (a writer occurring in the alarm thread) were all initialized to be inactive until other threads were no longer looking at the alarm list. A given writer can only access the alarm list if the other writer is not doing so, and while there are no readers. Meanwhile, an unlimited number of readers can simultaneously access the list, as long as there are no writers present. The indicator variables "want_to_write" and "want_to_cancel" were implemented to avoid starvation, so if a writer thread is waiting, then readers cannot execute until the writers have finished. Furthermore, a request cancellation takes priority over the addition of a new request. This means that if both writers happen to reach their critical sections simultaneously, the main thread adding a new request will remain inactive until the thread cancelling a request has finished doing so. It seemed more reasonable to give cancelling priority over adding, as a newly-added request could have the same ID as the request to be cancelled, which would be problematic.

Another functional design decision was to have the alarm thread wait differently from the periodic display threads. This seemed logical because the alarm thread should execute whenever the alarm list has been changed, like when main finishes adding a new request. This feature was implemented by having alarm thread wait until a variable "signal" was set to 1 (which would only happen after the main thread finishes executing), then resetting it to 0 for the next time. And because cancellation has higher precedence than addition, the program will not experience a writer activating once main finishes.

While searching for the alarm to be processed, the alarm thread tracks both the current and previous alarm, to account for the possibility that the new request is a cancel request. To avoid complications when cancelling the alarm with the lowest ID (in which case the alarm would be immediately found at the top of the list), the pointer to the previous alarm is initialized to NULL. In the event that the pointer is still NULL when processing the cancel request, rather than changing the previous alarm's link (which will cause a segmentation fault if the code does not check that last is NULL or not) the code will simply move the head of the alarm list.

Also, when creating a new periodic display thread, the program creates a copy of the displayed alarm request. The program can then pass the copy request to the periodic display thread, allowing the copy to be freely accessed and manipulated without changing any values beyond the periodic display thread. This is particularly useful for checking to see if the alarm with the same ID as the copy has been changed.

In the event that the periodic display thread cannot locate the alarm with the same ID as the copy in the list, then that alarm has been cancelled and the periodic display thread should free up the memory space of the alarm copy and exit. If it so happens that a periodic display thread executes after its corresponding alarm request has been replaced by a cancel request, but before the alarm thread actually deletes the request, then the periodic display thread will do the same whenever it discovers that the alarm whose ID matches that of the copy is a cancel request.

In this version of the alarm program, alarms no longer expire after a set time period, so the time parameter was changed to a creation parameter that recorded the time of the alarm's creation, measured in seconds since the Epoch. This proved to be useful in handling situations with two nearly-identical alarm requests. In this scenario, the user gives an alarm request similar to an existing request, and the only difference between the two requests is the creation times. This allows the periodic display thread to simply use the creation time in order to determine when a change has occurred.

After finding the appropriate alarm, the periodic display thread copies the "seconds" parameter into a variable called "sleep_time" before the decrementing the number of readers, then pauses by sleep_time seconds before executing again. This allowed the reader count to decrement before sleeping, which prevented writer threads from having to wait for reader threads to finish sleeping before being able to execute. However, because the alarms "seconds" parameter still represented the sleep time, it would have been improper to access it directly after decrementing the reader count. Doing so would still count as an example of reading, even if it was highly improbable that the "seconds" parameter value changed between decrementing the reader count and sleeping. Adding the intermediary step of copying the "seconds" parameter to a local variable prevented this problem.

2.2 Rectifying Challenges

There were many difficulties encountered in the process of implementing the alarm program. One of the major challenges was implementing the alarm via the "cond_wait", "cond_signal" and "cond_broadcast" methods provided by the pthreads API. Ultimately, the reason for this hindrance was that it was difficult to visualize how to properly set up the code within the limiting scope of these three methods. However, the task was made much easier with the use of personal int variables and while loops. Another, related cause of frustration came from the fact that threads blocked on "cond_wait" must lock the mutex when signaled. This was troublesome because the code was established to allow multiple simultaneous readers – if one of the readers locked upon activation, the others would stall. This error actually occurred in an earlier version of the program that used "cond_wait" and "cond_signal" to allow the alarm thread to execute whenever an alarm request was added to the list. Switching the int variable to "signal" fixed that problem.

There was also an early issue about differentiating between "cancel" requests and "non-cancel" requests. However, this was an easy matter to solve by simply adding a "cancel" parameter to the alarm, which would be set to one for cancel requests. An earlier version of the program was organized such that cancel requests were always placed at the front of the list (by setting the ID to 0 as mentioned above). In this case, the "cancel" parameter contained the ID of the request to be deleted. However, this decision was ultimately repealed because it was much easier to have cancel requests replace the request-to-be-cancelled in the list before the request is deleted.

Another problem was that, when entering an alarm request to replace an existing one, the program would start printing that particular request multiple times after every sleep. This happened because of the absence of a periodic display thread set up to not generate a new periodic display thread upon processing a command replacing an old one. This was remedied by adding the "new" parameter to alarm, which would be set to 0 only if the alarm was replacing a pre-existing alarm. That would allow anyone to easily check the value of the alarm (which was new in the alarm thread). If that value was set to 0, then the alarm thread would print that it processed the request and do nothing else, as the periodic display thread for that alarm ID would adjust its output accordingly.

Another, rather perplexing issue was when cancel requests were sometimes generating segmentation faults for unknown reasons. It was eventually determined that the issue was in the periodic display thread. The code that was purposed to find the appropriate alarm was designed to check if the current alarm matched the copy alarms ID before checking to see if the current alarm was NULL. This meant that if a user cancelled the only alarm in the list, the period display thread would attempt to compare the ID of a copy alarm to the ID of a NULL alarm, which would cause a segmentation fault. However, when the nature of the problem was identified, a simple switch of the two condition checks fixed the problem.

2.3 Testing

A test-driven development (TDD) approach was used when creating test cases for the program. A TDD approach is a specific software development process that relies on the repetition of a very short development cycle: first the developer writes an initially failing automated test case that defines a desired improvement or new function and then produces the minimum amount of code to pass that test. The final step is to refactor the new code to the acceptable standards.

Various types of test cases were used to ensure validity of the code including positive, negative and zero time arguments. An appropriate error message was produced when handling processes as shown in Appendix 5.2. By testing multiple alarm requests at a time, the program is able to treat each as a separate entity while guaranteeing the parallelism principle.

3 Conclusion

The combined use of semaphores and mutexes has facilitated the implementation of a correct alarm program. The program happens to yield the proper output, but it is important to remember that the operating system schedules threads randomly, so there is no guarantee as to the order in which threads will gain access to a resource. In striving for some semblance of order, it is necessary to properly use semaphores and mutexes to identify and enforce a standard thread execution order. A comprehensive understanding of how semaphores and mutexes cooperate in process synchronization yields valuable insights towards solving programming dilemmas of even higher complexities.

4 References

[1] D. R. Butenhof, *Programming with POSIX Threads.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. New York, NY, USA: John Wiley & Sons, Inc., 2012.

5 Appendix

5.1 *New_Alarm_Cond.c*

```
/*
 * New_Alarm_Cond.c
 *
 * This is an alteration to alarm_cond.c. Now, every time the main
 * thread receives an alarm request, a periodic display thread
 * activates, which prints the alarm request once every few
 * seconds (determined by the request), until the main thread
 * receives a corresponding "cancel" request.
 */
#include <pthread.h>
#include <time.h>
#include "errors.h"
// #define DEBUG

typedef struct alarm_tag
{
    struct alarm_tag    *link;
    int                 seconds;
    time_t              creation;
    // Time the alarm was created (in seconds from the Epoch).
    int     id;
    // Used to allow the user to specify which request to delete/change.
    char                message[64];
    int     cancel;
    // Indicates wether or not the request is a "cancel" request.
    int     new;
    // Used to indicate wether or not the request replaced another.
} alarm_t;

pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
alarm_t *alarm_list = NULL;
int current_id = 0; // The ID of the alarm to be processed.
int read_count = 0, write_count = 0;
// Counters of reader/writer threads for synchronization.
int want_to_write = 0, want_to_cancel = 0;
// Indicators for when something wants to write.
int signal = 0;
// Signal to indicate there is a new alarm request to process.

/*
 * Periodic Display Thread; each alarm requests activates one
 * which prints the alarm request every few seconds (as determined
 * by the request).
 */
void *periodic_display_thread(void *alarmArg)
{
 alarm_t *alarm_s, *alarm;
 int i, status, sleep_time;

 alarm_s = (alarm_t *) alarmArg; // Copy of the alarm to display.

 while(1)
 {
```

```
 while ((write_count > 0) || (want_to_write == 1) || (want_to_cancel == 1));
 // If there writer threads are running or who wish to run,
 // wait until they are done.

 read_count++;
 alarm = alarm_list;

 while ((alarm != NULL) && (alarm->id != alarm_s->id))
// Search the list until an ID matching the copy's is found,
// or NULL if the alarm with that ID has been deleted.
  alarm = alarm->link;

 if ((alarm == NULL) || (alarm->cancel == 1))
 // Alarm has been deleted or will be deleted soon.
 {
  printf("Display thread exiting at %d: %d Message(%d) %s\n",
    time (NULL), alarm_s->seconds, alarm_s->id, alarm_s->message);

  free(alarm_s);
  // Free up the memory used by the copy and
  // exit the alarm's periodic display thread.

  read_count--;

  break;
 }
 else if (alarm->creation != alarm_s->creation)

 // Alarm has been changed (if an alarm request is given for a
 // pre-existing ID, the creation time is guaranteed to be different
 // no matter what).
 {
   printf("Alarm changed at %d: %d Message(%d) %s\n",
     time (NULL), alarm->seconds, alarm->id, alarm->message);

   alarm_s->link = NULL; // The copy should not link to anything.
         alarm_s->seconds = alarm->seconds;
         // Replace the copy's values with the new alarm's.
         alarm_s->creation = alarm->creation;
         alarm_s->id = alarm->id;

         for (i = 0; i <= strlen(alarm->message); i++)
          alarm_s->message[i] = alarm->message[i];

         alarm_s->cancel = alarm->cancel;
 }

 printf("Alarm displayed at %d: %d Message(%d) %s\n",
    time (NULL), alarm->seconds, alarm->id, alarm->message);

  sleep_time = alarm->seconds;
 // Sleep time recorded before decrementing the reader count to allow for
 // sleeping to occur after reading.
  read_count--;

  sleep(sleep_time);
 }

}

/*
 * The alarm thread's start routine.
 */
void *alarm_thread (void *arg)
{
    alarm_t *last, *alarm, *alarmReceived;
    int i, status;
    pthread_t thread;

    /*
     * Loop forever, processing commands. The alarm thread will
     * be disintegrated when the process exits.
     */
    while (1)
    {
        while (signal == 0);
    // Wait until the main thread signals
    // that an alarm has been added to the list.

        signal = 0; // Reset the signal.
        read_count++;

        last = NULL;
        alarm = alarm_list;

        while (alarm->id != current_id)
       // Search the alarm list until the new request is found.
        {
         last = alarm;
         alarm = alarm->link;
        }

        if (alarm->cancel == 1) // The request is a cancel request.
        {
         read_count--;
         want_to_cancel = 1;
        // Indicate that alarm_thread wants to write to the alarm list.

        while ((read_count > 0) || (write_count > 0));
      // If there are reader or writer processes executing, do nothing.

        write_count++;
        want_to_cancel = 0;
    // Set indicator back to 0 after incrementing the writer count so that
   // reader threads don't activate the moment the indicator is set to 0.

        status = pthread_mutex_lock (&alarm_mutex);
    // Locking the mutex is largely unnecessary in this setup as
    // no other threads can access the list during cancelling,
    // but just to be safe the mutex is locked anyway.

            if (status != 0)
                err_abort (status, "Lock mutex");

        if (last == NULL) // The very first request is to be cancelled.
         alarm_list = alarm->link;
        else // A different request is to be cancelled.
         last->link = alarm->link;

        printf("Alarm processed at %d: Cancel: Message(%d)\n", time (NULL),
         alarm->id);

        free(alarm);
      // Free up the alarm's memory space after removing it from the list.

        status = pthread_mutex_unlock (&alarm_mutex);
      // Unlock the mutex so that other threads can access the list.

            if (status != 0)
                err_abort (status, "Unlock mutex");

            write_count--;
      // Decrement the writer count after unlocking the mutex so that no
      // other threads try accessing the list while the mutex is still locked.
       }
       else if (alarm->new == 0)
      // The request replaces a previous request,
      // in which case there is nod need for a new periodic display thread.
       {
        printf("Alarm processed at %d: %d Message(%d) %s\n",
               time (NULL), alarm->seconds, alarm->id, alarm->message);

        read_count--;
       }
```

```
         else
        // The request does not replace any existing request,
        // and as such a new periodic display thread is
        // needed to handle the request.
         {
          alarmReceived = (alarm_t*)malloc (sizeof (alarm_t));
       // Allocate memory space to a copy of the alarm
       //(so that the periodic display thread can test
       // to see if the alarm has been changed).

    if (alarmReceived == NULL)
            errno_abort ("Allocate alarm");

       alarmReceived->link = NULL; // The copy should not link to anything.
       alarmReceived->seconds = alarm->seconds; // Copy the alarm's parameters.
       alarmReceived->creation = alarm->creation;
       alarmReceived->id = alarm->id;

       for (i = 0; i <= strlen(alarm->message); i++)
       alarmReceived->message[i] = alarm->message[i];

       alarmReceived->cancel = alarm->cancel;

       status = pthread_create (&thread, NULL, periodic_display_thread,
       (void *) alarmReceived);

          if (status != 0)
           err_abort (status, "Create alarm thread");

          printf("Alarm processed at %d: %d Message(%d) %s\n",
                time (NULL), alarm->seconds, alarm->id, alarm->message);

          read_count--;
         }

    }

}

int main (int argc, char *argv[])
{
    int status;
    char line[128];
    alarm_t *alarm;
    alarm_t **last, *next;
    pthread_t thread;

    status = pthread_create (&thread, NULL, alarm_thread, NULL);
```

```
  if (status != 0)
      err_abort (status, "Create alarm thread");

  while (1)
  {
   printf ("Alarm> ");

       if (fgets (line, sizeof (line), stdin) == NULL)
        exit (0);

       if (strlen (line) <= 1)
        continue;

       alarm = (alarm_t*)malloc (sizeof (alarm_t));

       if (alarm == NULL)
           errno_abort ("Allocate alarm");

       /*
        * Parse input line one of two ways:
        *
        * a) Time Message(Message_Number) Message, where Time is the time to
        * sleep between displays, Message_Number is the ID number,
        * and Message is the alarm message.
        * b) Cancel: Message(Message_Number), where Message_Number
        * is the ID number.
        */
       if ((line[0] == 'C') && ((sscanf (line, "Cancel: Message(%d)",
         &alarm->id) < 1) || (alarm->id <= 0)))
      // The request is a cancel request; IDs should be one or higher.
       {
        fprintf (stderr, "Bad command\n");
        free (alarm);
       }
       else if ((line[0] != 'C') && ((sscanf (line, "%d Message(%d) %64[^\n]",
          &alarm->seconds, &alarm->id, alarm->message) < 3)
          || (alarm->id <= 0) || (alarm->seconds <= 0)))
      // For non-cancel requests, IDs should still be one or higher,
      // as should the sleep time.
       {
           fprintf (stderr, "Bad command\n");
           free (alarm);
       }
       else
{
 want_to_write = 1; // Indicate that main wants to write to the alarm list.
```

```
        while ((read_count > 0) || (write_count > 0) || (want_to_cancel == 1));
        // If there are reader or writer processes executing,
        // or if alarm thread wants to delete a request, do nothing.

               write_count++;
               want_to_write = 0;
       // Set indicator back to 0 after incrementing the writer count so
       // that reader threads don't activate the moment the indicator is set to 0.

               status = pthread_mutex_lock (&alarm_mutex);
       // Locking the mutex is largely unnecessary in this setup as no other
       // threads can access the list during adding,
       // but just to be safe the mutex is locked anyway.

               if (status != 0)
                   err_abort (status, "Lock mutex");

               if (line[0] == 'C')
               // If the request is a cancel request, initialize with dummy values.
               {
                alarm->seconds = 0;
                alarm->message[0] = '\0';
                alarm->cancel = 1;
                alarm->creation = time (NULL);

                printf("Alarm Request Received at %d: Cancel: Message(%d)\n",
                  time (NULL), alarm->id);
               }
               else // Otherwise, values are already initialized;
                // Simply set the creation time and
                // indicator that this isn't a cancel request.
               {
                alarm->creation = time (NULL);
                alarm->cancel = 0;

                printf("Alarm Request Received at %d: %d Message(%d) %s\n",
                  time (NULL), alarm->seconds, alarm->id, alarm->message);
            }

#ifdef DEBUG
     printf ("[listPre: ");

     for (next = alarm_list; next != NULL; next = next->link)
         printf ("ID:%d(%d)[\"%s\"] ", next->id, next->seconds, next->message);

     printf ("]\n");
#endif
            /*
            * Insert the new alarm into the list of alarms,
            * sorted by ID.
            */
           last = &alarm_list;
        next = *last;

        while (next != NULL)
        {
            if (next->id > alarm->id)
            {
                alarm->link = next;
                *last = alarm;
                alarm->new = 1; // Alarm does not replace an existing alarm.

                break;
            }
            else if (next->id == alarm->id)
            {
             alarm->link = next->link;
             *last = alarm;
             alarm->new = 0; // Alarm replaces an existing alarm.

             free(next); // Delete the old alarm from memory.

             break;
            }

            last = &next->link;
            next = next->link;
        }

        /*
         * If we reached the end of the list, insert the new alarm
         * there. ("next" is NULL, and "last" points to the link
         * field of the last item, or to the list header.)
         */
        if (next == NULL)
        {
            *last = alarm;
            alarm->link = NULL;
            alarm->new = 1; // Alarm does not replace an existing alarm.
        }

#ifdef DEBUG
    printf ("[listPost: ");

    for (next = alarm_list; next != NULL; next = next->link)
        printf ("ID:%d(%d)[\"%s\"] ", next->id, next->seconds, next->message);
    printf ("]\n");
#endif

      current_id = alarm->id; // Set the ID alarm thread should look for.

      status = pthread_mutex_unlock (&alarm_mutex);
      // Unlock the mutex so that other threads can access the list.

             if (status != 0)
                 err_abort (status, "Unlock mutex");

             write_count--;
             // Decrement the writer count after unlocking the mutex so
             // that no other threads try accessing the list while
             // the mutex is still locked.
             signal = 1;
             // Activate alarm thread after the mutex has been unlocked.
        }

    }
```

5.2 *Test_output*

```
inputs:

10 Message(3) yes
5 Message(1) no
8 Message(2) oui
5 Message(2) non
Cancel Message(4)
3 3 3
10 yes
20 Messge(4) wont work
Cncel Message(1)
10 message(4) lower case
cancel message(3)
Message(3) nor will this
15 Message(0) this wont either
0 Message(1) bad command
16 Message(-3) worse command
-10 Message(2) even worse command
Cancel Message(0) nothing to cancel here
Cancel Message(-5) or here
Cancel Message(3)
5 Message(3) final one
Cancel Message(3)
Cancel Message(2)
Cancel Message(1)
```

output:

```
Alarm> 10 Message(3) yes
Alarm Request Received at 1449505692: 10 Message(3) yes
Alarm> Alarm processed at 1449505692: 10 Message(3) yes
Alarm displayed at 1449505692: 10 Message(3) yes
Alarm displayed at 1449505702: 10 Message(3) yes
5 Message(1) no
Alarm Request Received at 1449505703: 5 Message(1) no
Alarm> Alarm processed at 1449505703: 5 Message(1) no
Alarm displayed at 1449505703: 5 Message(1) no
Alarm displayed at 1449505708: 5 Message(1) no
Alarm displayed at 1449505712: 10 Message(3) yes
Alarm displayed at 1449505713: 5 Message(1) no
8 Message(2) oui
Alarm Request Received at 1449505714: 8 Message(2) oui
Alarm> Alarm processed at 1449505714: 8 Message(2) oui
Alarm displayed at 1449505714: 8 Message(2) oui
Alarm displayed at 1449505718: 5 Message(1) no
Alarm displayed at 1449505722: 10 Message(3) yes
Alarm displayed at 1449505722: 8 Message(2) oui
Alarm displayed at 1449505723: 5 Message(1) no
5 Message(2) non
Alarm Request Received at 1449505724: 5 Message(2) non
Alarm> Alarm processed at 1449505724: 5 Message(2) non
Alarm displayed at 1449505728: 5 Message(1) no
Alarm changed at 1449505730: 5 Message(2) non
Alarm displayed at 1449505730: 5 Message(2) non
Alarm displayed at 1449505732: 10 Message(3) yes
Cancel: Message(4)
Alarm Request Received at 1449505733: Cancel: Message(4)
Alarm> Alarm processed at 1449505733: Cancel: Message(4)
Alarm displayed at 1449505733: 5 Message(1) no
Alarm displayed at 1449505735: 5 Message(2) non
Alarm displayed at 1449505738: 5 Message(1) no
Alarm displayed at 1449505740: 5 Message(2) non
Alarm displayed at 1449505742: 10 Message(3) yes
3 3 3
Bad command
Alarm> Alarm displayed at 1449505743: 5 Message(1) no
Alarm displayed at 1449505745: 5 Message(2) non
Alarm displayed at 1449505748: 5 Message(1) no
10 yes
Bad command
Alarm> Alarm displayed at 1449505750: 5 Message(2) non
Alarm displayed at 1449505752: 10 Message(3) yes
Alarm displayed at 1449505753: 5 Message(1) no
20 Messge(4) wont work
Bad command
Alarm> Alarm displayed at 1449505755: 5 Message(2) non
Alarm displayed at 1449505758: 5 Message(1) no
Cncel: Message(1)
Bad command
Alarm> Alarm displayed at 1449505760: 5 Message(2) non
Alarm displayed at 1449505762: 10 Message(3) yes
Alarm displayed at 1449505763: 5 Message(1) no
10 message(4) lower case
Bad command
Alarm> Alarm displayed at 1449505765: 5 Message(2) non
Alarm displayed at 1449505768: 5 Message(1) no
cancel: message(3)
Bad command
Alarm> Alarm displayed at 1449505770: 5 Message(2) non
Alarm displayed at 1449505772: 10 Message(3) yes
Alarm displayed at 1449505773: 5 Message(1) no
Message(3) nor will this
Bad command
Alarm> Alarm displayed at 1449505775: 5 Message(2) non
Alarm displayed at 1449505778: 5 Message(1) no
15 Message(0) this wont either
Bad command
Alarm> Alarm displayed at 1449505780: 5 Message(2) non
Alarm displayed at 1449505782: 10 Message(3) yes
Alarm displayed at 1449505783: 5 Message(1) no
0 Message(1) bad command
Bad command
Alarm> Alarm displayed at 1449505785: 5 Message(2) non
Alarm displayed at 1449505788: 5 Message(1) no
16 Message(-3) worse command
Bad command
Alarm> Alarm displayed at 1449505790: 5 Message(2) non
Alarm displayed at 1449505792: 10 Message(3) yes
Alarm displayed at 1449505793: 5 Message(1) no
Alarm displayed at 1449505795: 5 Message(2) non
-10 Message(2) even worse command
Bad command
Alarm> Alarm displayed at 1449505798: 5 Message(1) no
Alarm displayed at 1449505800: 5 Message(2) non
Alarm displayed at 1449505802: 10 Message(3) yes
Cancel: Message(0) nothing to cancel here
Bad command
Alarm> Alarm displayed at 1449505803: 5 Message(1) no
Alarm displayed at 1449505805: 5 Message(2) non
Alarm displayed at 1449505808: 5 Message(1) no
```

```
Cancel: Message(-5) or here
Bad command
Alarm> Alarm displayed at 1449505810: 5 Message(2) non
Alarm displayed at 1449505812: 10 Message(3) yes
Alarm displayed at 1449505813: 5 Message(1) no
Alarm displayed at 1449505815: 5 Message(2) non
Cancel Message(3)
Bad command
Alarm> Alarm displayed at 1449505818: 5 Message(1) no
Alarm displayed at 1449505820: 5 Message(2) non
Cancel: Message(3)
Alarm Request Received at 1449505821: Cancel: Message(3)
Alarm> Alarm processed at 1449505821: Cancel: Message(3)
Display thread exiting at 1449505822: 10 Message(3) yes
Alarm displayed at 1449505823: 5 Message(1) no
Alarm displayed at 1449505825: 5 Message(2) non
Alarm displayed at 1449505828: 5 Message(1) no
Alarm displayed at 1449505830: 5 Message(2) non
5 Message(3) final one
Alarm Request Received at 1449505831: 5 Message(3) final one
Alarm> Alarm processed at 1449505831: 5 Message(3) final one
Alarm displayed at 1449505831: 5 Message(3) final one
Alarm displayed at 1449505833: 5 Message(1) no
Alarm displayed at 1449505835: 5 Message(2) non
Alarm displayed at 1449505836: 5 Message(3) final one
Alarm displayed at 1449505838: 5 Message(1) no
Alarm displayed at 1449505840: 5 Message(2) non
Alarm displayed at 1449505841: 5 Message(3) final one
Cancel: Message(3)
Alarm Request Received at 1449505842: Cancel: Message(3)
Alarm> Alarm processed at 1449505842: Cancel: Message(3)
Alarm displayed at 1449505843: 5 Message(1) no
Alarm displayed at 1449505845: 5 Message(2) non
Display thread exiting at 1449505846: 5 Message(3) final one
Alarm displayed at 1449505848: 5 Message(1) no
Alarm displayed at 1449505850: 5 Message(2) non
Cancel: Message(2)
Alarm Request Received at 1449505851: Cancel: Message(2)
Alarm> Alarm processed at 1449505851: Cancel: Message(2)
Alarm displayed at 1449505853: 5 Message(1) no
Display thread exiting at 1449505855: 5 Message(2) non
Alarm displayed at 1449505858: 5 Message(1) no
Alarm displayed at 1449505863: 5 Message(1) no
Cancel: Message(1)
Alarm Request Received at 1449505865: Cancel: Message(1)
Alarm> Alarm processed at 1449505865: Cancel: Message(1)
Display thread exiting at 1449505868: 5 Message(1) no
```

5.3 *README*

```
1. First copy the files "New_Alarm_Cond.c", and "errors.h" into your
   own directory.

2. To compile the program "New_Alarm_Cond.c", use the following command:
   cc New_Alarm_Cond.c -D_POSIX_PTHREAD_SEMANTICS -lpthread

3. Type "a.out" to run the executable code.

4. At the prompt "ALARM>", type in the number of seconds at which
   the alarm should expire and the ID/tag, followed by the text of the message.
   For example:   ALARM> 2 Message(1) Good Morning!
   (To exit from the program, type Ctrl-d.)
```

5.4 *makefile*

```
a: New_Alarm_Cond.c errors.h
        cc New_Alarm_Cond.c -D_POSIX_PTHREAD_SEMANTICS -lpthread
        a.out
```

5.5 *error.h*

```
#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
```