EECS 3221: POSIX Threads Assignment Report

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

November 4, 2015

Contents

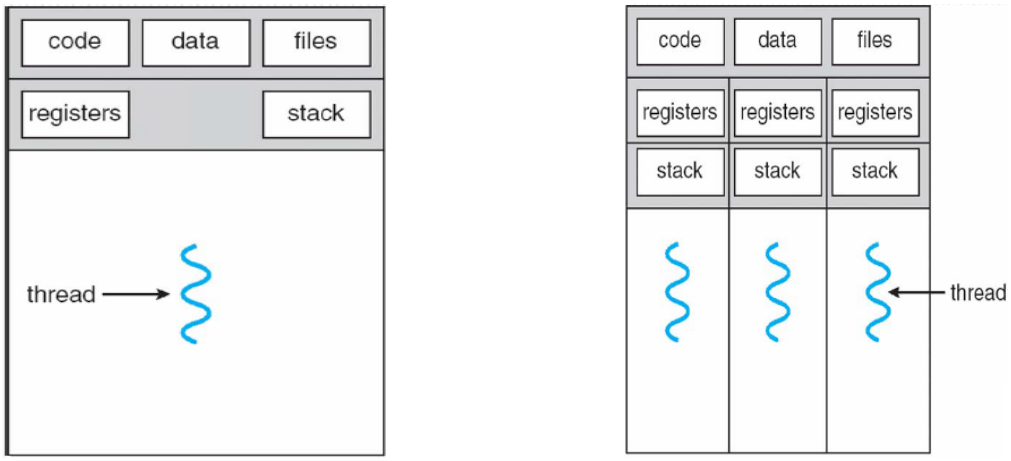
1 Abstract

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers [1]. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard [1]. Implementations that adhere to this standard are referred to as POSIX threads, or simply, Pthreads.

In this case study, the functionality of threads was explored through the programming of an alarm. The threads were implemented to handle incoming alarm requests and were tasked with outputting the machine time and an alarm message at specific intervals. This case study on Pthreads furthered the understanding of how multi-threads can be designed and implemented by the user to achieve a specified task.

1.1 POSIX Thread Structure

The following diagrams, as shown in Figure 1, detail the generic structure of single- and multi-threaded processes.



(a) Single-threaded Process

(b) Multi-threaded Process

Figure 1

As illustrated in Figure 1(a), a typical thread structure contains a code section, a data section and operating system resources (files). A thread also has its own set of dedicated registers that are used to handle specific tasks. This basic unit of the Central Processing Unit (CPU) utilizes stack space to store these processes. For this particular study, multi-threaded processes were used to handle multiple incoming alarm requests, as shown in Figure 1(b). The handling of multiple requests simultaneously emphasizes the parallelistic ability of thread architecture. A new thread is created each time an existing thread is already running. Such functionality is not possible with single-threaded processes because each subsequent alarm request must wait until the existing request concludes. Along with reduced waiting time, a structural advantage of utilizing multi-threaded processes is less overhead and the efficient sharing of information with adjacent threads.

2 Discussion

2.1 Design and Implementation

There were countless specific design decisions made during the course of the assignment's implementation process. One such decision was to not have the code in the output thread run in an infinite while loop. This decision was due to the fact that a new output thread needed to be created every time an alarm request was processed. It would of been rather pointless to have each thread run infinitely when the thread is only required to process the code once. As such, the output thread simply executes the request by way of a for loop that sleeps for one second after every printout,

followed by that "iteration" of the thread disintegrating.

Within the output thread, the decision was made to set the thread up in such a way that handling the situation where the alarm's expiry had already passed was a particular case not implemented in the for loop. This was due to the fact that the for loop ran once for every second, from the current time to the expiry time. As such, the for loop could not run whenever the number of seconds between the current time and expiry time was less than or equal to 0. However, the sleep time could not have been set to one when the expiry time had already passed as the program would have slept for one second before the alarm expired, thus resulting in an alarm that expires a second too late.

2.2 Rectifying Challenges

Various challenges were encountered while trying to program *My_Alarm.c*. The first problem that was stumbled upon was a simple mistake where the code to output "alarm retrieved at etc." was misplaced such that it was not inside an if-else statement that checked to see whether or not the alarm was null. This oversight resulted in a segmentation fault. The first step to fixing this problem required finding the specific location of the error. This was accomplished by organizing the program to print out the precise code as it was executed on each line. Once that particular line was discovered, fixing the issue was straightforward as the decision was made to place the printout statements in an if structure that checked whether or not the alarm was null.

Once the implementation of the code was complete, a more significant problem was encountered during the testing process. Although the code appeared to be working correctly, it was noticed that if the user provided a new alarm request while one was being processed, the output of the request being processed was changed to match the new request. Listing 1 outlines an example of the problem encountered.

Listing 1:

```
alarm> 9 hi
Alarm Received at 1446501215: 9 hi
Alarm Retrieved at 1446501215: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
Alarm: >: 9 hi
alarm> 3 yo
Alarm: >: 3 yo
Alarm: >: 3 yo
Alarm Expired at 1446501224: 3 yo
Alarm Received at 1446501224: 3 yo
Alarm Retrieved at 1446501224: 3 yo
Alarm: >: 3 yo
Alarm: >: 3 yo
Alarm: >: 3 yo
```

```
Alarm Expired at 1446501227: 3 yo
```

This error was a result of passing the output thread a pointer to the alarm which caused the same memory address to be interacted with by both the main and output thread. Initial experimentation with adding a mutex to the output thread caused further issues with the program execution halting altogether. The output thread normally just executed once, rather than having an infinite loop like the main and alarm threads. As such, setting up the locking and unlocking of the mutex such that it would keep trying to execute the code, rather than giving up the moment a lock failed, required an additional while loop. The while loop was set to keep running whenever the status did not equal 0 (0 if the lock/unlock was successful), but this only seemed to cause the program to hang altogether. A better idea was later brainstormed. Instead of each thread having a mutex, the alarm thread (i.e. not the output thread) would be set up to allocate memory for a second alarm request acting as a copy of the first, storing the data of the alarm request to-be-processed in the second one, then passing a pointer from the second one to the output thread. The alarm thread would then free up the memory space of the original alarm request, while the now mutex-free output thread would free up the memory space of the copy.

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. The inconsistency of the number of arguments in alarm_request prompted a bad command message. 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

Although threads are a powerful unit of the CPU, they must be utilized in an appropriate manner. While the output of the code may be correct, it is important to consider the fact that threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they were created in. Threads may also execute at different speeds. When threads are executing (racing to completion), they may give unexpected results (race condition). Mutexes and joins must be utilized to achieve a predictable execution order and outcome [2]. As evident in the code shown in Appendix 5.1, mutex locks were used extensively to overcome race conditions and to synchronize shared data among the threads. In closing, understanding when and how to use threads properly enables new and creative ways of approaching programming challenges.

4 References

[1] B. Barney. Posix threads programming. Lawrence Livermore National Laboratory.

[2] G. Blelloch. Posix thread (pthread) libraries. Carnegie Mellon University's School of Computer Science.

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

5 Appendix

5.1 *My_Alarm.c*

```
/*
 * My_Alarm.c
 *
 * This is an enhancement to the alarm_mutex.c file. Instead
 * of having a single alarm thread which retrieves and executes
 * alarm requests, now the thread retrieving the alarm requests
 * creates a new thread which then executes the alarm request.
 */

#include <pthread.h>
#include <time.h>
#include "errors.h"

// #define DEBUG // Debugger for checking the contents of the list and
// the status of the alarm request being executed.
// #define DEBUG2 // Debugger for checking exactly when each thread
// executes.

/*
 * The "alarm" structure is unchanged from before. As before,
 * the "time_t" is stored for each alarm, so that they can be
 * sorted. Storing the requested number of seconds would not be
 * enough, since the "alarm output 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[64];
} alarm_t;

pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
alarm_t *alarm_list = NULL;

/*
 * New alarm output thread designed to take a given alarm request,
 * and process it in much the same as the "retrieval" thread did
```

```
 * originally. This thread executes only once per alarm request.
 */

void *alarmOutput_thread (void *alarmArg)
{
        alarm_t *alarm;
    int exec_time, i;
    time_t now;

#ifdef DEBUG2
        printf("Output thread Executing\n");
#endif

    alarm = (alarm_t *) alarmArg;
    now = time (NULL);

    if (alarm->time <= now)
        exec_time = 0;
        // If the time at which the alarm was suppose to expire has
        // already passed, do not execute any print statements.
    else
        exec_time = alarm->time - now;
        // Otherwise, the amount of time to execute should be the
        // difference of the expiry time and the current since EPOCH.

#ifdef DEBUG
    printf ("[waiting: %d(%d)\"%s\"]\n",
        alarm->time, exec_time, alarm->message);
#endif

        for (i = exec_time; i > 0; i--)
        // If there is still time left before expiry, print the
        // alarm request once every second until expiry time.
        {
        printf("Alarm: >: %d %s\n", alarm->seconds, alarm->message);
        sleep(1);
        }

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

    free(alarm);
}

/*
 * The alarm thread's start routine.
 */
```

```
void *alarm_thread (void *arg)
{
    alarm_t *alarm;
    alarm_t *alarmReceived;
        // Copy of alarm to prevent the generation of a new alarm
        // request in main interfering with the one being
        // processed by alarmOutput_thread.
    int sleep_time;
    time_t now;
    int status, i;
    pthread_t thread;

    /*
     * Loop forever, processing commands. The alarm thread will
     * be disintegrated when the process exits.
     */

    while (1)
    {

#ifdef DEBUG2
                printf("Retrieval thread Executing\n");
#endif

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

        alarm = alarm_list;

        /*
         * If the alarm list is empty, wait for one second. This
         * allows the main thread to run, and read another
         * command. If the list is not empty, remove the first
         * item.
         */

        if (alarm == NULL)
            sleep(1);
        else
        {
            alarm_list = alarm->link; // Remove alarm from the list.
                alarmReceived = (alarm_t*)malloc (sizeof (alarm_t));
                // Allocate memory space to the copy of alarm.

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

```
                alarmReceived->link = NULL;
                alarmReceived->seconds = alarm->seconds;
                alarmReceived->time = alarm->time;

        for (i = 0; i <= strlen(alarm->message); i++)
                // Create a copy of alarm's message top store in
                // alarmReceived.
                alarmReceived->message[i] = alarm->message[i];
                status = pthread_create (&thread,
                NULL, alarmOutput_thread, (void *) alarmReceived);
                // Pass alarm's copy to the output thread instead
                // of alarm itself.

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

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

                status = pthread_mutex_unlock (&alarm_mutex);

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

        if (alarm != NULL)
        // Following code occurs separate from above if structure
        // because it should happen after unlocking the mutex,
        // but the mutex needs to be unlocked regardless of
        // whether or not alarm = NULL.

        {

        free (alarm); // Free the original alarm's memory space.
        sleep(1);

        }
        }
}

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

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

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

    /*
     * Loop forever, accepting alarm requests.
     */

    while (1)
    {

#ifdef DEBUG2
                printf("Main thread Executing\n");
#endif

        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 into seconds (%d) and a message
         * (%64[^\n]), consisting of up to 64 characters
         * separated from the seconds by whitespace.
         */

        if (sscanf (line, "%d %64[^\n]", &alarm->seconds,
                alarm->message) < 2)
        {
            fprintf (stderr, "Bad command\n");
            free (alarm);
        }
        else
        {
            status = pthread_mutex_lock (&alarm_mutex);

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

            alarm->time = time (NULL) + alarm->seconds;
```

```
            printf("Alarm Received at %d: %d %s\n",
                    alarm->time - alarm->seconds, alarm->seconds,
                    alarm->message);

            /*
             * Insert the new alarm into the list of alarms,
             * sorted by expiration time.
             */

            last = &alarm_list;
            next = *last;

            while (next != NULL)
            {
                if (next->time >= alarm->time)
                {
                    alarm->link = next;
                    *last = alarm;
                    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;
            }

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

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

            printf ("]\n");
#endif
```

```
            status = pthread_mutex_unlock (&alarm_mutex);

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

        }

    }

}
```

5.2 *Test_output*

```
inputs:

9 hi
7 yo
22 test
0 hey badabing
INCORRECT
-36 negation
10 works
yabba dabba doo
-0 ???
14 yah
-20 nah
1 good command


output:

alarm> Alarm Received at 1446568059: 9 hi
alarm> Alarm Retrieved at 1446568059: 9 hi
Alarm Received at 1446568059: 7 yo
alarm> Alarm Retrieved at 1446568059: 7 yo
Alarm Received at 1446568059: 22 test
alarm> Alarm Received at 1446568059: 0 hey badabing
alarm> Alarm: >: 9 hi
alarm> Alarm: >: 7 yo
Alarm Retrieved at 1446568059: 0 hey badabing
Alarm Expired at 1446568059: 0 hey badabing
Alarm Retrieved at 1446568059: 22 test
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
```

```
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm Expired at 1446568066: 7 yo
Alarm: >: 22 test
Alarm: >: 9 hi
Alarm: >: 22 test
Alarm Expired at 1446568068: 9 hi
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm: >: 22 test
Alarm Expired at 1446568081: 22 test
[From stderr: Bad command)]
Alarm Received at 1446570913: -36 negation
alarm> Alarm Received at 1446570913: 10 works
[From stderr: Bad command)]
alarm> alarm> Alarm Received at 1446570913: 0 ???
alarm> Alarm Received at 1446570913: 14 yah
alarm> Alarm Received at 1446570913: -20 nah
alarm> Alarm Received at 1446570913: 1 good command
alarm> Alarm Retrieved at 1446570913: -36 negation
Alarm Expired at 1446570913: -36 negation
Alarm Retrieved at 1446570913: -20 nah
Alarm Expired at 1446570913: -20 nah
Alarm Retrieved at 1446570913: 0 ???
Alarm Expired at 1446570913: 0 ???
Alarm Retrieved at 1446570913: 1 good command
```

```
Alarm: >: 1 good command
Alarm Retrieved at 1446570913: 10 works
Alarm: >: 10 works
Alarm Retrieved at 1446570913: 14 yah
Alarm: >: 14 yah
```

5.3 *README*

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

2. To compile the program "My_Alarm.c", use the following command:
   cc My_Alarm.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, followed by the text of the message.
   For example:   ALARM> 2 Good Morning!
   (To exit from the program, type Ctrl-d.)

5. Note that it may take a while between entering a command and
   the program actually receiving that command and processing it.
   If it takes too long, uncommenting "#define DEBUG2" in the
   code seems to drastically reduce the wait time for unknown reasons,
   though at the cost of adding unnecessary print statements.
   Either way, however, the code performs as intended.
```

5.4 *makefile*

```
a: My_Alarm.c errors.h
        cc My_Alarm.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
```