Understanding Threads in a UNIX System
Lab. #4

Arif A. Balik

Undergraduate Student
Sytstems Programming
Department of Computer Science
Arel University
Büyükçekmece, İstanbul 34537
Email: arifbalik@outlook.com

March 29, 2019

Abstract

This report include analysis of threads in UNIX environment based on programs which provided in System Programming course. First section explains theory of operation, second section answers some questions asked in the experiment paper, and third section suggests and implements some improvements and optimisations on the code.

This report is powered by LaTeX

Theory of Operation

Single Thread Application

For the first part, there is a single threaded application which does nothing but running a loop, calling two functions and calculating the amount of time it took to execute the program. Program executes in a very standart sequence as expected.

Following figure shows the timeline of the program. First column shows the time which functions occurred and the other columns show which one occurred. The number in square brackets is the value of the variable **r1**

Timeline (singlethread.c)

0 sec	main [0]		
3 sec	main [0]		
6 sec	main [0]		
9 sec		func1 [0]	
13 sec		func1 [0]	
17 sec		func1 [0]	
21 sec		func1 [0]	
25 sec		func1 [0]	
29 sec		func1 [0]	
33 sec		func1 [0]	
37 sec		func1 [0]	
41 sec		func1 [0]	
45 sec			func2 [0]
48 sec			func2 [0]
51 sec			func2 [0]
54 sec			func2 [0]
57 sec			func2 [0]
60 sec			func2 [0]
63 sec			func2 [0]

As seen in the output of the program, **main** thread runs 3 times with 3 second intervals, **func1** runs 9 times with 4 second intervals and **func2** runs 7 times with 3 second intervals.

Multi Thread Application

In **multithread.c**, program creates two threads as **func1** and **func2** respectively. Then it runs just as same as single threaded application. But observe the output of the program;

First as they execute simultaneously, their occurrence is somewhat random, it can be observed by running the program several times. And because they work at the same time, the total time of execution is half the size of single thread application. Also the reason of the randomness in the occurrence is because the delay time of each functions is different.

There is no **race condition** (race condition will be discussed in the **Q&A** section in further detail) because **mutex** is implemented, but then again this program can hardly create a race condition without mutex, since the occurrence of threads is relatively separate.

Timeline (multithread.c)

0 sec	main [101]		
0 sec			func2 [101]
0 sec		func1 [101]	
3 sec			func2 [101]
3 sec	main [102]		
4 sec		func1 [102]	
6 sec	main [103]		
6 sec			func2 [103]
8 sec		func1 [103]	
9 sec			func2 [103]
12 sec		func1 [103]	
12 sec			func2 [103]
15 sec			func2 [103]
16 sec		func1 [103]	
18 sec			func2 [103]
20 sec		func1 [103]	
24 sec		func1 [103]	
28 sec		func1 [103]	
32 sec		func1 [103]	

The program has been tested without waiting threads to terminate. The expected result should be that after **main** function terminates, all the threads will be lost and total execution time is 9 seconds since the loop in the main function runs 3 times wirh 3 second delay. The following output shows the results.

Timeline (multithread.c) without wait

0 sec	main [101]		
0 sec			func2 [101]
0 sec		func1 [101]	
3 sec			func2 [102]
3 sec	main [102]		
4 sec		func1 [102]	
6 sec			func2 [102]
6 sec	main [103]		
8 sec		func1 [103]	
9 sec			func2 [103]

Indeed, results proves the assumption.

Multi Process Application

In **multiprocess.c** the only difference is creation of threads is done by processes. But outcomes are not even close.

Following output shows the results of running the program.

Timeline (multiprocess.c)

0 sec	main [101]		
0 sec		func1 [0]	
0 sec			func2 [0]
3 sec			func2 [0]
3 sec	main [102]		
4 sec		func1 [0]	
6 sec			func2 [0]
6 sec	main [103]		
8 sec		func1 [0]	
9 sec			func2 [0]
12 sec		func1 [0]	
12 sec			func2 [0]
15 sec			func2 [0]
16 sec		func1 [0]	
18 sec			func2 [0]
20 sec		func1 [0]	
24 sec		func1 [0]	
28 sec		func1 [0]	
32 sec		func1 [0]	

First noticeable thing is that the values for **r1** is not the same with **main**. That is because processes copied the variables to new spaces so their addresses are no longer the same with **main**.

And the other interesting behaviour is when **main** function terminates without waiting anything to terminate, the threads continues their execution. Again that is because processes created their own environment which is isolated enough for threads to continue. So we would get the same output as above.[i]

[i] that is, when childs wait for the threads to terminate to terminate themselves

Q&A

Q1. *What is a program thread?*

Threads are some kind of process, **dependently**[ii] and concurrently running within the parent process.

> To Linux, a thread is just a special kind of process. (Linux Kernel Development, 2011, p. 29)

Q2. *What is the purpose of using threads in programs?*

As seen in the section **Theory of Operation**, it reduces the amount of time for a program to do a certain job and increases performance.

Q3. *Is there any difference between processes and threads (explain)?*

Yes. Threads are sharing the same address space as parent process, and threads naturally can only work locally.

Q4. *When is the real parallelism of thread execution is possible?*

In fact, there is no such thing, but when kernel decides to run the thread in a different **physical** core, then thread has the possibility to achieve parallel computation. Otherwise kernel just schedules the codes in an order to simulate parallelism.

Q5. *What thread standard is implemented in UNIX Systems?*

POSIX. Hence the name **p** in the first character of functions related to threading.

Q6. *Are threads available in Windows operating systems?*

Yes. By definition, an operating system should be able to run multiple programs simultaneously, this includes threads as well.

Q7. *How many threads are there in a program when it just starts (that is, at the very beginning)?*

There is only one called *main* thread.

Q8. *Is it possible to change the number of threads in a program during its execution (explain)?*

[ii] by means of memory

Yes. For threads to have meaning program has to have the ability to create and terminate threads. In UNIX this can be achieved with the function **pthread_create**.

Kernel hold all the information of threads and processes in a structure called **task_struct** which is located in the kernel **include/linux/sched.h**, this allows kernel to dynamically add and remove threads.

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup
         * (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;
```

Q9. *How can a new thread be created in a program?*

Using **pthread_create** in the standart library **pthread.h**.

Q10. *Assume that the main (primary) thread in a program have created two new (secondary) threads and wants to continue its work only after these two new threads finish their work. Which function call should be used by the main thread for this purpose?*

pthread_join

Q11. *How will a thread, after having been created by the main function, learn what routine (function) it should use to perform its work? Is this routine a part of the main function or it should be specified outside the main function?*

It is given by the main program as a pointer to function during its creation via the first argument of the function **pthread_create**. It can locate anywhere as long as it is compiled with the main program.

Q12. *Is it possible for two or more threads to use the same routine (function)?*

Yes, because kernel calls the function from different sources, but it can be dangerous as they try to access same variable.

Q13. *Suppose that main thread, after the creation of a number of new threads, terminates. What will be with the created secondary running threads in this case?*

It will terminate since it has no environment to exist after main terminates.

Q14. *Is it possible to run a multithreaded program on a multiprocessor or a few computers connected to a network?*

Threads meant to work on multiprocessor, they are more efficient that way but they can't work on different machines.

Q15. *Suppose that two or more threads use (read and/or write) the same global data defined in the program. What can happen if you don't undertake same precautions? What should these precautions be?*

When multiple threads tries to access and change a variable, something called **race condition** occurs. That may create some unpredictable results. For example let one thread have an *if* condition that checks value of a variable and inside of the *if* condition it changes value of that variable, but at the same time, when the thread checks the variable and goes into the *if* condition, some other thread may change the value of that variable and cause critical damage.

To prevent such conditions UNIX offers a method called **thread locking** and it can be enabled by using a set of functions, such as **pthread_mutex_lock** which locks the access to resources in the region between **pthread_mutex_lock** and **pthread_mutex_unlock**.

For example in the **multithread.c** the variable **r1** can be safe from race conditions with the following code;

```
pthread_mutex_lock(&lock);
r1 = 100 + j;
pthread_mutex_unlock(&lock);
```

Q16. *Is programming with threads easier than programming without threads (explain)?*

It depends. For developer, it is easier to think and act safely in a single threaded program. But in real world a lot of applications requires multithreading, for example, it would be almost impossible to stream data from ethernet and drive an LCD display smoothly with a single threaded application because both of those peripherals requires complex and long protocols. So, how hard it is to work with threads becomes more and more irrelevant as they become unavoidable with the needs of modern technology.

Optimization

By giving up the microsecond resolution, time calculation can rid of heavy arithmetic operations.

Original

```
(float)((1000000*time2.tv_sec + time2.tv_usec) -
(1000000*time1.tv_sec + time1.tv_usec))/1000000
```

Optimised

```
time2.tv_sec - time1.tv_sec
```

There are other optimisations. Check Appendix for more.

Appendix

singlethread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock; //we need this for func1 and func2 to work

int main(int argc, char **argv){
        char* text = (char *)malloc(20);
        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        for (int j = 1; j < 4; ++j){
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif

                sleep(3);
        }


        func1();
        func2();
        gettimeofday(&time2, 0);
        printf("Main: Total elapsed time = %ld seconds\n", time2.tv_sec - time1.tv_sec);
        printf("Main thread terminated...\n");

        free(text);
        return 0;
}
```

multithread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock;

int main(int argc, char **argv){
        pthread_t td1, td2;
        int p;
        char* text = (char *)malloc(20);

        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        p = pthread_create(&td1, NULL, (void *)func1, NULL);

        if(p != 0){
                perror("Thread 1 creation problem");
                exit(1);
        }

        p = pthread_create(&td2, NULL, (void *)func2, NULL);

        if(p != 0){
                perror("Thread 2 creation problem");
                exit(1);
        }

        for (int j = 1; j < 4; ++j){
                pthread_mutex_lock(&lock);
                        r1 = 100 + j;
                pthread_mutex_unlock(&lock);
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif
                sleep(3);
        }

        //pthread_join(td1, NULL);
        //pthread_join(td2, NULL);

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        delta = (time2.tv_sec - time1.tv_sec);
        pthread_mutex_unlock(&lock);
        printf("Main: Total elapsed time = %f seconds\n", delta);
        printf("Main thread terminated...\n");

        return 0;
}
```

multiprocess.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock;

int main(int argc, char **argv){
        pthread_t td1, td2;
        int pid, c_pid;
        int p;
        float delta;
        char* text = (char *)malloc(20);

        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        pid = fork();

        if(pid < 0){
                perror("Error during child1 creation");
                exit(1);
        }
        if(pid == 0){
                p = pthread_create(&td1, NULL, (void *)func1, NULL);

                if(p != 0){
                        perror("Thread 1 creation problem");
                        exit(1);
                }
                pthread_join(td1, NULL);
                exit(1);
        }

        pid = fork();

        if(pid < 0){
                perror("Error during child2 creation");
                exit(1);
        }
        if(pid == 0){
                p = pthread_create(&td2, NULL, (void *)func2, NULL);

                if(p != 0){
                        perror("Thread 2 creation problem");
                        exit(1);
                }
                pthread_join(td2, NULL);
                exit(1);
        }

        for (int j = 1; j < 4; ++j){
                pthread_mutex_lock(&lock);
                        r1 = 100 + j;
                pthread_mutex_unlock(&lock);
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif
                sleep(3);
        }

        while ((c_pid = wait(0)) > 0);

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        delta = (time2.tv_sec - time1.tv_sec);
        pthread_mutex_unlock(&lock);
        printf("Main: Total elapsed time = %f seconds\n", delta);
        printf("Main thread terminated...\n");

        return 0;
}
```

func1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "timing.h"

void func1(void){
        int i;
        char* text = (char *)malloc(20);
        for (int i = 1; i < 10; ++i){
                #ifdef DRAW_TIMING_DIAGRAM
                        pthread_mutex_lock(&lock);
                        sprintf(text, "\t|\t\t|func1[%d]\t|\n",r1);
                        pthread_mutex_unlock(&lock);

                        time_line(text);
                #else // Original
                        printf("Function func1 prints and then sleeps 4 s: %d\n", i);
                #endif
                sleep(4);
        }
        free(text);
        return;
}
```

func2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "timing.h"

void func2(void){
        int i;
        char* text = (char *)malloc(20);;
        for (int i = 1; i < 8; ++i){
                #ifdef DRAW_TIMING_DIAGRAM
                        pthread_mutex_lock(&lock);
                        sprintf(text, "\t|\t\t|\t\t|func2[%d]\n",r1);
                        pthread_mutex_unlock(&lock);
                        time_line(text);
                #else // Original
                        printf("Function func2 prints and then sleeps 3 s: %d\n", i);
                #endif
                sleep(3);
        }
        free(text);
        return;
}
```

timing.h

```
#define DRAW_TIMING_DIAGRAM
extern struct timeval time1, time2;
extern int r1;
extern pthread_mutex_t lock;

static void time_line(const char *text){

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        printf("|%ldsec", (time2.tv_sec - time1.tv_sec));
        printf("%s",text);
        pthread_mutex_unlock(&lock);
}
```