CST 8703 Lab 2 - Fresh Threads

Kyle Chisholm

2022-05-17

Real-Time Systems and Embedded Programming

Inter-thread communication is introduced with an example where data is shared and altered amongst many threads. Concurrent programming strategies are employed to ensure data integrity and the expected output.

Source code available at https://github.com/chishok/CST8703-Lab2.

Submission Deadline: June 23, 2022

Background

Multi-threaded programs involve sharing data between threads, commonly referred to as Inter Thread Communication (ITC). Using multiple threads has the advantage of running processes in parallel. For real-time processes, tasks in each thread can be scheduled to run within guaranteed deadlines. However, care must be taken to ensure the software is "thread-safe" and data passing between threads is not corrupted. There are several methods of passing and protecting data with POSIX threads. This lab will introduce Threads, Mutexes, and Condition Variables to demonstrate safe inter-thread-communication.

POSIX Threads

A specific function signature is required to launch a new thread. The function which will launch a new thread is `pthread_create` with the following signature:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

For more details, read the man page form your terminal `man pthread_create`. The following table provides a brief overview of each argument:

Argument	Description
`thread`	Output thread handle (a number associated with the thread).
`attr`	Options for the thread, such as scheduling priority and policy, stack size (memory available to thread), CPU affinity, etc.

Argument	Description
`start_routine`	A function to be run as a thread, with one input to a `void` (generic) pointer and a return pointer.
`arg`	The argument that will be passed to the `start_routine` function when it is run.

If data is to be shared between threads, it's best practice to pass the data structure as an argument (to `arg`) rather than using global variables. The thread that called `pthread_create` should then wait for the thread(s) to finish by calling `pthread_join` on each thread id.

An example of spawning two threads:

> **NOTE**: Return values should always be checked. This example does not check errors.

1. Define a struct data type to pass to thread

   ```c
   typedef struct Data
   {
       int value;
   } Data;
   ```

2. Define a function to be run as a thread. Inside the function, cast the pointer to the data type. Some of this data may be shared with other threads also spawned by the main (manager) process.

   ```c
   void *my_thread_example(void *opt)
   {
       Data *thread_data = (Data *)opt;
       printf("Data value is %d\n", thread_data->value);
       return NULL;
   }
   ```

3. In the main function, create data object

   ```c
   Data my_data = {.value = 1};
   ```

4. Create thread attributes

   ```c
   pthread_attr_t pattr;
   pthread_attr_init(&pattr);
   // Set desired attributes, or leave with default values.
   ```

5. Spawn the threads. Shared data value is modified after first call to `pthread_create`.

   ```c
   pthread_t id_a = 0;
   pthread_t id_b = 0;
   pthread_create(&id_a, &pattr, my_thread_example, &my_data);
   my_data.value = 2;
   pthread_create(&id_b, &pattr, my_thread_example, &my_data);
   ```

6. Joint threads. This will block (wait) until the threads finish processing.

```
    pthread_join(id_a, NULL);
    pthread_join(id_b, NULL);
```

The full example code is:

```
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

typedef struct Data
{
    int value;
} Data;

void *my_thread_example(void *opt)
{
    Data *thread_data = (Data *)opt;
    printf("Data value is %d\n", thread_data->value);
    return NULL;
}

int main(void)
{
    Data my_data = {.value = 1};

    pthread_attr_t pattr;
    pthread_attr_init(&pattr);

    pthread_t id_a = 0;
    pthread_t id_b = 0;

    pthread_create(&id_a, &pattr, my_thread_example, &my_data);
    my_data.value = 2;
    pthread_create(&id_b, &pattr, my_thread_example, &my_data);

    pthread_join(id_a, NULL);
    pthread_join(id_b, NULL);

    return 0;
}
```

In this example, the shared data has a value initialized to `1` and appears to be modified after creating the first thread, but it will in fact be set to `2` before printing "Data value is 2" twice. In order to properly control how and when data is read or written by multiple threads, mutexes and condition variables described in the following sections are required.

Mutexes

Mutexes can be used to manage concurrent access to data from multiple threads. A mutex is first locked by a thread that wishes to have exclusive access to shared resources. The thread will then operate on the shared data and once operations are complete, the mutex is unlocked and other threads can in turn lock it. POSIX mutexes have the type `pthread_mutex_t`, are initialized with the function `pthread_mutex_init` and can be locked and unlocked with the functions `pthread_mutex_lock` and `pthread_mutex_unlock`. Refer to the man pages for each of these functions for more information. The following sequence diagram shows how threads use a mutex:

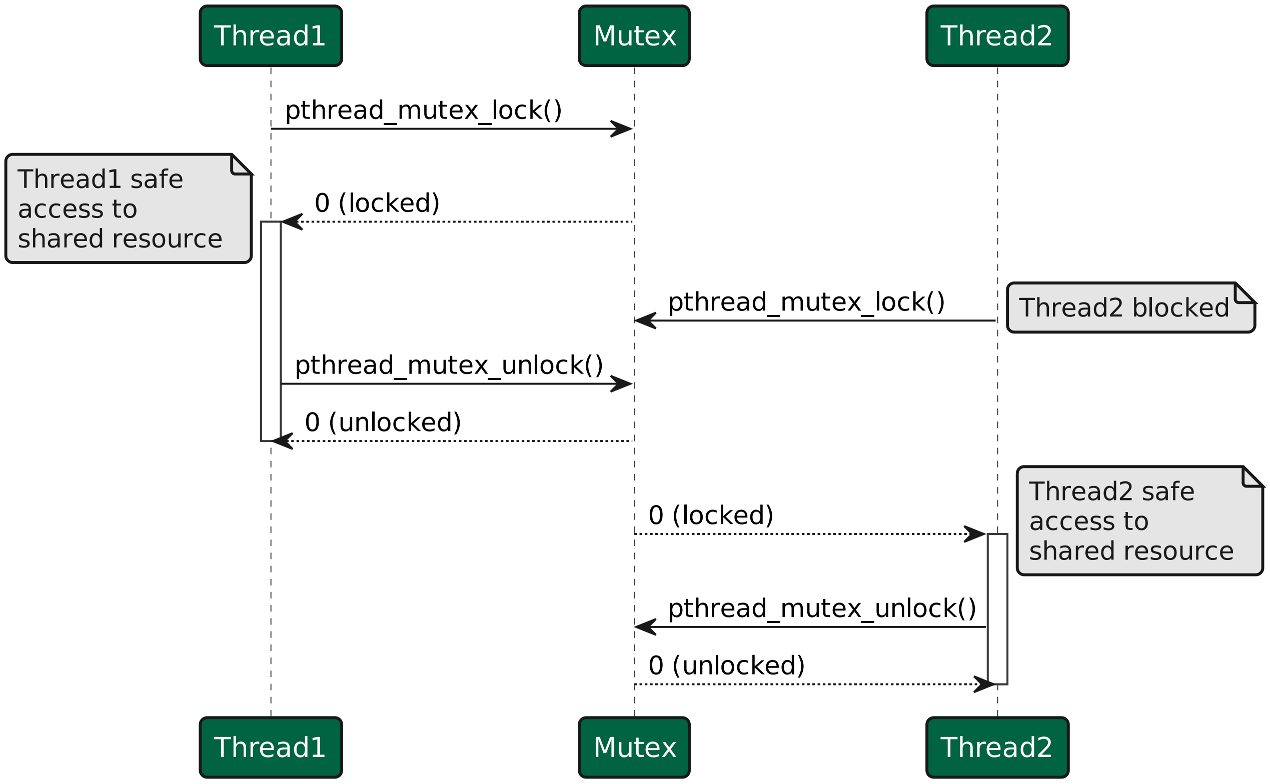

Figure 1: Mutex sequence with multiple threads

Mutexes can cause issues when multiple resources are being locked and accessed by multiple threads. A classic example is deadlock when it becomes impossible for multiple threads to proceed because they are waiting to lock mutexes which will never be unlocked. The following sequence diagram shows how a deadlock can occur:

A lock may be held for too long and a critical deadline can be exceeded in a real-time application such as a controller with a periodic task. In order to guard against excessive blocking durations and prevent tasks from exceeding their deadlines, a timeout can be used on blocking calls to lock a mutex. With POSIX, the function `pthread_mutex_timedlock` uses a timeout.

> **NOTE**: The mutex lock with a timeout (`pthread_mutex_timedlock`) is an **absolute** time with `CLOCK_REALTIME`. Be careful to take the current time, then add the duration of the timeout to be used with the function. The nanosecond component of `struct timespec` must also be greater or equal to zero and less than 1e9.

An example of spawning two threads with a mutex:

> **NOTE**: Return values should always be checked. This example does not check for errors.

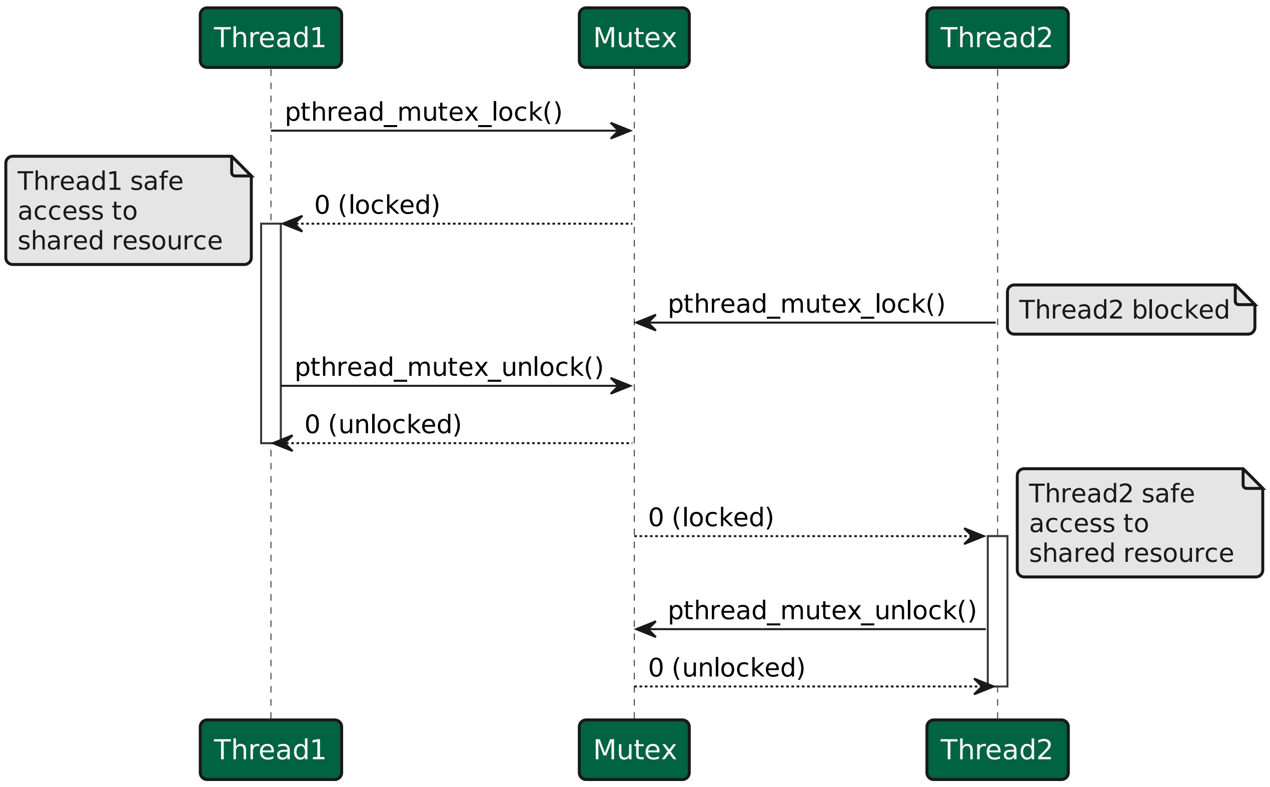

Figure 2: Mutex deadlock

```c
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

typedef struct Data
{
    pthread_mutex_t mutex;
    int value;
} Data;

void *my_thread_example(void *opt)
{
    Data *thread_data = (Data *)opt;

    pthread_mutex_lock(&thread_data->mutex);
    thread_data->value += 1;
    printf("Data value is %d\n", thread_data->value);
    fflush(NULL);
    pthread_mutex_unlock(&thread_data->mutex);

    return NULL;
}
```

```c
int main(void)
{
    Data my_data = {.value = 1};
    pthread_mutex_init(&my_data.mutex, NULL);

    pthread_attr_t pattr;
    pthread_attr_init(&pattr);

    pthread_t id_a = 0;
    pthread_t id_b = 0;

    pthread_create(&id_a, &pattr, my_thread_example, &my_data);
    pthread_create(&id_b, &pattr, my_thread_example, &my_data);

    pthread_join(id_a, NULL);
    pthread_join(id_b, NULL);

    return 0;
}
```

Output of the example is

```
Data value is 2
Data value is 3
```

In the thread function `my_thread_example`, the data is only modified and printed between mutex lock and unlock commands. This prevents the second thread from modifying the data at the same time.

Condition Variables

A condition variable is used to signal to other threads when shared data is modified and ready to check. A mutex is still necessary to prevent threads from accessing the same data but one or more threads can wait on a condition variable to be notified about changes in a shared state. The notification comes from another thread that changes the shared data state. The thread waiting on a signal will block with a call to `pthread_cond_wait` or `pthread_cond_timedwait`. The signalling thread would first modify the shared resource then signal or broadcast using `pthread_cond_signal` or `pthread_cond_broadcast` to threads waiting on the condition variable. The following sequence diagram shows how threads use a condition variable:

An example of spawning two threads with a mutex and condition variable:

> **NOTE**: Return values should always be checked. This example does not check for errors.

```c
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

typedef struct Data
```

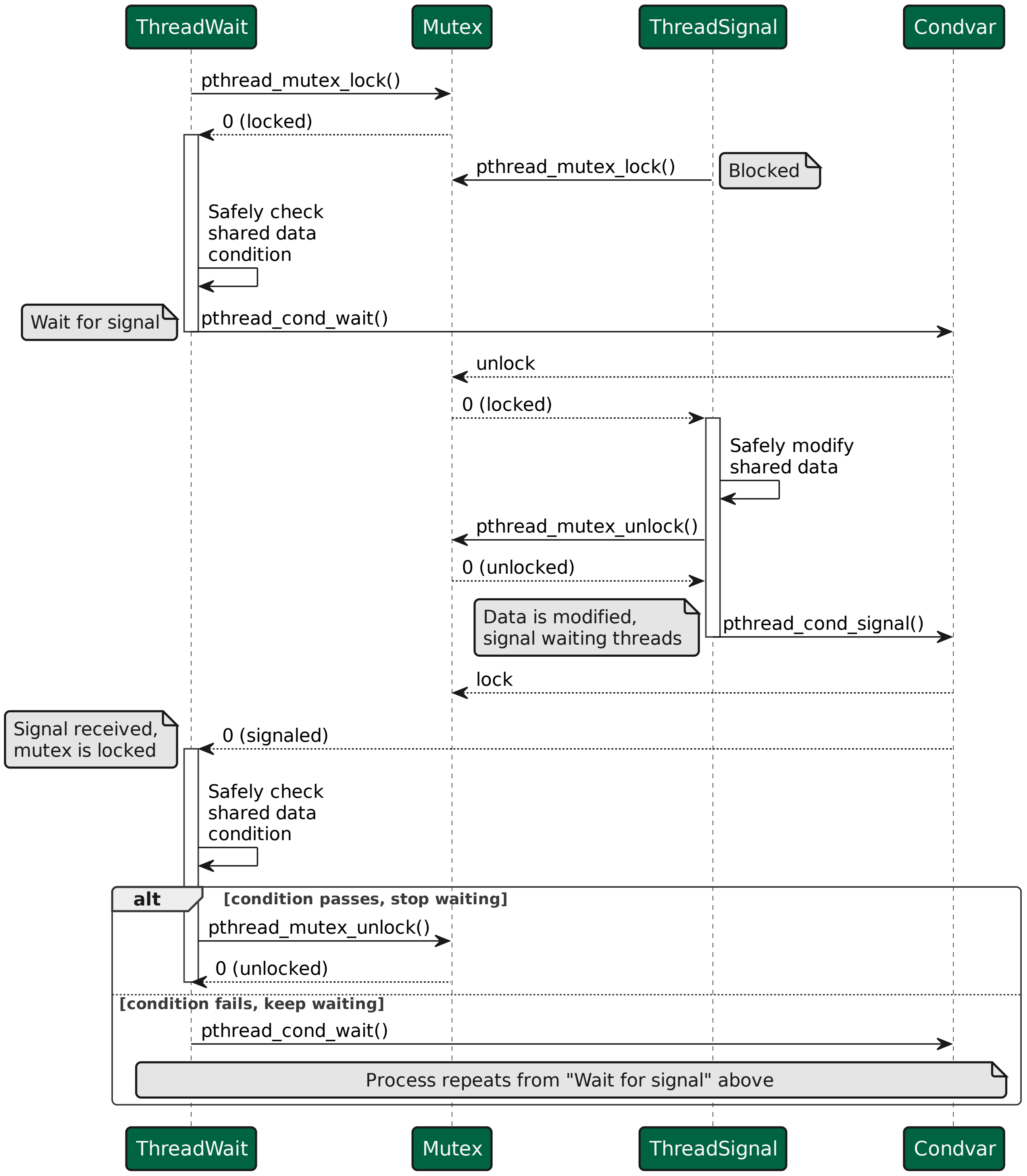

Figure 3: Condition variable sequence with multiple threads

```
{
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
} Data;

void *my_thread_example_wait(void *opt)
{
    Data *thread_data = (Data *)opt;

    pthread_mutex_lock(&thread_data->mutex);
    while (thread_data->value != 2)
    {
        printf("A: Waiting for condition %d == 2\n", thread_data->value);
        fflush(NULL);
        pthread_cond_wait(&thread_data->cond, &thread_data->mutex);
    }
    printf("A: Signal received. Data value is %d\n", thread_data->value);
    fflush(NULL);
    pthread_mutex_unlock(&thread_data->mutex);

    return NULL;
}

void *my_thread_example_signal(void *opt)
{
    Data *thread_data = (Data *)opt;

    pthread_mutex_lock(&thread_data->mutex);
    printf("B: Modifying data from %d to 2\n", thread_data->value);
    fflush(NULL);
    thread_data->value = 2;
    pthread_mutex_unlock(&thread_data->mutex);
    printf("B: Signalling threads that value is now %d\n", thread_data->value);
    fflush(NULL);
    pthread_cond_signal(&thread_data->cond);

    return NULL;
}

int main(void)
{
    Data my_data = {.value = 1};
    pthread_mutex_init(&my_data.mutex, NULL);
    pthread_cond_init(&my_data.cond, NULL);

    pthread_attr_t pattr;
    pthread_attr_init(&pattr);

    pthread_t id_a = 0;
    pthread_t id_b = 0;

    pthread_create(&id_a, &pattr, my_thread_example_wait, &my_data);
    pthread_create(&id_b, &pattr, my_thread_example_signal, &my_data);

    pthread_join(id_a, NULL);
    pthread_join(id_b, NULL);

    return 0;
}
```

Output of the example is

```
A: Waiting for condition 1 == 2
B: Modifying data from 1 to 2
B: Signalling threads that value is now 2
A: Signal received. Data value is 2
```

In this example the condition

Best Practices

- If using dynamic memory (so far, labs have not used any dynamic memory allocation):
  - Allocate all dynamic memory related to shared data before starting threads.
  - Do not resize shared data during execution of threads.
  - Initialize mutexes before thread creation and after memory allocations.
- Lock and unlock mutexes in the same scope and the same thread.
- Use condition variables for signalling to other threads that some data is ready to check or modify.
- Group data being shared between threads in a `struct` with associated mutexes and condition variables.
- Create and join all threads with common data from a single manager thread (often the main process).
- Avoid global variables. Pass data to threads from a manager (main) function instead.
- Lock mutexes for as short amount of time as possible. A good strategy is to only have mutex locked to copy data to a local variable so that other threads do not have to wait for the resource to become available.
- Don't use multiple threads if it's not necessary.
- Do not access or modify data concurrently by multiple threads if it's not necessary.

Coding Task: FreshThreads Background

Direct-to-consumer brands are thirsty to sell with large markups when demand is high but social media influencers can change a product's appeal so fast, it's hard to nail down a price! They decide to hire an embedded real-time systems software developer to help, for … reasons. It turns out the developer had not taken CST8703 and created a questionable simulation that used threads in a (quite possibly, most likely, definitely, horribly) wrong way. It's your job to make the simulation great again and fix your predecessor's mistakes. What's worse, nobody could remember what they're even selling anymore (not that it matters), so it's a dubiously abstract problem with an abstract product. What we do know is the product is fresh and involves threads. What does it matter if you're getting paid, anyways? (spoiler alert: you're not actually going to be paid for this work).

The rudimentary simulation consists of a pool of influencers on social media implemented using one thread per influencer acting on common shared data. Each influencer affects the "desirability factor" for a product which is simply a real number between 0.0 and 1.0. The change in the product's desirability is governed by a random variable v with a uniform distribution between a $-\alpha$ and α, where α is an adjustment factor between 0.0 and 0.5. If v_k is positive after k influencers have already promoted the product, the next influencer $(k+1)$ updates the latest desirability factor D_k as follows:

$$D_{k+1} = D_k + v_k(1.0 - D_k)$$

If the factor v is negative, the desirability is modified by:

$$D_{k+1} = D_k(1.0 + v_k).$$

There is a pool of N influencers. The simulation checks what price the product can be set to based on the desirability after M influencers showcase the product. The cost is set between a minimum value C_{min} and maximum value C_{max} scaled by the current desirability. The simulation is set up with the following initial values:

Parameter	Value	Meaning
N	1000	Number of influencers (threads in pool)
M	50	Number of influencer promotions before cost appraisal
D_0	0.5	Initial desirability factor
C_{min}	\$20	Minimum cost
C_{max}	\$500	Maximum cost
α	0.2	Desirability change factor
s	1234	Integer chosen as initial seed to the random number generator

Your task is to run the simulation and identify the problem, then implement safe thread syncronizaiton with Mutexes and Condition Variables. Follow the methods below.

Materials

1. Raspberry Pi connected to local network.
2. Desktop or laptop computer with Linux, Windows, or Mac operating system.
3. Wired or wireless local network.

Methods

Prerequisites

Clone source from GitHub repository https://github.com/chishok/CST8703-Lab2.

Coding Task: Make FreshThreads Thread-Safe

Without modifying the original source, build and run the target `FreshThreads` using the script command

```
./scripts/run_submission.sh
```

The program FreshThreads should build, run and generate a figure at the path `data/desirabilityLog.png`. An example of an output plot is

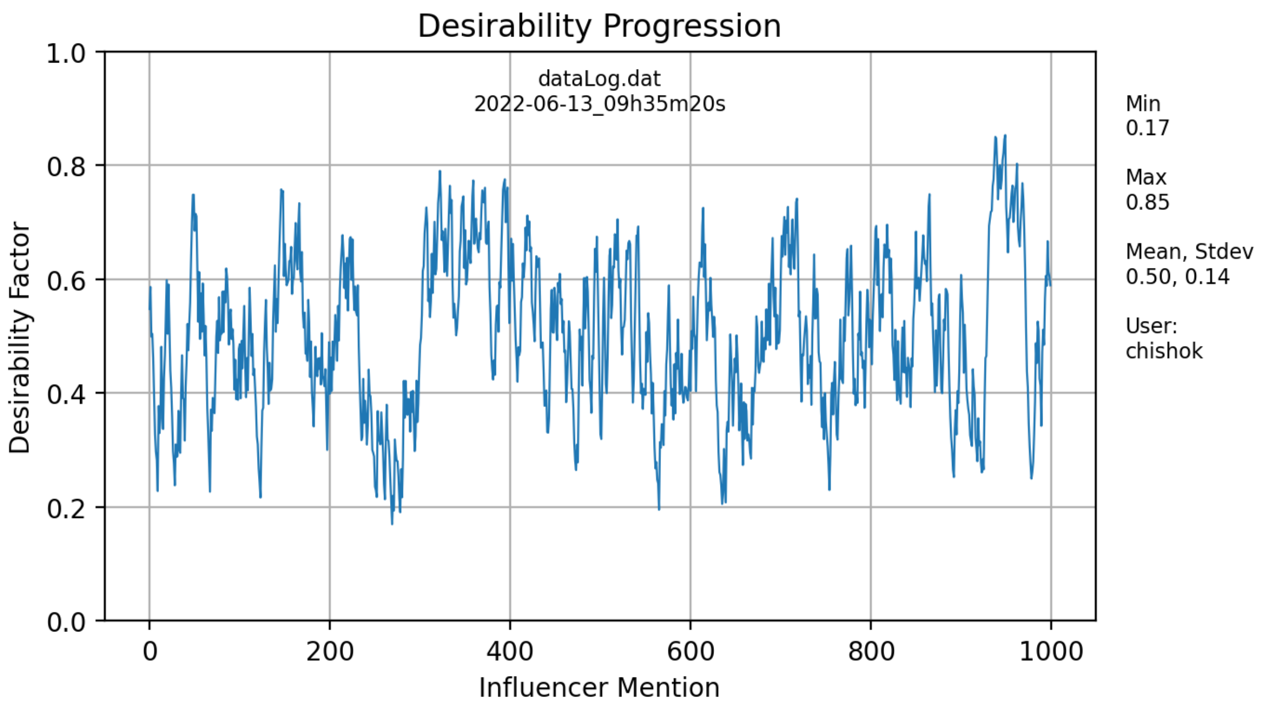

Figure 4: FreshThreads simulation plot result without safe thread synchronization

The plot shows the progression of the desirability factor after every influencer mention of the product. The algorithm was intended to provide a random walk around a desirability of 0.5 but there is clearly a problem with the result.

Alter the target program `FreshThreads` to safely access data from multiple threads. Make your modifications between all section with the comments

```
    /*
     * STUDENT WORK SECTION BEGIN
     * ===========================
     */
```

and

```
    /*
     * =========================
     * STUDENT WORK SECTION END
     */
```

The comments provide explicit instructions that must be followed to render the program thread-safe.

There are multiple source files in this example and the majority of the work is required in the file `src/ac_fresh.c`.

Once you've made the changes to the source code, run

```
./scripts/run_submission.sh
```

The expected contents of `data/output.txt`:

```
Arguments parsed
  filename: data/dataLog.dat
Thread info:
    thread name: CST8703Lab2-Fre
    thread policy: SCHED_OTHER
    thread priority: 0
    thread id: 140124116076288
    process id: 13902
    timestamp: 2022-06-13 23:28:24
Fresh Threads Social Marketplace
================================
Inputs:
    influencer change factor: 0.20
    number of influencers: 1000
    minimum cost: $20.00
    maximum cost: $500.00
Appraisal Results:
    latest desirability: 0.589209
    latest cost: $302.82
    appraisal after 50 influencer mentions:
        desirability 0.748357
        cost $379.21
```

The expected output plot at `data/desirabilityLog.png`:

Analysis

1. Compare the output desirability factor plots before and after the mutexes and condition variables were implemented.
    1. What was the issue when there was no thread synchronization?
    2. Why did the values drop to zero so frequently?
    3. How did adding mutexes solve the problem?
2. Evaluate modifications to the simulation parameters.
    1. In the `src/main.c` file line 34, rebuild and run with a different seed value. How does the output plot change when a new seed is provided to the pseudo random number generator?
    2. When the program is re-run multiple times with the same seed, does the output change? Are different random numbers generated for each run? Why?

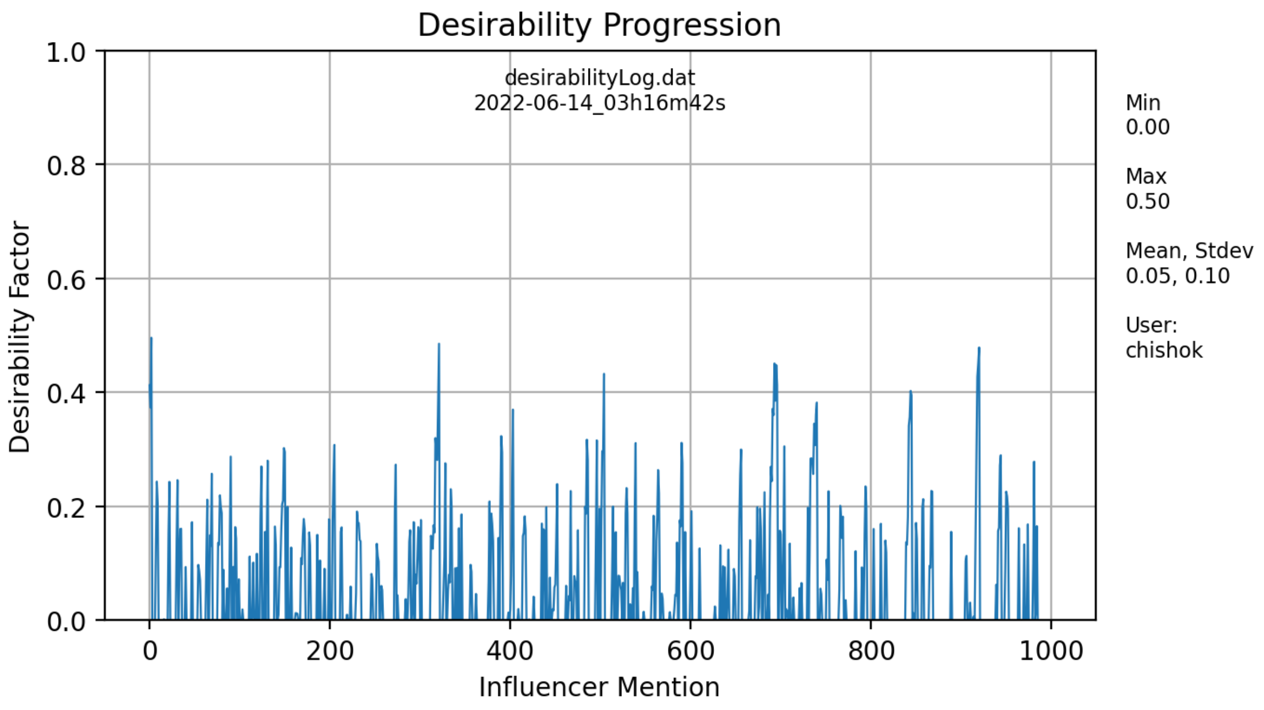

Figure 5: FreshThreads simulation expected output plot result

3. If the influencer change factor is modified in file `src/main.c` file line 36 to 0.5 and then again for 0.05. What changes do you see in the output desirability plot?

3. Reset seed and change factors to original values:

```
data.influencer.rng_seed = 1234;
data.config.influencer_change_factor = 0.05;
```

4. Comment on the effectiveness and validity of the simulation:

    1. Could this simulation be run on a single thread in a simple for loop?
    2. If you were to design the simulation from the ground up, would you recommend using multiple threads? Why or why not?
    3. What is meant by a pseudo-random number generator? Is it valid to use random number generators when simulating real-world dynamic systems?
    4. Speculate how one might validate the results of the simulation with real-world data.

You've completed your contract with the social media brand representative. Their marketing team is so impressed with your work that they promote it on an influential blog site and across social media platforms. The piece gets picked up Ars Technica, and they reach out for an interview. What do you really think about your work on this project? Do you tell the reporter that the algorithm was flawed from the beginning or do you promote your burgeoning career as a social media marketing and predictions specialist?

Submission

Include your modified source code and a lab report. If you forked the repo on your personal GitHub, commit and push your changes, then download your repo as a zip file. If you do not use GitHub, compress the source code in a zip file or tar file. Be sure that the command `./scripts/run_submission.sh` produces the expected output.