Borislav Sabotinov
CS5341 HW2

Complete source code and instructions are included in the submission ZIP.

They may also be found on GitHub here: https://github.com/bss8/tcp-nagle-and-threads

1. (25 pts) Please write a short program that will disable Nagle's algorithm. The program can be adapted from the client server code in the textbook (such as from Chapter 5, but be sure not to use the fgets and fputs functions as they belong to C standard library and buffer input and output). Describe the behaviours of the program after Nagle's algorithm is disabled and after Nagle's algorithm is enabled.

   Enabling the TCP_NODELAY option turns Nagle's algorithm off: "If set, this option disables TCP's Nagle algorithm (Section 19.4 of TCPv1 and pp. 858 859 of TCPv2). By default, this algorithm is enabled." (p. 268).

   The client.cpp and the first part of the server.cpp code serves to implement this.

```
**
 * disables Nagle's algorithm, although we really only need to worry about this on the sender (client end).
 * It is done here for practice and convenience.
 * if TCP_NODELAY is set (on), the algorithm is considered turned off.
 */
void disable_nagle_alg(int sockfd)
{
    int isEnabled = 1;
    int set_res = setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, (char *)&isEnabled, sizeof(isEnabled));
    if (set_res < 0)
        std::cerr << "Error disabling Nagle's algorithm via setting socket option for TCP_NODELAY!"
                  << strerror(errno) << std::endl;
}
```

   Not much of a difference is observed because of the small payload sent and the communication taking place on LAN. I tested locally on my machine and on TXST Linux servers but those are also technically LAN because Eros and Zeus are very close to each other and there is no delay/lag.
   Theoretically, what disabling Nagle's algorithm does is allow us to stream data like video or play multi-player video games with less lag.

Disabling Nagle is not very effective when communicating in one direction. In two directional communication, disabling this algorithm improves throughput. Why? Because it removes delays, which can accumulate as each node can send its response slightly quicker. This allows the other side to respond even earlier than if the algorithm was enabled and we were stuck waiting for ACK to be received before sending the next packet.

2. (5 + 7 + 8 = 20 pts) This problem is pertaining to pthreads. You need to write and submit a program to test and support your answer.
    a. (1) Can a thread still exist if the thread that creates it terminates by calling pthread exit()?

        Yes, that is one of the intents of the pthread_exit() function. The main() thread is a thread on its own. If, for example, we want our main thread to launch a number of other threads to do some processing and then terminate the main thread, which we only use to launch others, pthread_exit() is what we would use.

        Here is what it looks like:

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./server
Question 1 - program regarding Nagle algorithm.
server: got connection from 127.0.0.1 on port 53574
Client msg: test
Question 2 - program regarding pthreads.
Thread Function :: Start
non detached, non-main thread increment: 0
non detached, non-main thread increment: 1
non detached, non-main thread increment: 2
non detached, non-main thread increment: 3
non detached, non-main thread increment: 4
non detached, non-main thread increment: 5
non detached, non-main thread increment: 6
non detached, non-main thread increment: 7
non detached, non-main thread increment: 8
non detached, non-main thread increment: 9
detached, non-main thread increment: 0
detached, non-main thread increment: 1
detached, non-main thread increment: 2
detached, non-main thread increment: 3
detached, non-main thread increment: 4
detached, non-main thread increment: 5
detached, non-main thread increment: 6
detached, non-main thread increment: 7
detached, non-main thread increment: 8
detached, non-main thread increment: 9
Thread Function :: End
```

Figure 1: pthread_exit on main()

The other threads continue while main terminates (we do not see the for-loop in main() execute here).

b. (2) Can a detached thread still exist if the main thread of the whole process terminates by calling exit function?

Detaching a thread does not permit it to continue existing past process termination of the main thread. Calling exit() will destroy the detached thread along with all other threads spawned. Thus a call to pthread_detach() on a created thread will not ensure it continues after exit() is called from main().

```
^[[Aboris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./server
Question 1 - program regarding Nagle algorithm.
server: got connection from 127.0.0.1 on port 53584
Client msg: test
Question 2 - program regarding pthreads.
Thread Function :: Start
Thread Function :: Start
```

Figure 2: calling exit() in main()

We do not see any printouts here, from either main or the threads it launched.

c. (3) Will the process still continue if one of the normal (not detached) and non-main threads within it calls exit function?

Yes; I tested this and if thread_function_one(void *arg) calls exit(0), the main function still continues and prints out the following (which occurs after the non-detached, non-main thread exits). The thread launched by main terminates while main continues.

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./server
Question 1 - program regarding Nagle algorithm.
server: got connection from 127.0.0.1 on port 53592
Client msg: test
Question 2 - program regarding pthreads.
Thread Created with ID : 140495259215616
non detached, main thread increment: 0
non detached, main thread increment: 1
non detached, main thread increment: 2
non detached, main thread increment: 3
non detached, main thread increment: 4
non detached, main thread increment: 5
non detached, main thread increment: 6
non detached, main thread increment: 7
non detached, main thread increment: 8
non detached, main thread increment: 9
Thread Function :: Start
```

Figure 3: Call exit() from non-main non-detached function

d. (4) Can the original process still continue (hence all other threads within the process) if a detached thread calls exit function to terminate?

Yes, it appears so. Detaching a thread and then calling exit() in that thread allows main to continue execution. We do not get any output/processing from the detached thread as it terminates but main continues to process and we see the output.

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./server
Question 1 - program regarding Nagle algorithm.
server: got connection from 127.0.0.1 on port 53600
Client msg: test
Question 2 - program regarding pthreads.
Thread Created with ID : 140250193327872
non detached, main thread increment: Thread Function :: Start
0
non detached, main thread increment: 1
non detached, main thread increment: 2
non detached, main thread increment: 3
non detached, main thread increment: 4
non detached, main thread increment: 5
non detached, main thread increment: 6
non detached, main thread increment: 7
non detached, main thread increment: 8
non detached, main thread increment: 9
non detached, non-main thread increment: 0
non detached, non-main thread increment: 1
non detached, non-main thread increment: 2
non detached, non-main thread increment: 3
non detached, non-main thread increment: 4
non detached, non-main thread increment: 5
non detached, non-main thread increment: 6
non detached, non-main thread increment: 7
non detached, non-main thread increment: 8
non detached, non-main thread increment: 9
```

Figure 4: detached thread calls exit()

We see output from main and from the thread before it was detached but no output from the thread once it's detached and exit() is called.

e. (5) Can the process still continue if the main thread terminates normally (ie the control of the main thread falls off the last statement of the main function)? Can the main thread detach itself?

No, the main thread will die when its process returns from the main function, regardless of whether it is detached or not.

thread_test.cpp:

```
#include <iostream>
#include <string>
#include <thread>
#include <chrono>

void some_thread() {
  std::this_thread::sleep_for (std::chrono::seconds(2));
  std::cout << "Do you see me?" << std::endl;
}

int main()
{
    std::thread t1(some_thread);
    t1.detach();

    return 0;
}
```

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ make
g++ -pthread -o tclient readline_client.cpp -g -std=c++11
g++ -pthread -o mclient modified_client.cpp -g -std=c++11
g++ -pthread -o thread thread_test.cpp -g -std=c++11
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./thread
```

Figure 5: Test if thread continues once main terminates normally

The message will never print because the operating system will clean up some_thread() once the main() function exits.

3. (15 + 15 = 30 pts) This problem is pertaining the thread-specific data technique discussed in class.
    a. (1) Compile and run the thread version of TCP client-server program that uses the thread-safe readline function in Fig.26.11 and 26.12 (p.692 & p.693). Verify (print if possible) the value returned in the variable rl_key. Does each thread have the same rl_key value?

       Yes, multiple threads use the same key, but get different storage space per thread. The pthread_key_t rl_key variable is static, so there is only one of it shared between all instances of a class or threads.

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./mclient localhost 5001
rl_key before pthread_key_create call: 0x558baea7e3f0
rl_key after pthread_key_create call: 0x558baea7e3f0
rl_key before pthread_setspecific call: 0x558baea7e3f0
rl_key after pthread_setspecific call: 0x558baea7e3f0
value of boris_key thread specific data that we set: Just some data
Hello world! Repeat String!
 Repeat String!
 Repeat String!
 Repeat String!
 Repeat String!
test
```

    b. (2) Can a thread use more than one thread-specific data item? Modify the program in (1) to demonstrate your answer. Please provide detailed comments explaining the purpose of each thread-related function call.

       We can create as many thread-specific data items as we want (within reason): “Each system supports a limited number of thread-specific data items. POSIX requires this limit be no less than 128 (per process)” (p. 795).

       How to create a thread specific data item? “A thread calls pthread_key_create to create a new thread-specific data item, the system searches through its array of Key structures and finds the first one not in use.” (p. 796).

```
// I create a new pthread_key_t variable to hold the key for the additional thread specific data.
static pthread_key_t boris_key;


    // Additional thread specific data with a custom separate key

    // the boris_key is the key and the data will be some simple string.
    // I am testing simply if it can be created and accessed.
    pthread_key_create(&boris_key, readline_destructor);

...

// I am using some simple string to test adding additional thread specific data.
        // I just generate another unique key (which gets created only once with pthread_once
        // but can be used by multiple threads to store thread specific data)
        std::string thread_specific_string = "Just some data";

        // we set the data associated with this key to be the string above
        pthread_setspecific(boris_key, &thread_specific_string);

        // Here I test accessing the data we just created by using the key, which we
        // pass to the pthread_getspecific function. It returns to us a pointer to where the data is
        // stored in memory. When we print it out, we must cast it to the right value but we recall it is a
        // pointer so we must also dereference it.
        const void *p = pthread_getspecific(boris_key);
        std::cout << "value of boris_key thread specific data that we set: " << *(const std::string *) p << std::endl
;
```

4. (15 + 10 = 25 pts)
   a. (1) *The readline function in Fig.26.12, p.693 calls **pthread_once** at line 42. What is the purpose of this function call?*

   At it's most basic, the manual page for the pthread_key_create function tells us directly that it is the programmer's responsibility to ensure it is called only once: "The *pthread_key_create*() function performs no implicit synchronization. It is the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization functions, using mutexes, and using *pthread_once*()."

   Thus, we see that pthread_once() is a mechanism for ensuring we only ever invoke something one time during execution. Then it logically follows we can place the call of pthread_key_create inside a function that is called by pthread_once(). This is what we do – we call pthread_once, which as its second argument takes in a void (*__init_routine)() or a function. We pass it readline_once, and inside this function we call pthread_key_create. Thus we ensure it is only ever called once.

   For further detail, we turn to the text:

   We first call ***pthread_once*** so that the first thread that calls **readline** in this process calls ***readline*** once to create the thread-specific data key. We will use the pthread_once function to guarantee that pthread_key_create is called only by the first thread to call readline. readline calls pthread_once to initialize the key for this thread-specific data item, but since it has already been called, it is not called again. The first two functions that are normally called when dealing with thread-specific data are pthread_once and pthread_key_create. pthread_once is normally called every time a function that uses thread-specific data is called, but pthread_once uses the value in the variable pointed to by onceptr to guarantee that the init function is called only one time per process. (p. 798)

   "Every time readline is called, it calls pthread_once. This function uses the value pointed to by its onceptr argument (the contents of the variable rl_once) to make certain that its init function is called only one time. This initialization function, readline_once, creates the thread-specific data key that is stored in rl_key, and which readline then uses in calls to pthread_getspecific and pthread_setspecific." (p. 799)

   *What will happen if that function is not called? Does the **readline** function still work correctly without making the function call?*

My program does appear to still work correctly but that could be because there is no 2nd thread that comes in and tries to generate a key again. We recall from the text it is the programmer's responsibility to ensure pthread_key_create is called only once but commenting out pthread_once can create undefined/unexpected behavior. While I personally did not observe it in testing, I cannot rule out that it could happen. So pthread_once should remain uncommented and should be used!

Client with commented out pthread_once:

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./tclient localhost 5001
rl_key before pthread_setspecific call: 0x55ca92c033f0
rl_key after pthread_setspecific call: 0x55ca92c033f0
Hello world! Repeat String!
rl_key before pthread_setspecific call: 0x55ca92c033f0
rl_key after pthread_setspecific call: 0x55ca92c033f0
test message to server
```

Server receiving message from client:

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2  (main)
$ ./server
Question 1 - program regarding Nagle algorithm.
server: got connection from 127.0.0.1 on port 53764
Client msg: test message to server

Question 2 - program regarding pthreads.
Thread Created with ID : 140142826202880
non detached, main thread increment: 0
non detached, main thread increment: 1
non detached, main thread increment: 2
non detached, main thread increment: 3
non detached, main thread increment: 4
non detached, main thread increment: 5
non detached, main thread increment: 6
non detached, main thread increment: 7
non detached, main thread increment: 8
non detached, main thread increment: 9
Waiting for client connection....
Thread Function One :: Start
non detached, non-main thread increment: 0
non detached, non-main thread increment: 1
non detached, non-main thread increment: 2
non detached, non-main thread increment: 3
non detached, non-main thread increment: 4
non detached, non-main thread increment: 5
non detached, non-main thread increment: 6
non detached, non-main thread increment: 7
non detached, non-main thread increment: 8
non detached, non-main thread increment: 9
detached, non-main thread increment: 0
detached, non-main thread increment: 1
detached, non-main thread increment: 2
detached, non-main thread increment: 3
detached, non-main thread increment: 4
detached, non-main thread increment: 5
detached, non-main thread increment: 6
detached, non-main thread increment: 7
detached, non-main thread increment: 8
detached, non-main thread increment: 9
Thread Function One :: End
```

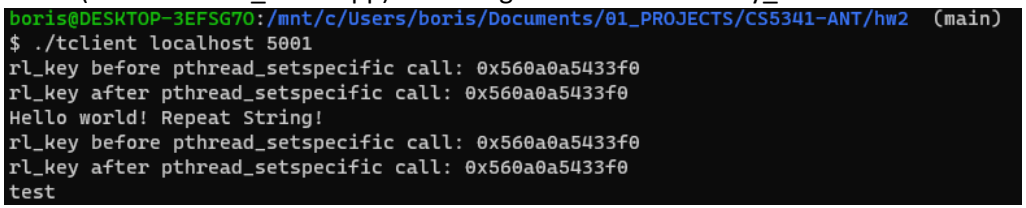
b. *(2) Why is the return value of the **my_read** function (called by **readline**) of static type? What is the consequence of changing it to no-static?*

The internal function my_read reads up to MAXLINE characters at a time and then returns them, one at a time. By using static variables in readline.c to maintain the state information across successive calls, the functions are not re-entrant or thread-safe. Why is this not thread safe? A function that keeps state in a private buffer, or one that returns a result in the form of a pointer to a static buffer, is not thread-safe because multiple threads cannot use the buffer to hold different things at the same time.

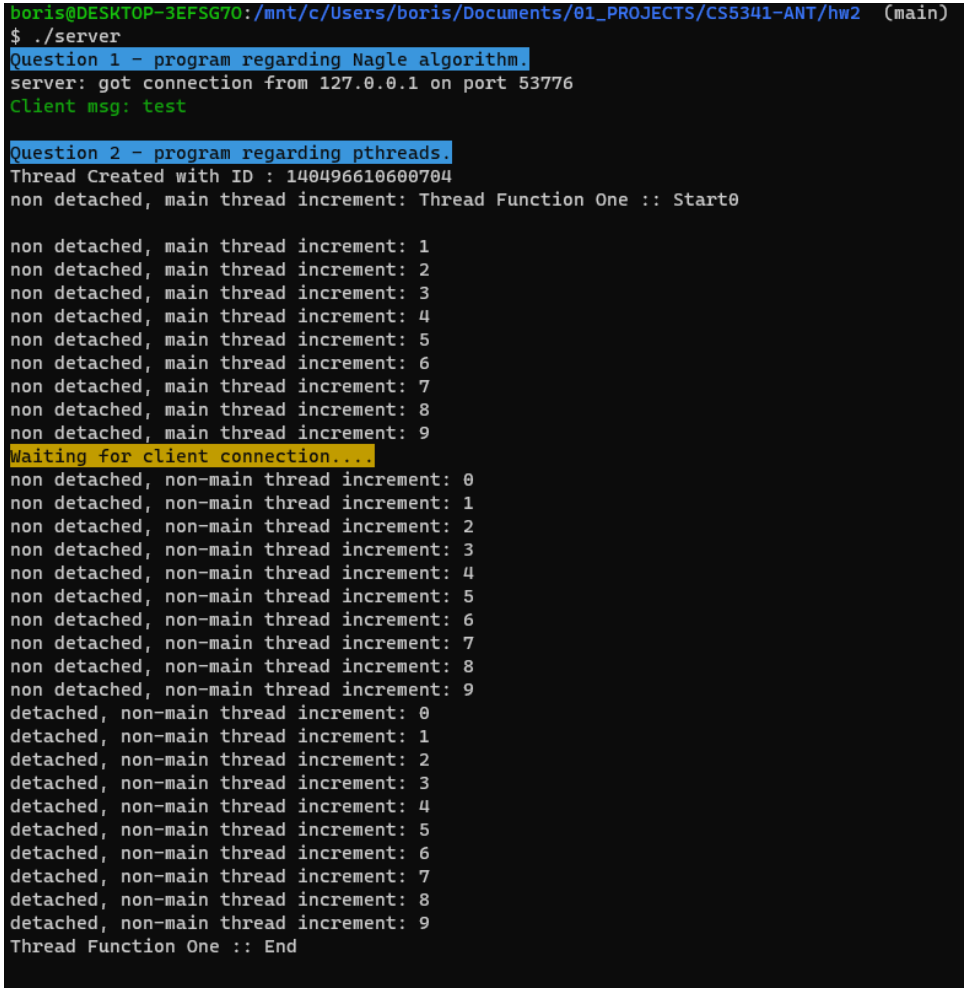
But the way we make my_read thread safe and re-entrant isn't through removing the static variable. We observe Figure 3.18 and Figure 26.12 and compare. The latter my_read has an added 1st argument – Rline *tsd. The first argument is a pointer to the Rline structure that is allocated for this specific thread, containing thread-specific data. Thus we get around the issue of other threads potentially overwriting our data by maintaining it locally to the thread itself.

What happens when we remove the static declaration for this function? From my observation, nothing of note – the program worked as expected. Again it could be because I do not have multiple threads competing and I do not have race conditions. It is just the one client sending one message to the one server. But the code compiles fine and executes without any runtime error. The client runs, sends a message to the server, which In turn receives the message as expected.

Client (from readline_client.cpp) executing with a non-static my_read function:



Server receiving the client message:



Unrelated to the questions above but interesting to note, if the program is prematurely terminated after a bind, the operating system will place a TIME_WAIT on the port and it will be unavailable until that time elapses.

```
boris@DESKTOP-3EFSG7O:/mnt/c/Users/boris/Documents/01_PROJECTS/CS5341-ANT/hw2
$ netstat --tcp --numeric | grep 5001
tcp        0      0 127.0.0.1:5001          127.0.0.1:58844         TIME_WAIT
tcp        0      0 127.0.0.1:58844         127.0.0.1:5001          TIME_WAIT
```

References

1. https://stackoverflow.com/questions/37240869/get-the-content-of-a-const-void
2. https://stackoverflow.com/questions/9005955/how-does-pthread-key-t-and-the-method-pthread-key-create-work
3. W. R. Stevens, Bill Fenner, and Andrew M. Rudoff. UNIX Network Programming – Networking APIs: Sockets and XTI (3nd ed.). Addison-Wesley, 2004. ISBN: 0-13-141155-1.
4. https://www.man7.org/linux/man-pages/man3/readline.3.html
5. https://man7.org/linux/man-pages/man3/pthread_once.3p.html
6. https://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_key_create.html
7. https://linux.die.net/man/3/pthread_key_create
8. https://linux.die.net/man/3/pthread_setspecific
9. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_setspecific.html
10. https://linux.die.net/man/3/pthread_getspecific