Assignment 3 Threads

Operating Systems - Fall '23

Contents

1 Introduction **3**

1.1 Why Threading Matters 3

1.2 Rationale for Implementation 3

2 Objectives **3**

3 Overview **4**

4 Getting Started **4**

4.1 A brief description of the TCB and the fields that are required for the implementation 4

4.2 A brief description of the libraries we are going to use 5

4.3 Information about the scheduling algorithm 5

5 API Specification **6**

5.1 Threading 6

5.1.1 `void init_lib ()` 6

5.1.2 `int thread_create (thread_t *thread, void (*start_routine))` 6

5.1.3 `void thread_exit ()` 7

5.1.4 `void thread_join (thread_t *thread)` 8

5.1.5 `void thread_sleep (unsigned int milliseconds)` 8

5.1.6 `void thread_yield (void)` 8

5.1.7 `void context_switch (thread_t *old, thread_t *new)` 9

5.2 Control Primitives 9

5.2.1 `void mutex_init (mutex *m)` 9

5.2.2 `void mutex_acquire (mutex *m)` 9

5.2.3 `void mutex_release (mutex *m)` 9

5.2.4 `void sem_init (semaphore *sem, int value)` 9

5.2.5 `void sem_wait (semaphore *sem)` 9

5.2.6 `void sem_post (semaphore *sem)` 9

6 Problem Solving **10**
6.1 The Doctor's Waiting Room . . . 10
6.2 TipTop Dry Cleaning Scheduler . . . 11
7 Important Information **12**
7.1 Time Slice . . . 12
7.2 Dealing with Interrupts . . . 12
7.3 The Mangle Function . . . 12
7.4 The `setjmp.h` Library . . . 12
7.5 Architecture . . . 13
7.6 The `LOG_OUT` function . . . 14
7.7 Variable Intialization . . . 14
8 Setup **15**
8.1 Docker Container Creation: . . . 15
8.2 Docker Container Creation: . . . 15
8.3 Troubleshooting on MacOS . . . 15
8.4 Testing . . . 15
9 Submission **16**

1 Introduction

Embark on a foundational journey into the realm of threading concepts with this assignment — a cornerstone in the expansive landscape of computer science. Whether you're threading through the intricate paths of development, navigating the avenues of research, or even overseeing the fundamental tapestry of system administration, the significance of threading resonates across diverse domains.

Your mission is both simple and profound: forge a refined rendition of the `pthread` library, a pivotal thread in the fabric of computing. Delve into the ensuing sections, where meticulous details await to guide your odyssey through implementation.

1.1 Why Threading Matters

Threading stands as a cornerstone in computer science, particularly within the realm of Operating Systems. It facilitates the concurrent execution of programs, a pivotal capability in optimizing system performance. To comprehend its significance, consider a scenario with a basic example.

Picture yourself immersed in a chat application, like WhatsApp, on a system devoid of the magical touch of concurrency. Here, operations unfold in a rigidly sequential manner—either sending a message or receiving one at any given moment. This rigid structure translates to the inability to compose a message while concurrently receiving another, and vice versa.

Now, shift your perspective to an environment fortified with threading capabilities. In this empowered setting, the same chat application transcends limitations. Users can seamlessly engage in both sending and receiving messages concurrently. This transformative capability not only enhances the overall user experience but also bestows upon applications a heightened level of functionality. This is particularly crucial in scenarios where fluid interaction with users is paramount.

Threading, by unlocking the doors to parallel operations, emerges as a linchpin. It not only contributes to the creation of responsive systems but also lays the foundation for crafting applications that are not just efficient but also user-friendly.

1.2 Rationale for Implementation

While it's true that pthread, a robust threading library, already exists, there are nuanced considerations for creating your own implementation. As a computer scientist, delving into the mechanics of fundamental concepts and translating that understanding into practical implementation is integral to your expertise.

The pthread library stands as a pivotal tool, but certain scenarios prompt the need for a bespoke threading solution. Consider instances where access to any threading library is restricted, such as in embedded systems. Additionally, a customized threading library proves invaluable when program requirements deviate—perhaps necessitating low-latency performance or adopting a specialized thread scheduling system. In such contexts, the development of a tailored threading library becomes imperative to meet specific program demands.

Hence, this assignment not only cultivates a deep comprehension of threading principles but also equips you with the versatility to address diverse programming scenarios where custom solutions are indispensable.

2 Objectives

As you embark on this assignment, we assume a foundational familiarity with the C programming language. However, if you are not already proficient, this task is crafted to be a learning vehicle, ensuring that you acquire the necessary skills.

By the culmination of this assignment, our objectives are for you to adeptly:

1. **Comprehend Thread Execution:** Gain a comprehensive understanding of how individual threads operate, each with its distinct context.

2. **Navigate Multithreading:** Acquire insights into the orchestration of multiple threads within the confines of a single CPU.
3. **Master Thread Scheduling:** Familiarize yourself with the intricate art of scheduling threads for optimal system performance.
4. **Implement Concurrency Control Mechanisms:** Grasp the conceptual intricacies and practically implement concurrency control mechanisms.
5. **Problem Solving through Concurrency Control:** Apply acquired knowledge to address challenges, employing concurrency control mechanisms as strategic problem-solving tools.
6. **Craft Robust Programs in C:** Develop programming proficiency by creating clear, readable, well-documented, and well-designed programs using the C programming language.

By achieving these objectives, you not only deepen your comprehension of threading principles but also hone practical skills essential for constructing efficient and reliable programs.

3 Overview

For the sake of simplicity and a systematic approach, the assignment is subdivided into three essential parts. It is imperative to complete each part sequentially, as they build upon one another. All parts are compulsory, ensuring a comprehensive understanding of the threading concepts under examination.

Part	**Brief description**	**Grade weightage**
Threading Primitives	Implement simple threading primitives and scheduling	50%
Control Primitives	Implement concurrency and synchronization primitives	20%
Problem Solving	Solve real world problems using the threading library	30%

This assignment is to be approached as an individual endeavor. Strict adherence to this policy is vital, and any form of plagiarism is unequivocally prohibited. Students are expressly forbidden from engaging in discussions regarding their solutions with peers or consulting external sources on the internet.

However, we recognize that challenges may arise during the assignment. In such instances, students are encouraged to seek assistance exclusively from the course staff. Moreover, referencing documentation related to libraries is permissible and encouraged to augment the learning process.

Adhering to these guidelines ensures a fair and equitable assessment of individual comprehension and skill development.

4 Getting Started

4.1 The TCB

In alignment with your studies on process management in operating systems, an analogous mechanism is employed for user-level threading libraries. Instead of Process Control Blocks, we employ Thread Control Blocks (TCBs) in this assignment. The TCB, denoted by the struct thread_t, serves as a vital repository for essential information.

Key elements to be tracked within the TCB include the thread ID, the thread's current state, the allocated stack for the thread, the program counter (initially set to the address of the designated function to be executed), and the stack pointer.

The following fields are mandatory in the TCB:

- `int thread_id:` Identifying the unique ID of the thread.
- `enum THREAD_STATE state:` Reflecting the current state of the thread.

While you have the flexibility to incorporate additional information into the TCB as needed for your implementation, it is imperative not to omit or modify the provided fields. These fields serve as the basis for evaluating your code through testing, and any removal may result in non-assignation of marks for the associated test cases.

Feel free to augment the TCB with any pertinent information by utilizing the provided files. Ensure, however, that the prescribed fields are retained for comprehensive testing and evaluation.

You also need to use the provided `thread_t* current_thread` variable to keep track of the current thread.

4.2 Libraries and Implementation

Before any queries arise on Slack, it is imperative to note that the utilization of pthread functions or any library that independently implements thread-like functionalities is strictly prohibited for this assignment.

For those seeking a helpful starting point, a strong recommendation is to delve into the functionalities of `setjmp.h`. When used in conjunction with `signal.h`, it can provide a foundational basis for creating a functional threading library—consider this a subtle nudge in the right direction.

To aid in your exploration, the documentation for these libraries is accessible:

- `setjmp.h`: Link
- `signal.h`: Link

To facilitate an engaging and constructive tutorial session, it is advisable to familiarize yourself with the documentation beforehand. This proactive approach ensures a more interactive and productive tutorial session.

4.3 The Scheduler

A crucial component of every threading library is the scheduler, tasked with the management of threads and the pivotal decision of determining which thread to execute next. For this assignment, the prescribed scheduling algorithm is the Round Robin scheduler.

It is imperative to note that all test cases are designed with the assumption that you are utilizing a Round Robin scheduler. Deviating from this algorithm will result in non-assignation of marks for the associated test cases.

For those eager to delve deeper into threading concepts, various scheduling algorithms offer insights into diverse strategies. While the following scheduling algorithms are intriguing:

- First Come First Serve
- Shortest Job First
- Shortest Remaining Time First
- Priority Based Scheduling
- Multi-Level Queue Scheduling
- Multi-Level Feedback Queue Scheduling

Regrettably, credit will not be awarded for implementing these advanced scheduling algorithms in this assignment. The focus remains on the Round Robin scheduler as specified.

5 API Specification

In the development of your threading library, you enjoy the freedom to introduce as many helper functions as needed, provided they strictly adhere to the API specification. Furthermore, the latitude extends to the inclusion of global variables; however, the sanctity of function signatures must remain unaltered.

Given the nature of this user-level library and the objective to simplify your learning experience, the course staff has thoughtfully integrated certain functions within the subsequent sections. These functions are designed as practical workarounds to streamline the implementation process and enhance your efficiency. It is imperative to diligently integrate the functionalities of these designated functions at specified points in your library to ensure the seamless realization of the required functionalities.

A stern reminder is warranted: any deviation from the API specification will result in deductions, underscoring the importance of meticulous adherence to the prescribed guidelines.

5.1 Threading

In this section, your task is to implement the foundational functionality of a threading library. You are required to create functions that adhere to the following API specifications.

5.1.1 `void init_lib ()`

This crucial function serves to initialize the library, encompassing the setup of any essential state-tracking variables. Given the library's nature, this function must be invoked before any other function in the library when running within the main program. It is a one-time call, mirroring the structure of a program that utilizes your thread library. Consider the following hypothetical structure:

```c
#include "thread.h"
int main ()
{
    init_lib();
    // Other code
    return 0;
}
```

This example illustrates the necessity of invoking `init_lib()` at the outset of the main program to ensure proper initialization.

5.1.2 `int thread_create (thread_t *thread, void (*start_routine))`

This integral function is tasked with creating a new thread within the library. It accepts a pointer to a `thread_t` structure and a pointer to the function that the newly created thread should execute. The function returns the ID of the thread it has just created.

An important note for clarity: in typical C programs, the entry point is the `main` function. However, in our threading library, there is no requirement to implement the `main` function as a thread. Since the `main` function is already executing, creating a thread with a NULL function as the callback is sufficient. This thread becomes the first thread in the program, and you need not manually save or restore the contents of this function.

Here's a snippet illustrating the suggested structure of the main function:

```c
#include "thread.h"
int main()
{
    init_lib();
    thread_t *main_thread;
    main_thread = malloc(sizeof(thread_t));
    thread_create(main_thread, NULL);
    // Other code
    return 0;
}
```

In a scenario where you wish to run a specific function in a thread, the process is straightforward:

```c
#include "thread.h"

void my_function()
{
    // Do something
}

int main()
{
    init_lib();
    thread_t *main_thread;
    main_thread = malloc(sizeof(thread_t));
    thread_create(main_thread, NULL);

    thread_t *my_thread;
    my_thread = malloc(sizeof(thread_t));
    thread_create(my_thread, &my_function);
    // Other code
    return 0;
}
```

This structure enables seamless creation of threads, whether for the main program or for specific functions, showcasing the versatility of the threading library.

5.1.3 void thread_exit ()

This essential function serves as the designated exit point when a thread decides to terminate. It operates akin to a return statement for threads, providing a streamlined mechanism for thread termination.

As part of the facilitated implementations, you can reliably assume that this function will be positioned at the conclusion of every thread function. Consequently, any function intended to run in a thread should adopt the following structure:

```c
void my_function()
{
    // Do something
    thread_exit();
}
```

This standardized structure ensures uniformity and simplifies the process of handling thread terminations within your threading library.

5.1.4 void thread_join (thread_t *thread)

This function is invoked when a thread desires to wait for another specific thread to complete its execution. The calling thread enters a blocked state until the identified thread concludes its execution. Following the completion of the target thread, the calling thread resumes execution from the point where `thread_join` was initially called.

```c
#include "thread.h"

void my_function()
{
    // Do something
}

int main()
{
    init_lib();
    thread_t *main_thread;
    main_thread = malloc(sizeof(thread_t));
    thread_create(main_thread, NULL);

    thread_t *my_thread;
    my_thread = malloc(sizeof(thread_t));
    thread_create(my_thread, &my_function);
    // Other code

    // start running the timer before you join threads
    timer_start();
    thread_join(my_thread);

    // stop the timer before exiting
    timer_stop();

    return 0;
}

```

5.1.5 void thread_sleep (unsigned int milliseconds)

This function introduces a pause in the execution of a thread for the specified duration. The thread remains uncheduled until the designated time has elapsed. It is essential to implement spin-waiting, tracking the elapsed time without yielding the CPU.The provided `get_time` function offers a means to obtain the current time in milliseconds, aiding in the implementation of spin-waiting.

5.1.6 void thread_yield (void)

Invoked when a thread voluntarily relinquishes its allocated CPU time, this function can be called at any point within a thread function—excluding after `thread_exit` has been called. Upon invocation, the thread is rescheduled, and it is crucial to note that it should not be considered terminated.

A visual representation of `thread_yield` in a thread function might resemble the following:

```c
void thread_function()
{
    // Do something
    thread_yield();
    // Do something else
    thread_yield();
    // Do something else
    thread_exit();
}
```

5.1.7 void context_switch (thread_t *old, thread_t *new)

This pivotal function orchestrates the transition between threads. It receives pointers to the old and new threads, saving the context of the former and restoring the context of the latter. Additionally, it updates the states of the involved threads. The implementation of this function is paramount and is accomplished using the `setjmp` and `longjmp` functions. Detailed insights into these functions can be gleaned from the `setjmp.h` documentation and the recommended Important Information [7] section of this document.

5.2 Control Primitives

In this section, your task is to implement the following essential functions for control primitives:

5.2.1 void mutex_init (mutex *m)

This function serves to initialize the mutex. Its invocation can be assumed to occur only once for each mutex.

5.2.2 void mutex_acquire (mutex *m)

Invoked to acquire the mutex, this function ensures that a thread gains exclusive access to the mutex.

5.2.3 void mutex_release (mutex *m)

This function facilitates the release of the mutex, allowing other threads waiting for it to potentially acquire access. Notably, when a thread releases a mutex and multiple threads are in a blocked state, only the first thread that was initially blocked should be unblocked.

5.2.4 void sem_init (semaphore *sem, int value)

This function initializes the semaphore with the specified value.

5.2.5 void sem_wait (semaphore *sem)

Invoked when a thread intends to acquire the semaphore, this function handles the necessary blocking if the semaphore cannot be immediately acquired.

5.2.6 void sem_post (semaphore *sem)

This function is called when a thread releases the semaphore. Similar to mutex release, if multiple threads are waiting, only the first one blocked should be unblocked.

Note: It is imperative to note that your implementation must include the blocking of threads when acquiring mutexes or semaphores. These functions are designed to handle the synchronization of threads effectively. When a thread attempts to acquire a mutex or semaphore that is already held by another thread, it should be blocked until the mutex or semaphore is released. The thread that releases the mutex or semaphore should unblock the first thread that was blocked when attempting to acquire it.

6 Problem Solving

Time to team up with your buddies, Mutex and Semaphore! They're your trusty sidekicks in tackling the challenges ahead. Keep it straightforward—no fancy algorithmic detours allowed on this adventure!

6.1 The Doctor's Waiting Room

You've landed a gig creating a top-notch application for the doctor at the Health and Wellness Center at LUMS. Now, this doctor isn't your average stethoscope wielder — she's a fervent gamer with a penchant for Candy Crush (Level 5000 and counting). Eager to game during downtime, she's handed you a prescription: a program that juggles her two passions - medicine and gaming.

Prescription Specifications:

- Patients are like surprise guests; they can show up anytime (denoted by a_i). However they can't just barge in; they have to wait their turn.
- Each patient takes a different amount of time to be treated (denoted by b_i), some take a quick chat, others a prolonged heart-to-heart.
- The doctor is a solo performer; only one patient on stage at a time. It's a one-on-one show, not a group therapy session.

Implementation Dose:

To pull this off, envision yourself as the chief scriptwriter for this medical sitcom. Use the provided threading library as your trusty prop and synchronize the actors (doctors and patients) for a seamless performance.

Testing Instructions:

Write your program in `doctor.c`. Your program will receive input from `stdin` and output to `stdout`.

Input:

- The first line contains a single integer n denoting the number of patients.
- The next n lines contain two space-separated integers a_i and b_i denoting the arrival and treatment time of the i^{th} patient.

Output:

- When a patient enters, output: `Patient` n `got in line` (where n is the patient number).
- When a patient starts seeing the doctor, output: `Patient` n `starting seeing the doctor` (where n is the patient number).
- When a patient finishes seeing the doctor, output: `Patient` n `finished seeing the doctor` (where n is the patient number).
- After all patients are done, print: `All patients have left`.

Be meticulous with spaces and spelling, as the test files will scrutinize your outputs against the golden standards.

6.2 TipTop Dry Cleaning Scheduler

So, TipTop Dry Cleaners has summoned you, the laundry maestro, to orchestrate a program managing customers and laundry machines in a university laundry room. Picture it: a laundry symphony with clothes instead of violins.

- The laundry room boasts n laundry machines — think of them as your trusty instruments.
- There are m customers itching to use these machines. They're not just clothes; they're the divas of the laundry world.
- Each customer takes their sweet time, marked by a mysterious variable t. Maybe it stands for 'time-to-sparkle.' Maybe it stands for 'time-to-dry.' Maybe it stands for 'time-to-leave.' Who knows? All we know is that it's a positive integer.

Operatic Program Requirements:

Craft a program that ensures laundry harmony:

- No two diva-customers should be caught using the same machine at the same time. We're avoiding laundry duets here. **No two customers should use the same machine at the same time**.
- No customer should twiddle their thumbs waiting for an available machine. We're not in the business of thumb twiddling. **Every customer should be able to use a machine as soon as it's free**.
- Every laundry machine should be kept busy. Idle machines are like actors on break — not a good look for the show. **No machine should be free if there's a customer waiting for their turn**.
- Customers are served in the order they arrive.

Implementation Overture:

Now, take center stage with the `laundry.c` file. Utilize your previously crafted threading library. Think of control mechanisms as the conductors of this laundry symphony, ensuring the perfect choreography.

Testing Instructions:

Your program will receive input from `stdin` and output to `stdout`.

Input:

- The first line contains two space-separated integers n and m, where n is the number of washing machines, and m is the number of customers.
- The next m lines contain one integer each, denoting the time required by the m^{th} customer.

Output:

- When a customer enters, output: `Customer` n `got in line` (where n is the customer number).
- When the customer starts using the machine, output: `Customer` n `starting washing their clothes` (where n is the customer number).
- When the customer finishes washing their clothes, output: `Customer` n `finished washing their clothes` (where n is the customer number).
- After all customers are done, print: `All customers have left`.

Be meticulous with spaces and spelling, as the test files will scrutinize your outputs against the golden standards.

7 Important Information

7.1 Time Slice

Let's break down the essential time control functions – your timekeeping allies:

- `get_time ()`: Retrieves the current time in milliseconds.
- `timer_start ()`: Activate a system timer in a separate thread, thanks to `signal.h`. This timer facilitates periodic function calls, enhancing your control over time.
- `timer_stop()`: Cease the system timer. Always remember to use this before your program takes its final bow to avoid lingering in the background.

Already Provided:

Rest easy; these functions are already in your arsenal, ready to be wielded for precise time management.

7.2 Dealing with Interrupts

Employing `signal.h` introduces interrupt concerns, as interrupts can disrupt any function, including critical sections like context switching. To tame the signal chaos, we've enlisted the `sigprocmask` function.

Signal Management

- `SIG_BLOCK:` Use this to temporarily block signals, crucial during sensitive operations like context switching.
- `SIG_UNBLOCK:` Release the signals back into the wild when it's safe to do so.

Note: Ensure you leverage the same time mask provided in `time_control.c` to maintain harmony among signals. Mastering this tool is key to navigating the nuances of interrupt management. **It is recommended that you implement these functions before you start working on the threading library.**

7.3 The Mangle Function

Let's demystify the mangle function - a crucial piece in threading libraries, and good news: we've got it ready for you!

Simplified Explanation:

The mangle function is like a secret decoder for threading magic. In a world where memory is kept under lock and key for security, mangle steps in. It helps us smoothly talk to computer registers, allowing us to navigate different parts of the program effectively. Think of it as the conductor orchestrating a symphony of actions, creating the illusion that things are happening all at once.

Already Provided:

And here's the good part - you don't need to conjure this magic yourself. We've already handed you the mangle function, straight from the glibc playbook. Simply put, it's your backstage pass for making threads work seamlessly.

Ensure you acquaint yourself with this function as it's a key player in your threading library adventure.

7.4 The `setjmp.h` Library

Now, let's unravel the mysteries of the setjmp library — your ticket to non-local jumps and seamless thread tracking.

Understanding `sigjmp_buf`:

- For thread state tracking, embrace the `sigjmp_buf` in your Thread Control Block (TCB). This struct holds the essential `__jmpbuf` and `__saved_mask` fields.
- Specifically, `__jmpbuf` stores addresses residing in each register, while `__saved_mask` cradles the thread's signal mask. Dive into the setjmp.h documentation for a deeper exploration of `sigjmp_buf`.

Thread Creation Symphony

When birthing a thread (excluding the main thread), the `sigsetjmp` function becomes your ally. Save the context using `mangle` to decode the function's address and stack pointer. Ensure the `sigjmp_buf` fields are in tune by emptying the saved context in `__saved_mask` via `sigemptyset`. A well-orchestrated algorithm for thread creation emerges:

1. Initialize the TCB.
2. Allocate the stack.
3. Set the stack pointer.
4. Set the instruction pointer.
5. If this is the first thread, don't initialize context since the thread already has one.
6. Else, initialize the context.
   - Make the stack pointer in the context point to the stack (use mangle).
   - Make the instruction pointer in the context point to the function (use mangle).
   - Empty the saved mask in the context.
7. Return thread id.

Thread Switching Ballet

When it's time to pirouette between threads, the `siglongjmp` function takes center stage. It accepts a `sigjmp_buf` variable and gracefully leaps to the stored context.

Switching Algorithm

```
1. Reset the timer.
2. ret_val <- sigsetjmp(current_thread->context, 1).
3. if (ret_val == 0):
       Switch to the next thread.
4. else:
       Return from the function.
```

Feel free to enhance the steps if needed; this dance is designed for flexibility.

7.5 Architecture

Now, let's delve into the architectural foundations of our threading library.

Target Architecture: `x86_64`

When crafting our masterpiece, the first question that arises is, 'Which architecture shall we adorn with our creation¿ The answer for this assignment is the venerable `x86_64` architecture. This architectural gem graces most modern computers, including the IST lab machines.

Docker Container Assistance

The provided Docker container is finely tuned for the `x86_64` architecture. To verify your system's allegiance, employ the `arch` command. A swift check ensures you're in sync with the architectural harmony.

Register Guidance

In our symphony of threads, registers play a vital role. To guide you through this orchestration, we present `jmpbuf-offsets.h`. This header file is a treasure trove of information, offering insights into the offsets of all available registers in the `x86_64` architecture.

Accessing Registers

Unlock the potential of registers by leveraging the offsets provided in `jmpbuf-offsets.h`. For instance, to command the mighty R13 register, follow this spell:

```
#include "jmpbuf-offsets.h"
sigjmp_buf env;
env->__jmpbuf[JB_R13] = 0x12345678;
```

Seamlessly traverse the registers using these offsets, crafting a masterpiece on the `x86_64` canvas.

7.6 The `LOG_OUT` function

We've got a special guest in the form of the `LOG_OUT` function. It's here to shine where `printf` might dim the lights due to library quirks. `LOG_OUT` is your go-to maestro; it can play with integers, strings, and characters.

You may use `LOG_OUT` as you would normally use `printf`. However, as long as you dont have to deal with annoying segfaults, you can use `printf` as well.

7.7 Variable Intialization

It is recommended that you call `init_lib` after you have intialized other variables that you might need in your program.

```
#include "thread.h"
int main ()
{
    int *a = malloc(sizeof(int));
    // other initializations
    init_lib();
    // Other code
    return 0;
}
```

This is also a limitation of the threading library we are writing - consider this the backstage rule for our threading library's grand show.

8 Setup

For our `x86_64` journey, a specialized Docker setup awaits. Let's navigate the process effortlessly.

8.1 Docker Container Creation:

To summon your Docker container with all the necessary enchantments, cast the following bash spell:

```
bash docker-create.sh
```

This script orchestrates the setup, ensuring that your container emerges with the required dependencies ready for the grand thread symphony.

8.2 Docker Container Creation:

Whenever you're ready to embark on your assignment odyssey, unfurl the sails with the following script:

```
bash docker-start.sh
```

This script hoists the anchor, launching your container and delivering you to a shell where you can weave your thread magic.

8.3 Troubleshooting on MacOS

If you run into problems while compiling the program on MacOS, you can try the following:

1. Navigate to `Settings > Features in Development`
2. Unveil the hidden power by checking the box next to `‘Use Rosetta for x86/amd64 emulation on Apple Silicon‘`.
3. Cast the spell of transformation by clicking on `‘Apply and Restart‘`.

May these mystical configurations align the stars in your favor and grant you a harmonious coding experience.

8.4 Testing

In order to test your assignment, you can run the following command:

```
make test
```

However, you may also run individual tests. The flags for the tests are as follows:

1. `-t` for the `thread` test.
2. `-m` for the `mutex` test.
3. `-s` for the `semaphore` test.
4. `-d` for the `doctor` problem test.
5. `-l` for the `laundry` problem test.

You can use these by running the following command:

```
# for the thread test
make test ARGS="-t"
```

9 Submission

Submit your assignment through `LMS` on the assignment tab. The submission should include a `zip` file containing the following folders:

1. `src`: This folder should contain all the source files for your implementation.
2. `include`: This folder should contain all the header files for your implementation.
3. `app`: This folder should contain `doctor.c` and `laundry.c` files.

Please don't include any other files, or directories, as they'd just be making our lives harder, and would be removed nonetheless. Also, please make sure to follow the specified directory structure and not just dump all the files directly in the archive.

Consolidate these folders into a zip file named `<rollnumber.zip>`. For instance, if your roll number is 24100022, the submitted zip file should be named as `24100022.zip`.

Note: This assignment has a strict deadline. No late or email submissions will be entertained.

Good Luck, and Happy Coding!