OS Exercise 1

Operating Systems Reichman University

21.4.2025

Submission Notes:

- **Submission Deadline:** Via Moodle by `12.5.2025`. **No late submissions will be accepted!**
- **Collaboration:** This assignment is to be done in pairs (i.e., two students per team).
- **Submission Format:** Submit a single ZIP file containing all the requested files.
  1. **Ensure that your submission also contains the header files (that contain the type definitions of the structs)**
  2. Your zip file should contain 6 directories, task1,task2,task3,task4,task5,task6 where each directory contains the solution for the given task.
- **README:** Include a `README.txt` file at the root of the ZIP. This file must contain the submitting students' information. Each line should follow this format:

  ```
  First name, Last name, Student ID
  ```

- **Compilation Environment:** The checker uses `gcc13` on Ubuntu 24.04 LTS with the C23 standard.
- **Compilation and Execution:** Ensure your code compiles without errors and runs as expected.
- **Testing:** Write your own unit tests to validate your implementation by creating a `testmain.c` file containing a `main` function that tests your functions. **Do not submit** `testmain.c`. Make sure your submitted files **do not contain** a `main` function unless stated otherwise!
- **Covered Material:** This assignment covers topics from Presentation 2 and Recitation 2.
- **Allowed Atomic Libraries:** You are allowed to use only the **stdatomic.h** library for implementing synchronization mechanisms.
- You can use the **sched_yield()** function (from the **sched.h** library) as was shown in the lecture.
- **Restrictions:**
  - **Do Not Use:** The **pthread** library or any native implementation libraries such as **semaphore** for your synchronization mechanisms unless stated otherwise. The only synchronization primitives allowed are the ones you have implemented or provided in class.
  - **Dynamic Memory Allocation:** Avoid using any dynamic memory allocation functions (e.g., `malloc`, `calloc`, `realloc`) unless stated otherwise.

Task 1

Implement a semaphore by using the Test-And-Set (TAS) spinlock presented in class (you can use the code provided). The semaphore should support basic wait (decrement) and signal (increment) operations. Implement the functions in the provided `tas_semaphore.h`.

Mandatory Functions:

```
void semaphore_init(semaphore* sem, int initial_value);
/*
 * Initializes the semaphore pointed to by 'sem' with the specified initial value.
 */

void semaphore_wait(semaphore* sem); // Decrement the semaphore value (wait
    operation)

void semaphore_signal(semaphore* sem); // Increment the semaphore value (signal
    operation)
```

Listing 1: Mandatory functions for Task 1

Submit your implementation in `tas_semaphore.c` and any additional files (if there are any).

Task 2

Implement a semaphore using the Ticket Lock mechanism **presented in class**. Implement the functions in the provided `tl_semaphore.h`.

Mandatory Functions:

```
void semaphore_init(semaphore* sem, int initial_value);
/*
 * Initializes the semaphore with an initial value.
 */

void semaphore_wait(semaphore* sem); // Decrement the semaphore value (wait
    operation)

void semaphore_signal(semaphore* sem); // Increment the semaphore value (signal
    operation)
```

Listing 2: Mandatory functions for Task 2

Ticket Lock From Lecture

```
typedef struct
{
    atomic_int ticket;
    atomic_int cur_ticket;
} ticket_lock;

void ticketlock_init(ticket_lock* lock)
{
    atomic_init(&lock->ticket, 0);
    atomic_init(&lock->cur_ticket, 0);
}

void ticketlock_acquire(ticket_lock* lock)
{
    // get my ticket
    int my_ticket = atomic_fetch_add(&lock->ticket, 1);

```

```
    // wait until it is my turn
    while (atomic_load(&lock->cur_ticket) != my_ticket)
    {
        sched_yield();
    }
}

void ticketlock_release(ticket_lock* lock)
{
    atomic_fetch_add(&lock->cur_ticket, 1);
}
```

Listing 3: Ticket Lock From Lecture

Submit your implementation in `tl_semaphore.c` and any additional files (if there are any).

Task 3

Implement a condition variable. This is a synchronization primitive that allows threads to wait until a particular condition is met. Implement the functions in the provided `cond_var.h`.

Mandatory Functions:

```
void condition_variable_init(condition_variable* cv);
/*
 * Initializes the condition variable pointed to by 'cv'.
 */

void condition_variable_wait(condition_variable* cv, ticket_lock* ext_lock);
/*
 * Causes the calling thread to wait on the condition variable 'cv'.
 * The thread should release the external lock 'ext_lock' while waiting and
    reacquire it before returning.
 */

void condition_variable_signal(condition_variable* cv);
/*
 * Wakes up one thread waiting on the condition variable 'cv'.
 */

void condition_variable_broadcast(condition_variable* cv);
/*
 * Wakes up all threads waiting on the condition variable 'cv'.
 */
```

Listing 4: Mandatory functions for Task 3

Submit your implementation in `cond_var.c` and any additional files (if there are any).

Task 4

Implement a read-write lock that allows multiple readers or a single writer in the mutual exclusion section. Implement the functions in the provided `rw_lock.h`.

Mandatory Functions:

```c
void rwlock_init(rwlock* lock);
/*
 * Initializes the read-write lock.
 */

void rwlock_acquire_read(rwlock* lock);
/*
 * Acquires the lock for reading.
 */

void rwlock_release_read(rwlock* lock);
/*
 * Releases the lock after reading.
 */

void rwlock_acquire_write(rwlock* lock);
/*
 * Acquires the lock for writing. This operation should ensure exclusive access.
 */

void rwlock_release_write(rwlock* lock);
/*
 * Releases the lock after writing.
 */
```

Listing 5: Mandatory functions for Task 4

Hints:

- **Consider fairness and prevent writer starvation.**
- **You may use your previously implemented synchronization primitives.**

Submit your implementation in `rw_lock.c` and any additional files (if there are any).

Task 5

Objective: Implement a thread-local storage (TLS) mechanism using a statically allocated array. This TLS must reside in the data segment (i.e., no dynamic memory allocation) and you must **not** use C11 or compiler-specific thread-local keywords (e.g., `__thread` or `_Thread_local`).

Important: The TLS resides within the global variable `g_tls` which is the array that holds the TLS entries. For simplicity we are limiting the size of the array, so you don't need to do any dynamic allocations.

Details:

- **Structure:** Define a structure `tls_data_t` as follows:

```c
#include <stdint.h>
#include <pthread.h>

#define MAX_THREADS 100

typedef struct {
    int64_t thread_id;  // For an unused entry, initialize to -1
    void*   data;       // Initialize to NULL
} tls_data_t;
```

- `g_tls` is an array of `tls_entry_t`, which holds the thread ID and a void* pointer that points to the data.
- Initialize `g_tls` so that all `thread_id` fields are set to -1 and all `data` pointers are NULL, so we will know all the entries are not usable by any thread.
- When initializing TLS for a thread:
  - Search `g_tls` for the first empty slot and set the thread ID (using `pthread_self()`).
  - If the thread ID already exists, simply return.
  - **If the TLS is full, print:**

    `thread [thread id] failed to initialize, not enough space`

    exit with return code 1 with `exit(1)`.
  - Search should have a runtime complexity of $O(n)$.
- When setting data (i.e. void*) to the TLS, find the entry corresponding to the calling thread and update its `data`.
- When retrieving data, find the relevant entry in the TLS and return the data.
- if the thread's entry is not found durting **get or set** operations, **print:**

  `thread [thread id] hasn't been initialized in the TLS`

  exit with return code 2 with exit(2).
- To free a TLS entry, reset the `thread_id` to -1 and `data` to NULL.
- **Important:** **Ensure** that the mapping and TLS updates are mutuality excluded using your own synchronization mechanisms.
- **Important:** If the data points to a variable on the stack, the pointer will become dangling if the variable goes out of scope. For your tests, you may also use `malloc` to provide a `void*` to data on the heap, but notice, once you free the data, the pointer in the TLS dangles. Both pointers to variables on the stack and on the heap must work. You cannot assume in the TLS anything about the `void*` data.

Mandatory Definitions and Functions:

```
void init_storage();
/*
 * Initializes g_tls as an array of MAX_THREADS entries.
 */

void tls_thread_alloc();
// Initializes the TLS entry for the calling thread.

void *get_tls_data(void);
/*
 * Returns a pointer to the arbitrary data stored for the calling thread.
 */

void set_tls_data(void* data);
/*
 * Sets the given void* data to the TLS for the calling thread.
 */

void tls_thread_free();
// Frees the TLS entry for the calling thread.
```

Listing 6: Mandatory Definitions and Functions for Task 5

Submit your implementation in `local_storage.c` and any additional files (if there are any).

Task 6

Objective: Implement a Producer-Consumer pattern program that checks if numbers are divisible by 6.

Detailed Requirements:

- **Producers:**
  - Generate random numbers in the range from a variable [lowest number generated] (initially 0) up to 1,000,000.
  - Continue generating until all numbers in the range have been produced (ensure the same number is never passed twice).
  - Print to stdout in the following format:

    `Producer {thread ID} generated number: {number}`
  - After printing, push the number to a shared queue.
- **Consumers:**
  - Pull numbers from the shared queue.
  - Check if the number is divisible by 6.
  - Print to stdout in the following format:

    `Consumer {thread ID} checked {number}. Is it divisible by 6? {result}`

    where `{result}` is either `True` or `False` (case-sensitive).
  - Ensure that printing is synchronized to avoid overlapping output.
- **Program Input:** The program should accept exactly 3 command-line arguments:
  1. Number of Producers.
  2. Number of Consumers.
  3. Seed for random number generation.
- **Configuration Printout:** Before starting the producers and consumers, print:

  Number of Consumers: {Number of Consumers}
  Number of Producers: {Number of Producers}
  Seed: {Seed}
- **Error Handling:** If the argument count or values are incorrect, print the usage message:

  `usage: cp_pattern [consumers] [producers] [seed]`

  and exit with return code 1.
- Use `srand()` with the provided seed and generate numbers using `rand() % x`.
- Once all numbers have been generated, a producer should signal a global condition variable.
- Provide a function to check if the consumer queue is empty.
- In the `main` function (which you **must submit** for this task), use the condition variable to wait until the producers have finished, then busy-wait until the consumers complete processing.
- After the consumers finish processing, shutdown all consumer and producer threads and exit with code 0.

- **Note:** Dynamic memory allocation is allowed in this task; **however, ensure that all allocated memory is properly freed**.
- After **all** the numbers were generated and pushed to the queue, the producers threads finish and return from their thread function.
- **Main Thread - General Flow**
  1. Starts the producers
  2. Starts the consumers
  3. Waits until the producers have produced all the numbers – `wait_until_producers_produced_all_numbers()`
  4. Waits until the consumers have consumed all the jobs the producers produced – `wait_consumers_queue_empty()`
  5. Stops the consumers "thread pool" cleanly – `stop_consumers()`
- **Note:** In class, we discussed the general producer/consumer pattern using two condition variables (is_full and is_work). In this exercise, you only need to use a single condition variable (e.g., is_empty) to signal when there is work in the queue. You can assume the queue is large enough that it will never become full, so you do not need to handle the is_full case.

Mandatory Functions:

```c
void start_consumers_producers(int consumers, int producers, int seed);
/*
 * Starts the consumers and producers, and passes the seed to the producers.
 */

void stop_consumers();
/*
 * Stops the consumer threads.
 */

void print_msg(const char* msg);
// Prints a message synchronously to avoid overlapping output.

void wait_until_producers_produced_all_numbers();
/*
 * Waits until all numbers between 0 and 1,000,000 have been produced.
 */

void wait_consumers_queue_empty();
/*
 * wait until queue is empty, if queue is already empty
 * return immediately without waiting.
 */
```

Listing 7: Mandatory functions for Task 6

Submit your implementation in `cp_pattern.c` and any additional files (if there are any).

Final Notes

- **Read the Instructions Carefully:** Ensure that you adhere to all restrictions and requirements.
- **Testing:** Test your code thoroughly with various inputs to ensure correctness, especially under concurrent execution.

  - **Also test without a debugger** since debuggers may affect synchronization behavior and hide bugs.
  - **Debugging:** Start with a small number of threads (e.g., one, then two, then five) and increase gradually.
  - Use debug printouts to help identify issues, but **make sure to remove or comment out** any debug messages before submission as they may affect grading.

- **Code Quality:** Write clean, well-commented code to help the graders understand your implementation. Comments will also help you understand what you are trying to achieve.
- **Unit Tests:** You are (more than) encouraged to use unit tests to validate your synchronization mechanisms.
- **Grading Reminder:** A happy grader is a merciful grader!

Good Luck!