UNIVERSITY OF MALTA

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY

Department of Computer Science

CPS1012 (Operating Systems and Systems Programming I)

Assignment: TID Manager

Instructions:

1. This is an **individual** assignment; the implementation carries **40**% of the final CPS1012 grade.
2. The report and all related files (including code) must be uploaded to the VLE by the indicated deadline. Before uploading, archive all files into a compressed format such as ZIP. It is your responsibility to ensure that the uploaded file and all contents are valid.
3. Everything you submit must be your original work. All source code will be run through plagiarism detection software. Please read carefully the plagiarism guidelines on the ICT website.
4. Reports (and code) that are difficult to follow due to poor writing style, organisation or presentation will be penalised.
5. Always test function return values for errors; report errors to the standard error stream.
6. Each individual may be asked to discuss their implementation with the examiners, at which time the program will be executed and the design explained. The outcome may affect the final marking.

Introduction

In Linux, each thread within a process is assigned a unique identifier known as a Thread ID (TID). The TID serves as a way for the kernel to distinguish between individual threads within the same process. When a new thread is created, the kernel allocates a TID for the thread. This TID is unique within the context of the process to which the thread belongs, allowing the kernel to manage and schedule threads efficiently. Threads can access their TIDs internally, enabling them to perform various operations based on their TID. TIDs are scoped within the context of each process, ensuring that threads within different processes can have the same TID without conflict. Within a single process, each thread must have a distinct TID. The goal of this assignment is to implement a process-level TID manager, similar to an operating system's Process ID (PID) manager that is responsible for managing process identifiers.

This assignment will be evaluated on a number of criteria, including correctness, structure, style and documentation. Therefore, your code should do what it purports to do, be organised into functions and modules, be well-indented, well-commented and descriptive (no cryptic variable and function names), and include a design document describing your solution.

Deliverables

Upload your source code (C/C++ files together with any accompanying headers), including any unit tests and additional utilities, through VLE by the specified deadline. Include makefiles with your submission which can compile your system. Make sure that your code is properly commented and easily readable. Be careful with naming conventions and visual formatting. Every system call output should be validated for errors and appropriate error messages should be reported. Include a report describing:

- the design of your system, including any structure, behaviour and interaction diagrams which might help;
- any implemented mechanisms which you think deserve special mention;
- your approach to testing the system, listing test cases and results;
- a list of bugs and issues of your implementation.

Avoid incorporating a complete code listing in your report; however, you may include code snippets as needed.

Development Environment

Repl.it is an online development environment that supports various programming languages, including C/C++. It also provide a Linux container that you can use to develop and run your applications. To use repl.it for this assignment:

- Go to the repl.it website and sign in or create a new account. Use you University of Malta email.
- Click on the ”Create Repl” button to create a new Repl.
- Select ”C” or ”C++” as the language for your Repl.
- Develop your assignment code in the main file, adding additional files if required.
- Use the built-in shell in repl.it to compile and run your code.
- Test your code thoroughly.

Task 1

*This task concerns the implementation of the TID manager's Application Programming Interface (API) and carries **25%** of the total marks.*

The TID manager is responsible for assigning unique thread IDs within a process. When a new thread is created, it will invoke an API call that will return a unique TID, and will call a second API call to relinquish the TID before exiting, such that the TID can be reassigned.

Problems

(a) Use the following constants to identify the range of possible TID values:

```
#define MIN_TID 300
#define MAX_TID 5000
```

Use any data structure of your choice to represent the availability of thread identifiers.

Hint There are several strategies you can employ here, including using bitmaps, array of structures, linked lists, and so on.

[10 marks]

(b) Implement the following API function call for creating and initializing a data structure representing TIDs. It should return 1 if successful and -1 if unsuccessful.

```
int allocate_map(void);
```

[5 marks]

(c) Implement the following API function for allocating and returning a TID. It should return -1 if it is unable to allocate a TID (all TIDs are in use).

```
int allocate_tid();
```

[5 marks]

(d) Implement the following API function for releasing a TID, called by a thread when before it exits.

```
void release_tid(int tid);
```

[5 marks]

Task 2

This task concerns the development of a multi-threaded program that uses the API defined in Task 1 and carries ***30%*** *of the total marks.*

This task involves the implementation of a multithreaded program that creates a number of threads, where each thread will request a TID, sleep for a random period of time, and then release the TID. Sleeping for a random period of time approximates typical TID usage in which a TID is assigned to a new thread, the thread executes and then terminates, and the TID is released on the thread's termination. You will be using the pthread library for creating and managing threads. An overview of how to use this library is provided below.

To use the pthread functions and data types, you need to include the pthread.h header file in your C program

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

When compiling your program, you need to link it with the pthread library:

```
gcc program.c -o program -pthread
```

You can create a new thread using the pthread_create function. It takes four arguments:

1. A pointer to a pthread_t variable where the thread ID (provided by the kernel) will be stored
2. Thread attributes (usually set to NULL for default attributes)
3. A pointer to the function the thread will execute
4. Optional arguments to pass the function

```
void *thread_function(void *arg) {
    // Thread code goes here
    return NULL;
}

pthread_t kernel_thread_id;
pthread_create(&kernel_thread_id, NULL, thread_function, NULL)
```

To wait for a thread to finish its execution, you can use the pthread_join function. It takes two arguments:

1. The kernel provided thread ID of the thread to join
2. A pointer to a variable where the return value of the thread function will be stored

```
void *result
pthread_join(thread_id, &result);
```

To exit from a thread, you can simply return from the thread function or call pthread_exit explicitly

```
void *thread_function(void *arg) {
    // Thread code goes here
    pthread_exit(NULL);
}
```

Problems

(a) Write a simple C program that supports command-line arguments to specify:

- The number of threads the program will create
- The minimum and maximum random sleep time for the threads

A summary of the command line arguments is given in the table below:

Switch	Description	Default
-n	number of threads to generate	100
-l	minimum (low) random sleep time	1
-h	maximum (high) random sleep time	10

[5 marks]

(b) Extend your program so that it calls the allocate_map function defined in Task 1 to initialize the TID manager data structures.

[5 marks]

(c) Create the thread function that will sleep for a random number of seconds within the range specified through the command-line arguments. You can use the sleep function, which is passed an integer value representing the number of seconds to sleep. To use this function you will need to include the unistd.h header file.

[5 marks]

(d) Extend the thread function to use the TID manager API implemented in Task 1. The first thing that the thread function does should be to call the allocate_tid function, whilst just before returning, or exiting, it should call the release_tid function.

[5 marks]

(e) Extend the C program so that, after initializing the tid manager, creates the number of threads requested through the command-line flag (using pthread_create) and subsequently wait for all the threads to finish (using pthread_join).

[10 marks]

Task 3

*This task involves adding synchronization between threads and carries **25%** of the total marks*

In this task, the solution from the previous task will be modified to ensure that the data structures used to represent the availability of thread identifiers, defined in Task 1, are free from race conditions. You will be using the mutex functionality of the pthread library for synchronization. An overview of how to use these functions is provided below.

A mutex variable needs to declared and initialized before it can be used. This can either be performed through the default initialization macro:

```
PTHREAD_MUTEX_INITIALIZER
```

or by dynamically initializing the mutex using the pthread_mutex_init function

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

To protect a critical section of code, you need to lock the mutex before accessing the shared resource and unlock it when done:

```
pthread_mutex_lock(&mutex);
// Critical section
pthread_mutex_unlock(&mutex);
```

After you're done with the mutex, you should destroy it to release system resources:

```
pthread_mutex_destory(&mutex);
```

Here's a complete example that create two threads that update a shared variable, with access to the variable synchronized through mutex locks.

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_variable = 0;

void *thread_func(void *arg) {
    pthread_mutex_lock(&mutex);
    shared_variable++;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    pthread_t thread_id1, thread_id2;

    pthread_create(&thread_id1, NULL, thread_func, NULL);
    pthread_create(&thread_id2, NULL, thread_func, NULL);

    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

The pthread_mutex_trylock() function is used to attempt to acquire a lock without blocking. It tries to acquire the mutex lock, and if it is currently locked by another thread, the function returns immediately.

```
if (pthread_mutex_trylock(&mutex) == 0) {
    // Mutex lock acquired
    // Critical section
    // Release the mutex when done
    pthread_mutex_unlock(&mutex);
} else {
    // Mutex is locked by another thread
    // Handle the case where the lock cannot be acquired
}
```

Problems

(a) Use mutex locks to synchronize access to the TID manager data structures implemented in Task 1. Note that even though it is tempting to simply acquire a lock at the beginning of the function and release the lock at the end, efficiency and performance will be improved if you manage to keep the number of statements executed in the critical section as low as possible.

[10 marks]

(b) If there are multiple threads waiting for a mutex lock to become available, the next thread that will be given the lock once it is released is determined by the OS scheduler. A specific ordering can be enforced by using the non-blocking mutex acquisition function pthread_mutex_trylock and using a backoff scheme based on a certain criterion. A backoff scheme is a strategy to handle situations where contention for a shared resource is high. The idea is to temporarily reduce or delay further attempts to access the resource to alleviate contention.

Use the pthread_mutex_trylock function to acquire mutex locks, and if they are currently locked by another thread, apply a backoff scheme based on the thread ID provided by the kernel. You can get this thread ID using the pthread_self function as follows:

```
pthread_t kernel_thread_id = pthread_self();
```

Apply a sleep in microseconds based on the value of the thread ID. You can do so by using the usleep function, which takes an integer number of microseconds to sleep for.

[15 marks]

Task 4

This task concerns the deliverables, and carries ***20%*** *of the total marks*

Deliverables

(a) Generate a Makefile for you implementation to facilitate easy compilation. Use the Makefile template, which assumes that all your source and header files are in the same SRC_DIR directory:

```
CC = g++
CFLAGS = -Wall -Wextra -pthread

TARGET = tid_manager
SRC_DIR = [Your directory]


SRC_FILES = $(wildcard $(SRC_DIR)/*.c) $(wildcard $(SRC_DIR)/*.cpp)
INC_FILES = $(wildcard $(SRC_DIR)/*.h)

all: $(TARGET)

$(TARGET): $(SRC_FILES) $(INC_FILES)
  $(CC) $(CFLAGS) -I$(INC_FILES) $(SRC_FILES)           -o $(TARGET)

clean:
  rm -f $(TARGET)
```

[5 marks]

(b) Write the report containing the content defined in the Deliverables section.

[15 marks]
