Machine Problem 3: Simple Thread Scheduling

Total Machine Problem: 40 points
This Handout is for Step 1 of MP3

Introduction

In this machine problem, you are to develop a simple thread system for mixed applications. We will approach this in several steps, starting from a simple threading library (everything will be based on POSIX threads) in C++, continuing to schedulable threads, and on to more sophisticated scheduling with guaranteed performance, such as needed for multimedia applications.

We very loosely mimic a simplified version of the thread paradigm in Java and the scheduling paradigm in Real-Time Specification for Java (`www.rtsj.org`), which is one of the industry standards for real-time java platforms.

Threads will be implemented using the class `Task`, which is similar in spirit to the Java `Thread` class.

```
class Task {
protected:
  /* -- NAME */
  static const int   MAX_NAME_LEN = 15;
  char               name[MAX_NAME_LEN];

  /* -- IMPLEMENTATION */
  pthread_t thread_id;
  /* If you implement tasks using pthread, you may need to store
     the thread_id of the thread associated with this task.
  */

public:
  /* -- CONSTRUCTOR/DESTRUCTOR */
  Task(const char _name[]) {
    /* Create, initialize the new task. The task is started
       with a separate invocation of the Start() method. */
    strncpy(name, _name, MAX_NAME_LEN);
  }
  ~Task();
  /* -- ACCESSORS */
  char * Name();
  /* Return the name of the task. */

  /* -- TASK LAUNCH */
  virtual void Start();
  /* This method is used to start the thread. For basic tasks
  implemented using pthreads, this is where the thread is
  created and started. For schedulable tasks (derived from
  class Task) this is where the thread is created and handed
  over to the scheduler for execution. The functionality of
  the task is defined in "Run()"; see below. This method is
  called in the constructor of Task.
  */

  /* -- TASK FUNCTIONALITY */
  virtual void Run() = 0;
  /* The method that is executed when the task object is
  started. When the method returns, the thread can be
  terminated. The method returns 0 if no error. */

  /* -- MAIN THREAD TERMINATION */
  static void GracefullyExitMainThread();
  /* This function is called at the end of the main() function.
     Depending on the particular thread implementation, we have
     to make sure that the main thread (i.e., the thread that
     runs executes the main function) either waits until all
     other threads are done or exits in a way that does not
     terminate them.
  */
};
```

In a first step, you will implement tasks that are basically pthreads, and which are scheduled using the pthread scheduler. In the following steps, you will take over the scheduling yourself, by implementing the so-called *schedulable* tasks in form of the class `Schedulable`, which is derived from class `Task`. In order to schedule tasks, you will also have to implement the class `Scheduler`, which is derived from class `Task` as well.

(Note: It is unlikely that a real system would be implemented in this way, as the overhead is significant. In particular, the scheduler is a separate thread, which causes a large number of context switches during run-time.)

Step 1:

Implement tasks using `pthreads`. The application should be able to use threads in a very simple fashion, along the following lines.

```
#include <Task.H>

class FlutterController : public Task {
  FlutterController(char _name[]) : Task(_name) {}
  virtual void Run() {
     /* This is the flutter control code. */
  }
};
class AvionicsTask : public Task {
  AvionicsTask(char _name[]) : Task(_name) {}
  virtual void Run() {
     /* This is the avionics code. */
  }
};
int main(int argc, char * argv[]) {
  FlutterController("flutter control");
  AvionicsTask task2("avionics task");
  Task::GracefullyExitMainThread();
}
```

Passing Class Methods as Function Parameters: If you implement Step 1 using PThreads, you face the problem that at some point you have to pass the method `Run()` as a function parameter during process creation. You will see that directly passing the method will not work.

A solution to this may look as follows (uses a so-called *thunk* function):

```
void * tfunc(void * args) {
  Task * task_instance = (Task *)args;
  task_instance->Run();
  return NULL;
}
```

This function is then used during thread creation, and the pointer to the Task object is passed as argument to the function.

Compiling Programs that use pthreads: In order to make the pthread functions available in your program, you have to link in the pthread library when you compile your code. This is done using the `-l` option when you link the code. The compilation command would then look similar to the following:

```
g++ -o task_test1 task_test_step1.C task.o -lpthread
```

The Assignment (Step 1)

You are to implement a thread-based Task system with the use of PThreads. The system shall be implemented in C++ and hide the details of thread management behind the interface defined by the class `Task` described above. Your implementation should be able to correctly execute programs of the type described above in a multithreaded fashion.

What to Hand In (Step 1)

- You are to hand in two files, with names `task.H` and `task.C`, which implement your tasking system. A copy of file `task.H` as listed in this handout is made available on the web. This header file should not change significantly in your implementation. If you need to change it, give a compelling reason for the change in the modified source code.
- Test your implementation with the provided test program, called `task_test_step1`, whose source code is given on the web.

[STAY TUNED FOR STEP 2 and STEP 3!]