OS Api

Søren Hansen <sha@ase.au.dk>
V2.5

Introduction

In this exercise you will get the skeleton for an OO OS API, and it will be your job to implement the missing pieces based on the knowledge you have acquired. This will enable you to compile your own OO OS Api library for use in your applications/project.

Prerequisites

You must have:

- Knowledge about mutex, conditional and semaphore implementations.
- Know how state/sequence diagrams can be implemented when using the paradigm *Event Driven Programming* and thus sending messages between threads.
- Understand makefiles
- How you link static libraries to your program when using `g++`. This includes parsing the compiler the proper options for handling paths for the library's associated header files as well as where the static library itself is placed in a dir.
- Understand what contribution the *pimpl/cheshire cat* idiom provides.

Absolute requirements (Must haves for approval)

- The special `while` loop that must be in every thread function.
- Design consideration as well as diagrams.
- The 2 questions in exercise 4.

In the exercise folder, where you found this document you will find the *OSApiStudent.zip* file where *most* of the source code for the OS Api can be found. Remember to refresh you knowledge of the OS Api in general. Read *"Specification of an OS Api.pdf"* (available on CampusNet).

Exercise 1 Getting to know the OO OS Api

In this first exercise you to acquaint yourselves with the OO OS Api library.

There are numerous files thus getting an in-depth understanding on how they are connected helps completing the library in the next exercise.

Make sure that you have read *"Specification of an OS API"* so you know what the different parts do.

Places to start reading/inspecting - do note that it might be a good idea to go back and forth between the various files:

- File `example/main.cpp` - also discussed in the presentation
  Read the other files in the `example` directory
- File `osapi/Semaphore.hpp` - read it thoroughly
- File `Makefile` + `compiler_setup.host` & `compiler_setup.target`
- File `linux/Semaphore.cpp` in respect to implementation as well how it is compiled

Questions to answer:

- Inspecting the *OSApi* library, describe the chosen setup and how it works.
  Hint: The chosen: Directory structure, define usage, platform handling (e.g. windows, linux etc.) etc.
- When using the POSIX thread API one must provide a free C function as the thread function. This means that some *glue* is inserted such that it is possible to have a method of a class as the thread function (`run()`). Describe which classes are involved, their responsibilities and how they interact. A good approach could be show a call stack or a sequence diagram.
- In the windows implementation the so-called *pimpl/cheshire cat* idiom is used. Explain how it is used and what is achieved in this particular situation.
  See for instance the files related to the `Semaphore` class.
- Why is the `class Conditional` a friend to `class Mutex`? What does it mean to be a friend?
  Hint: See the interface for `class Mutex` as coded for the windows platform.

Exercise 2 Completing the Linux skeleton of the OO OS Api

In this exercise you will be tasked with completing the *OSApi* library. The following files are missing or have not been completed.

Read the whole exercise text before you commence!!!

- `inc/osapi/linux/Mutex.hpp` - *Missing*
- `linux/Mutex.cpp` - *Missing*
- `linux/Conditional.cpp` - *Already started*
- `linux/Utility.cpp` - *Missing*
- `linux/Thread.cpp` - *Already started*
- `linux/ThreadFunctor.cpp` - *Already started*

Use the Linux documentation either via `man` pages, *Kerrisk*, the net and remember to inspect the Windows-edition of the OS API. As stated above, the text *"Specification of an OS API"* is also valuable when you are to complete the Linux-edition of the OO OS API. However do note that the windows implementation is somewhat more complex in that it uses the *pimpl/cheshire cat* idiom. This is especially evident when you compare the linux implementation for `class Semaphore` with that of windows. The linux implementation has just has one member variable `semId_` of type `sem_t`.

It is of course very important that the interface to the OS resources is exactly as specified in the *OS API specification*. Otherwise, it will be impossible to interchange the OS APIs as seen from the user when he/she tries to compile it for another platform.

To compile your revised version of the OSApi, use the following command:

```
make DEBUG=1 TARGET=host all
```

The `makefile` has multiple different combinatorial uses, which means that you should take the time to study it and determine what it can and what it can't. It is a reasonable implementation with some interesting approaches that you might benefit by, however "feel free to improve".

Having completed part or all of the *OSApi* library, some appropriate tests are needed. This means that you have to create simple tests that verify that the library does indeed work. *Be sure to make individual tests before using the combined OSApi.* Otherwise determining actual errors may be much more problematic.

To help verify your program thread-wise use the source files found in the `example` and `test` path, however do note that the `makefiles` found here have not in any way been completed. Therefore *do not* assume that simple changes may be enough...

Exercise 3 On target

Verify that your OSApi library can compile for both Linux host and target. The same applies to your test applications.

Questions to answer:

- Did you do need to change code for this to work, if so what?

Exercise 4 PLCS now the OS Api

At this point we have created the PLCS system and it works great with Message and Message queues :-).

The next natural step would obviously be to port your PLCS application such that it now it uses your newly created *OS Api*.

Before you start coding you must make a class diagram as well as a sequence state/event diagram. Elaborate on your chosen design. Diagrams *and* design considerations must be part of your handin.

Again remember that the loop in the thread method `run()` ***must*** resemble the below code snippet in Listing 4.1.

Listing 4.1: Loop in thread function

```
1  void MyThread::run()
2  {
3    /* Whatever code you want */
4    while(running_)
5    {
6      unsigned long id;
7      osapi::Message* msg = mq_.receive(id);
8      handleMsg(msg, id);
9      delete msg;
10   }
11   /* Whatever code you want */
12 }
```

What is important here is that the `receive()` function must be in this loop and NOT anywhere else!

These questions must be answered, either here or part of your design discussion above:

- Which changes did you make and why?
- Does this modification whereby you utilize the *OSApi* library improve your program or not. Substantiate you answer with reasoning.