Matt Broadstone

Object Oriented Threads

Abstract:

This is a tutorial meant to walk you through integrating pthreads with an object oriented design. Specifically, we will be looking a the use of pthreads in creating an ISR (interrupt service routine) in QNX.

Motivation:

Threading simply has not been implemented with object oriented design in mind. Threads are created and dispatched in code blocks and are provided with functor callbacks to perform actions. This promotes the use of global variables, floating code segments, and overall poor design.

General Form:

Traditionally, a creation and dispatch of a thread looks like this:

```
void * callback_function(void *object)
{
      // Execute some code
}

int main(int argc, char **argv) {
      ...
      int threadId;
      threadId = ThreadCreate(0, &callback_function, NULL, NULL);
      ...
}
```

ThreadCreate is a function that creates and dispatches a thread. It's arguments are as follows:

```
#include <sys/neutrino.h>

int ThreadCreate(
          pid_t pid,
          void* (func)( void* ),
          void* arg,
          const struct _thread_attr* attr );
```

pid
The ID of the process that you want to create the thread in, or 0 to create the thread in the current process.
func
A pointer to the function that you want the thread to execute. The arg argument that you pass to ThreadCreate() is passed to func() as its sole argument. If func() returns, it returns to the address defined in the exitfunc member of attr.
arg
A pointer to any data that you want to pass to func.
attr
A pointer to a _thread_attr structure that specifies the attributes for the new thread, or NULL if you want to use the default attributes.

It may occur to you that a simply solution to make this object oriented would be to simply pass a method of a class to ThreadCreate, and you would be half right. Unfortunately, ThreadCreate (and subsequently pthreads) only accepts static methods, so something like this will not work:

```
class SomeClass
{
public:
      SomeClass() {
            ThreadCreate(0, &SomeClass::someMethod, NULL, NULL);
      }

private:
      void *someMethod(void *) { ... }
};
```

Fortunately we can just define someMethod as a static method and it will work! Not quite yet...

Generally in a callback_method, we will want to modify some variable (say a Boolean flag that changes according to the threads observation of another class). Well, this won't be possible in your static method unless the variable is also defined as static:

```
class SomeClass
{
public:
	SomeClass() {
		ThreadCreate(0, &SomeClass::someMethod, NULL, NULL);
	}

private:
	static int someVariable;

	static void *someMethod(void *) {
		while (!done) {
			if (somethingChanged)
				someVariable++;
		}
	}
};
```

This works just fine, however, we may want to have this thread work on a number of different instances of SomeClass, and we want to maintain multiple instances of someVariable for each class. Instead of making static variables for each possible instance of the class, we have another solution.

ThreadCreate not only allows you to pass a callback function, but a data argument to that function as well. So instead of having the method modify the static member, we can actually pass in the exact class the thread is associated with. When passing in the class, we simply need to dynamically cast the void* argument to check if its of this class type, and then modify accordingly. Our code now looks like:

```
class SomeClass
{
public:
	SomeClass() {
		ThreadCreate(0, &SomeClass::someMethod, NULL, NULL);
	}

private:
	int someVariable;

	static void *someMethod(void *object) {
		SomeClass *classInstance =
			dynamic_cast<SomeClass*>(object);

		if (!classInstance)
			return NULL;

		while (!done) {
			if (somethingChanged)
				classInstance->someVariable++;
		}
	}
};
```

Because of RTTI (Run-Time Type Information), C++ will either be able to cast the void* argument to a SomeClass instance or not, if it is we go on our way working with it, otherwise just exit the callback. While the second part should never happen, we check just in case, it's good etiquette.