Introduction

The concept of threading is quite simple to understand. Yet, few experienced engineers can tackle the implementation of a multi-threaded program. No one wants to deal with the nightmare of maintaining a multi-threaded program. Maintaining a multi-threaded application on multiple platforms is essentially formidable.

Some libraries help in porting a multi-threaded program to several platforms. Some languages provide a solution equivalent to the libraries by being available on several platforms. However, the real problem is the distance between the intuitive understanding of the concept and its linguistic representation.

The solution provided here reduces the implementation of threads to its conceptual understanding. The elimination of details also disposes of the costly and unpleasant maintenance for synchronization, critical section and other confusing notions, which by the way are indispensable at lower level of system programming.

Task is an object-oriented embodiment of the notion of thread. However, just as global functions are useful, so are global threads. These are discussed in the following sections, along with mechanisms for communication among global threads.

Tasks

The definition of a task is identical to that of class, except the keyword class is replaced with “task”. However, when an instance of a task is created, it is endowed with its own thread. Furthermore, calls to public methods of a task are queued, and callers are blocked. None of these is visible to the code using instances of tasks.

The following example illustrates the simplicity of using tasks.

```
task MyTaskType
    //private member and methods
public:
    //constructors and other methods

    void setAge(string, int);
    int getAge(string);

end; //end of definition

MyTaskType myTask; // myTask begins on its own thread

// The following call blocks this thread, and puts the call into
// the queue of myTask until myTask gets its execution slice

myTask.setAge(“Joe”, 47);

// The same thing happens here with regard to the call getAge().

output << “The age is: “ << myTask.getAge(“Joe”) << ‘\n’;
```

Tasks can participate in multiple-inheritance or be members of classes and tasks. Such combinations create multi-threaded applications free of thread related maintenance.

Signaling

Global threads communicate via Z++ signaling mechanism. We digress briefly to discuss signaling as used in context of threads.

Signals are of enumeration type. In Z++ one can extend enumeration types to contain more values. The base type for signals in the enumeration `signalEventType` defined in the system header file `exception.h`.

Suppose we wish to create a new signal, `_SIGNAL_TerminateYourself`. This is done as follows, by extending `signalEventType` to `MySignals`.

```
enum MySignals : signalEventType {_SIGNAL_TerminateYourself};
```

We can now generate the new signal and the system will deliver it to any thread that asks for it. The term “signal” identifies an object within Z++ that can accept and retrieve signals the same way input and output is done. The following statement generates the signal we just defined.

```
signal <- _SIGNAL_TerminateYourself;
```

The following statement returns a boolean. If the signal has been generated, it will be removed and the result will be true. Otherwise the result will be false.

```
signal ? _SIGNAL_TerminateYourself;
```

The usual usage would be in a loop or an if-else construct. For instance, the following loop will exit only when the signal is generated, presumably by another thread. Any loop, including a for-loop can be used instead of the do-loop shown here.

```
do enddo(signal ? _SIGNAL_TerminateYourself);
```

Signals can be used for all types of inter-thread asynchronous communication.

Global Threads

Any global function with a void return is a candidate for a global thread. One simply specifies the function to be a thread, as illustrated below.

```
void MyGlobalThread(double)<thread>;
```

Now, whenever this function is called, a thread is created and the function is executed in the new thread. The caller does not block, either. Note that the return is void or the compiler would not let you specify the function as a thread.

We can now answer the question of how to terminate a global thread. The do-loop in the previous section is the key. You can use it in many different ways, but let us assume that you put that loop at end of function body. Then, the thread will remain in that loop until the caller generates the terminate signal as illustrated in previous section. The thread is terminated and cleaned up when execution reaches the end of body of a global thread. That is where the function would have normally returned to its caller.

The termination of a global thread is the only implementation detail that an engineer must be aware of. However, the signaling mechanism provides a simple and yet sophisticated solution for multi-threaded applications.

Conclusion

Using a switch statement within a loop, a global thread can continually check for signals of interest and perform actions accordingly. Any number of threads can generate all kinds of signals that can be trapped by any number of threads to perform specific tasks.

Any combination of global threads and task threads can be used for creating abstract designs, without having to deal with synchronization details. Task threads are ideal abstractions for managing resources without having to deal with semaphores.

The automation of the notion of threading presented here factors out costly details without limiting an engineer’s ability to create multi-threaded applications. As a matter of fact, it increases an engineer’s throughput and reduces the cost of maintenance in direct proportion to the complexity of the application.