BlockingCallsWithActiveObjects

How to make calls to blocking Open C functions from an Active Object

It is often necessary to mix Open C with Active Objects. In particular, user interfaces are normally done with Active Objects. But if you want to call a blocking Open C function, such as the network communications functions, then you are faced with a problem. The blocking call will freeze your user interface until the call returns. If there is a non-blocking version of the call you *could* use a CTimer to repeatadly poll the function. But it is not battery efficient to have the periodic CPU activity involved in polling.

So you must turn to threads.

Okay.... here's pseudo code for a wrapper class around blocking calls named do_blocking_read_call and do_blocking_write_call. CMyCaller could be declared as a subclass of CBase.

The general idea is that the blocking calls are made from a new thread. The thread initializes then goes into a RSemapore::Wait. The main thread sets a command variable iCallerState and then calls RSemaphore::Signal to trigger the thread to execute the command. The two versions of RThread::Rendezvous are used together to coordinate with an Active Object.

After each Signal() the thread returns to the Wait state until the next Signal is given. A single thread with a repeating loop is used instead of creating and deleting a new thread each time.

```
class CMyCaller : public CBase
{
  public:
   // usual NewL construction stuff omitted
   void ConstructL();
   void DoCallAsynch(TRequestStatus& aStatus)
  public:
   void CallerThreadFn();
  private:
   RSemaphore iRequestSignal;
   RThread iCallerThread;
   TInt iCallerState;
}
```

The ConstructL function launches the new thread. The thread initializes, uses Rendezvous(TInt) to release User::WaitForRequest, and then Waits for a Signal.

```
void CMyCaller::ConstructL()
 {
        TRequestStatus status;
 iRequestSignal.CreateLocal(0,EOwnerProcess); // 0=wait for first Signal()
 iCallerThread.Create(_L("FN_Caller"), caller_thread_fn, stackSize, minHeap, maxHeap, this, EOwnerProcess);

 iCallerState = ECallerInitializing;
 iCallerThread.Rendezvous(status); // request that status get signaled
 iCallerThread.Resume();
 User::WaitForRequest(status);
        // here you could check status for initialization failure.
 }
```

The thread function(s).

```
TInt caller_thread_fn(TAny *aAddrSelf)
 {
 CMyCaller* self = (CMyCaller *)aAddrSelf;
 self->CallerThreadFn();
 return 0;  // the docs don't say whether this value has any meaning
 }

void CMyCaller::CallerThreadFn()
 {
 // NOTE: This code runs in its own thread with a different heap.
 int ret;
 TBuf8<8> port;
 port.Num(iPort);
 ret = init_server();

 if (ret <0)
  {
  iCallerState = EInitializingFailed;
  RThread::Rendezvous(KErrFailed);
  return;
  }
 else {
  iCallerState = ECallerReader;
  RThread::Rendezvous(KErrNone);
 }

 FOREVER {
              int success;
  iCallerState = ECallerReady;
  iRequestSignal.Wait();

  switch (iCallerState) {
                  case ECallingReadFunction:
```


(C) Copyright Nokia 2013. All rights reserved.

```
        success = do_blocking_řead_call(iSharedBuffer);
                      break;
                  case ECallingWriteFunction:
        success = do_blocking_write_call(iSharedBuffer);
                      break;
                }

  if (success) RThread().Rendezvous (KErrNone);
  else RThread().Rendezvous (KErrGeneral);
   }
 }
```

Now add an asynchronous function to be called from your Active Object.

```
// Call from an active object
void CMyCaller::DoCallAsynch(TRequestStatus& aStatus)
{
    // Rendezvous(TRequestStatus) requests that the AO's RunL
    // get called later when the thread calls Rendezvous(TInt).
    iCallerThread.Rendezvous(aStatus);

    // Tell the thread function which operation do execute
    //
    iCallerState = ECallingReadFunction;

    // Release the thread from its Wait call.
    iRequestSignal.Signal();
}
```

Note that because CallerThreadFn runs in a different thread, it does not manipulate any Symbian objects from another thread. It also is important to use the version of RThread::Create that will create its own heap. Otherwise you may have multiple threads manipulating the same heap - since the heap is not thread safe it can become corrupted. Don't allocate a heap object and then pass its ownership to another thread because it may not be safe if it later gets deleted by a thread with a different heap.

In this pattern all data is passed by copying bytes into a shared buffer.

Note: It's not documented that every RThread must be given a unique name in the Create function.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.