Active Scheduler

In Symbian an Active Scheduler is responsible for scheduling active objects. As described here, active objects provide support for asynchronous processing. Using multiple threads for this purpose is discouraged (though not forbidden) in Symbian OS due to its resource-intensive nature in a resource-constrained environment.

It is worth noting that the Active Scheduler schedules active objects in a non-pre-emptive way so that one active object cannot interrupt another while that is running. This is in contrast with thread scheduling (even in Symbian OS), since the execution of one thread can be pre-empted at any given time by another thread.

It is possible to customize the Active Scheduler in order to better suit a program's needs. For example, `CBaActiveScheduler` directly derives from CActiveScheduler in order to add some new features in addition to fine-tuning some existing ones.

Typically, a Symbian OS thread has a single Active Scheduler. As mentioned above, **that** Active Scheduler is responsible for managing all active objects belonging to the thread. However, sometimes it's desirable to use more than one Active Scheduler - and it can be done by ***nesting*** Active Schedulers. Nesting in this context means that a new Active Scheduler instance takes over the role of the current one (which in turn might be another nested Active Scheduler, too) including the management of all active objects.

Why is it good to nest Active Schedulers? Due to the blocking nature of nesting. Sometimes we'd like the execution flow to stop for a while and wait until an external event occurs (example: show a dialog and wait for a keypress). We could use, e.g., `User::WaitForRequest()` for this purpose, but that blocks the entire thread. We'd like that this blocking happen so that the (rest of the) active objects remain active and responsive. And nesting Active Schedulers is the right tool for that: it blocks the current execution flow (and the *outer* Active Scheduler), whilst lets the active objects be responsive (managed by the *inner* Active Scheduler).

How to use the active scheduler

The CONE environment (CCoeEnv) provides schedulers, so you just need to call the `Add()` method of the active scheduler.

```
// Just adding the Active Object to scheduler
CActiveScheduler::Add(this);
```

For applications that do not use the CONE environment (exe's), you have to create active scheduler by your own, as follows.

Add following header.

```
#include <e32base.h>
```

Link against: `euser.lib`, so add the following line to the .mmp file.

```
LIBRARY       euser.lib
```

```
// Creating  Active Scheduler
CActiveScheduler* Scheduler = new ( ELeave ) CActiveScheduler;
CleanupStack::PushL( Scheduler );
//Installing Active Scheduler
CActiveScheduler::Install( Scheduler );
// Delete active scheduler
CleanupStack::PopAndDestroy(scheduler);
```

How to start a wait loop

`CActiveScheduler` provides two methods, start() and stop(), to start and stop a new wait loop under the control of the current active scheduler. But this API is deprecated for that specific functionality, so it is recommended to use CActiveSchedulerWait object instead of CActiveScheduler for start/stop wait loop.

How it works

This is just an approximation. Install, error handling, priorities are not covered, and the linked list (of active objects) is treated as an array "Actives"

```
class CBasicScheduler
{
public:
    static void Start();
    static void Stop();
protected:
    static TBool *LoopStatus=NULL; // (7)
    static RPointerArray<CActive> Actives;
};

void CBasicScheduler::Start()
{
    TBool *oldloopstatus=LoopStatus; // (7)
    TBool localloopstatus=ETrue; // (7)
```


(C) Copyright Nokia 2013. All rights reserved.

```
    LoopStatus=&localloopstatus; // (7)

    while(localloopstatus) // (6)
    {
        User::WaitForAnyRequest(); // (1)

        TBool die=ETrue; // (5)
        for(TInt i=0;i<Actives.Count();i++) // (2)
        {
            if(Actives[i]->iStatus!=KRequestPending && Actives[i]->iActive) // (2)
            {
                die=EFalse; // (5)
                Actives[i]->iActive=EFalse; // (3)
                TRAPD(err,Actives[i]->RunL()); // (3)
                break; // (4)
            }
        }
        if(die) // (5)
            User::Panic(_L("E32USER-CBase"),46); // (5)
    };

    LoopStatus=oldloopstatus; // (7)
}

void CBasicScheduler::Stop()
{
    *LoopStatus=EFalse; // (7)
}
```

Explanation

(1) User::WaitForAnyRequest waits for a request (any) to be completed in the given thread

(2) The loop searches for the active object which is waiting for something (iActive is true), and the something is completed (iStatus is anything else than KRequestPending)

(3) RunL gets invoked - prior to that iActive gets cleared, so RunL may freely re-schedule the given active object, if it is needed. "err" could be used for checking the error, invoking CActive::RunError or CActiveScheduler::Error as a last attempt

(4) Only one RunL gets executed at a time

(5) If no eligible active object was found, the famous stray signal panic is raised

(6) Otherwise the endless loop continues

(7) Handling of Start-Stop, and nested scheduler loops is probably done via some similar way

Conclusion

(2) The Active Scheduler is aware of the inherited CActive::iStatus only - if someone defines an own TRequestStatus member variable (either under the name iStatus or anything else), it will not be checked by the scheduler

(2) SetActive is important too - if it is forgotten, the scheduler will not find which RunL it should invoke

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.