symbian
Academy

Active Objects

Active Objects

Active objects are used for event-driven multitasking

- They are a fundamental part of Symbian OS

This lecture explains why they are so important

- It explains how they are designed for responsive and efficient event handling

Active Objects

Event-Driven Multitasking on Symbian OS

- Demonstrate an understanding of the difference between synchronous and asynchronous requests and be able to differentiate between typical examples of each
- Recognize the typical use of active objects to allow asynchronous tasks to be requested without blocking a thread
- Understand the difference between multitasking using multiple threads and multiple active objects, and why the latter is preferred in Symbian OS code

Event-Driven Multitasking on Symbian OS

Synchronous and asynchronous requests

- When program code makes a function call to request a service - the service can be performed either synchronously or asynchronously

A **synchronous** function

- Performs a service to completion and then returns to its caller, usually returning an indication of its success or failure

An **asynchronous** function

- Submits a request as part of the function call and immediately returns to its caller
- The completion of the request occurs some time later

Event-Driven Multitasking on Symbian OS

After calling an asynchronous request

- The caller is free to perform other processing or it may simply wait, which is often referred to as "blocking"
- Upon completion the caller receives a signal which indicates the success or failure of the request

This signal is known as an event

- The code can be said to be event-driven

A timer wait is an example of a typical asynchronous call

- Another is the **`Read()`** method on the Symbian OS **`RSocket`** class which waits to receive data from a remote host

Threads in Symbian OS

Threads are scheduled pre-emptively by the kernel

- The kernel runs the highest-priority thread eligible
- Each thread may be suspended while waiting for a given event to occur and may resume whenever appropriate

The kernel controls thread scheduling

- Allowing the threads to share system resources by time-slice division - pre-empting the running of a thread if another higher-priority thread becomes eligible to run
- This constant switching to run the highest-priority ready thread is the basis of pre-emptive multitasking

Threads in Symbian OS

A context switch occurs when the current thread is suspended

- The context switch incurs a run-time overhead in terms of the kernel scheduler
- If the original and replacing threads are executing in <u>different processes</u> a larger overhead is incurred due swapping process memory and flushing the cache
- A 100 times slower than a thread context switch!

Event-Driven Multitasking

Asynchronously generated events can arise

- From external sources - such as user input or hardware peripherals that receive incoming data.
- By software - for example by timers or completing asynchronous requests

Event-Driven Multitasking

Events are managed by an event handler

- An event handler waits for an event and then handles it
- A high-level example of an event handler is a web-browser application
  - a) Waits for user input and responds by submitting requests to receive web pages which it then displays
  - b) The web browser may use a system server which waits to receive requests from its clients. The system server services the request and returns to waiting for another request. In servicing requests, the system server in turn submits requests to other servers, which later generate completion events.
- Each of the software components described is event-driven and needs to be responsive either to user input or to requests from the system
- It soon becomes complex!

Event Handling Considerations for Symbian OS

In response to an event, an event handler may request another service that will cause another event (and so on)

- The operating system must have an efficient event-handling model to handle each event as soon as possible after it occurs and in the most appropriate order
- It is important that user-driven events are handled rapidly to give feedback and a good user experience

Event Handling Considerations for Symbian OS

Code should avoid polling constantly between events

- This can lead to significant power drain and must be avoided on a battery-powered device

The system should instead wait in a low-power state

- The software should allow the operating system to move to an idle mode while it waits for the next event

The memory used by event-handling code is minimized

- And the processor resources are used efficiently

Active objects achieve these requirements and provide a model for lightweight event-driven multitasking

Active Objects and the Active Scheduler

Active objects and the active scheduler

- Collectively known as the "active object framework"
- Used to simplify asynchronous programming making it easy to write code:
  a) To submit asynchronous requests
  b) Manage their completion events
  c) Process the results
- In general, a Symbian OS application or server will consist of a single main event-handling thread with an associated active scheduler
- A number of active objects run in the thread
- Active objects have event-handling methods that are called by the active scheduler

Each active object encapsulates a task

- It requests an asynchronous service from its service provider and handles the completed event when the active scheduler calls it to do so

Active Objects and the Active Scheduler

The active object framework is used to schedule

- The handling of multiple asynchronous tasks in the same thread

All the active objects run in the same thread thus a switch between them incurs a lower overhead than a thread context switch

- This makes it generally the most appropriate choice for event-driven multitasking on Symbian OS

Active objects still run independently of each other

- In much the same way that threads are independent of each other in a process
- However, being in the same thread, memory may be shared more readily

Active Objects and the Active Scheduler

The active object framework

- Is an example of cooperative or non-pre-emptive multitasking
- Each active object function runs to completion before any other active object in that thread can start to perform an operation

When an active object is handling an event

- It cannot be pre-empted by any other running within that thread
- Note the thread itself is scheduled pre-emptively (previous slides)

Active Objects and the Active Scheduler

A Win32 application (i.e. running on Windows) uses a simple pattern of message loop and message dispatch

- On Symbian OS the active scheduler takes the place of the Windows message loop and the event-handling function of an active object acts as the message handler
- The event completion processing performed by the active scheduler is decoupled from the specific actions invoked by the event - these are performed by individual active objects
- e.g. email send event completes - the action removes the ‘sending’ dialog

A Note on Real Time Considerations

Some events require a response within a guaranteed time

- This is called "real-time" event handling
- For example, a real-time task may be required to keep the buffer of a sound driver supplied with sound data — a delay in response delays the sound decoding which results in the sound breaking up
- Other typical real-time requirements may be even more strict, say for low-level telephony

The various tasks have different requirements for real-time responses

- These can be represented by task priorities
- Higher-priority tasks must always be able to pre-empt lower-priority tasks in order to guarantee to meet their real-time requirements
- The shorter the response time required - the higher the priority

Active Objects are Not Suitable for Real-Time Tasks

When an active object is handling an event it may not be pre-empted by the event handler of another active object within the same thread

- Thus active objects are not suitable for real-time tasks

On Symbian OS, real-time tasks should be implemented using high-priority threads and processes, with the priorities chosen as appropriate for relative real-time requirements

Active Objects

Class `CActive`

- Understand the significance of an active object's priority level
- Recognize that the active object event handler method (`RunL()`) is non-pre-emptive
- Know the inheritance characteristics of active objects, and the functions they are required to implement and override
- Know how to correctly construct, use and destroy an active object

Active Objects

Introduction

- An active object requests an asynchronous service and handles the resulting completion event some time after the request
- It also provides a way to cancel an outstanding request and may provide error handling for exceptional conditions
- An active object class must derive directly or indirectly from class **`CActive`** defined in **`e32base.h`**

Active Object Class Construction

`CActive` is an abstract class with two pure virtual functions

- `RunL()` and `DoCancel()` - all concrete active object classes must inherit from `CActive`, define and implement these methods
- It also has a `TRequestStatus` member variable which is passed to asynchronous requests to receive the completion result

On construction

- Classes deriving from `CActive` must call the protected constructor of the base class
- Passing in a parameter to set the priority of the active object
- Like threads, all active objects have a priority value to determine how they are scheduled

Why Have Active Object Priorities?

When the asynchronous service associated with the active object completes it generates an event which the active scheduler detects.

1. The active scheduler determines which active object is associated with each event
2. The active scheduler calls the appropriate active object to handle the event

When an active object is handling an event

- It cannot be pre-empted until the event-handler function has returned back to the active scheduler
- It is quite possible that a number of events may complete before control returns to the scheduler ...

Why Have Active Object Priorities?

To resolve which active object gets to run next

- The scheduler orders the active objects in highest priority order - rather than in order of completion
- Otherwise, an event of low priority that completed just before a more important one would lock out the higher-priority event for an undefined period

The priority value is only use to determine the order in which event handlers are run

- If an active object with a high priority value receives an event while a lower-priority active object is already handling an event, the lower-priority event handler will not be pre-empted

Priorities

A set of priority values are defined

- In the **`TPriority`** enumeration of class **`CActive`**
- In general the priority value **`CActive::EPriorityStandard (=0)`** should be used unless there is good reason to do otherwise

Active Object Class Construction

As an additional part of construction the active object code should call a static function on the active scheduler **`CActiveScheduler::Add()`**

- This will add the active object to a list of event-handling active objects on that thread.
- The list is maintained by the active scheduler
- This list is ordered by the active objects' priorities with the highest-priority objects first

Active Object Class Construction

An active object typically owns a handle to an object

- To which it issues requests that complete asynchronously, such as a timer object of type **`RTimer`**
- This object is generally known as an asynchronous service provider and it may need to be initialized as part of construction.
- If the initialization can fail it must be performed as part of the second-phase construction

Submitting an Asynchronous Service Request

An active object class

- Supplies public "request issuer" methods for callers to initiate requests
- These will submit requests to the asynchronous service provider associated with the active object using a well-established pattern
- And later complete, generating an event

As follows ...

Submitting an Asynchronous Service Request

1. Check for previous outstanding requests

- Request methods should check that there is no request already submitted before attempting to submit another.
- Each active object must only ever have one outstanding request. Depending on the implementation, the code may:
  - Panic if a request has already been issued (if this scenario could only occur because of a programming error)
  - Refuse to submit another request, if it is legitimate to attempt to make more than one request
  - Cancel the outstanding request and submit the new one.

Submitting an Asynchronous Service Request

2. Issue the request

- The active object should then issue the request to the service provider, passing in its own **`iStatus`** member variable as the **`TRequestStatus&`** parameter
- The service provider will set this value to **`KRequestPending`** before initiating the asynchronous request

Submitting an Asynchronous Service Request

3. Call **`SetActive()`** to mark the object as "waiting"

- If the request is submitted successfully, the request method then calls the **`SetActive()`** method of the **`CActive`** base class
- To indicate to the active scheduler that a request has been submitted and is currently outstanding
- This call is not made until after the request has been submitted

Event Handling

Each active object class

- Must implement the pure virtual **`RunL()`** method of the **`CActive`** base class
- This is the event handler invoked when a completion event occurs from the associated asynchronous service provider
- The active scheduler selects the active object to handle the event and calls this method

The **`RunL()`** method

- Has a slightly misleading name as the asynchronous function has already run
- Perhaps a clearer description would be **`HandleEventL()`** or **`HandleCompletionL()`**

Event Handling

Typical implementations of `RunL()`

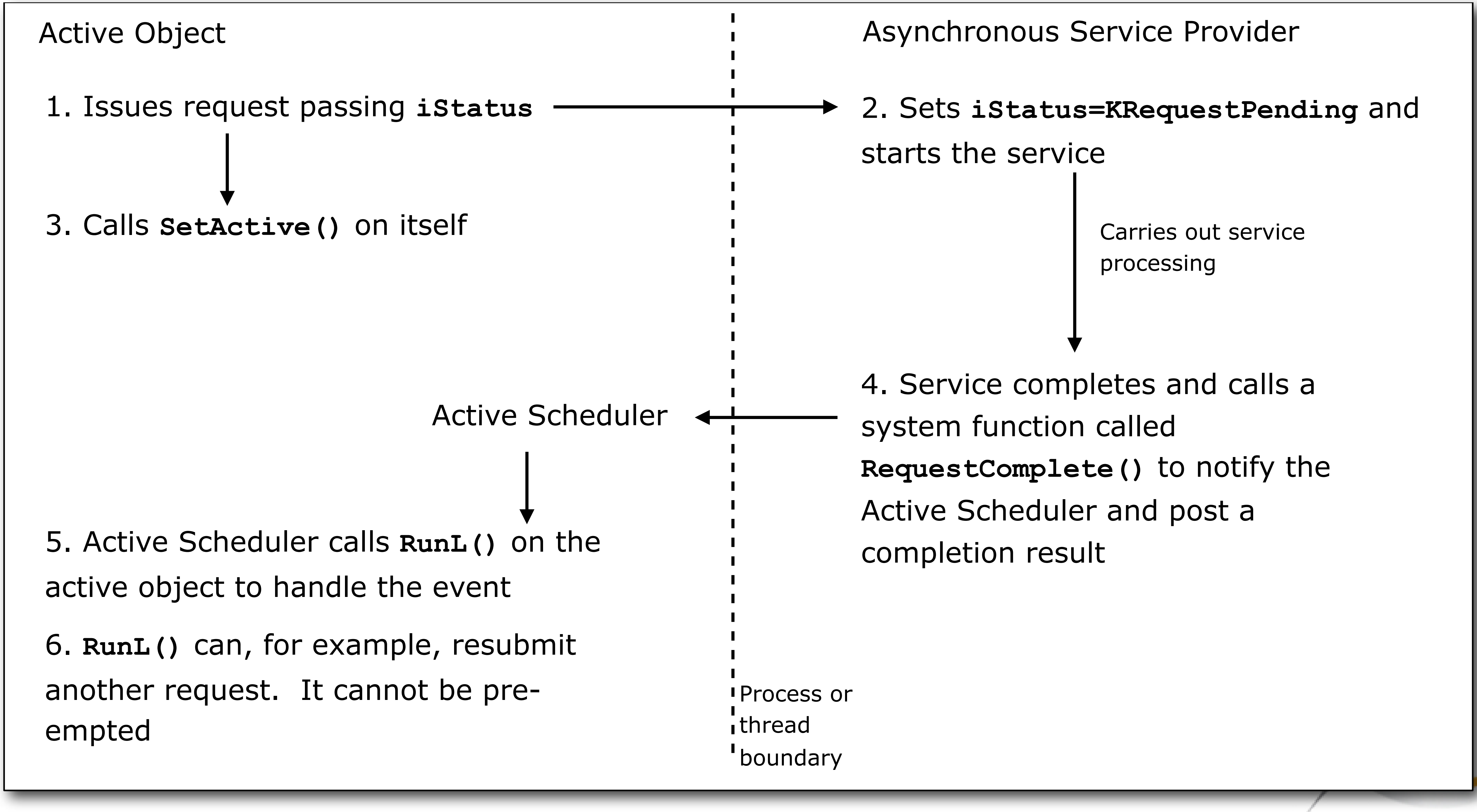
- Determine whether the asynchronous request succeeded by inspecting the completion code, a 32-bit integer value in the `TRequestStatus iStatus` object of the active object
- `RunL()` usually either issues another request or notifies other objects in the system of the event's completion
- The degree of complexity of `RunL()` code can vary considerably

Once `RunL()` is executing

- It cannot be pre-empted by other active objects' event handlers
- For this reason the code should complete as quickly as possible so that other events can be handled without delay

Event Handling

This diagram illustrates the basic sequence of actions performed when an active object submits a request to an asynchronous service provider. The request later completes, generating an event which is handled by RunL()



Active Object

Asynchronous Service Provider

1. Issues request passing **`iStatus`**

2. Sets **`iStatus=KRequestPending`** and starts the service

3. Calls **`SetActive()`** on itself

Carries out service processing

4. Service completes and calls a system function called **`RequestComplete()`** to notify the Active Scheduler and post a completion result

Active Scheduler

5. Active Scheduler calls **`RunL()`** on the active object to handle the event

6. **`RunL()`** can, for example, resubmit another request. It cannot be pre-empted

Process or thread boundary

Canceling an Outstanding Asynchronous Request

An active object

- Must be able to cancel any outstanding asynchronous requests it has issued
- For example, if the application thread in which it is running is about to terminate, it must stop the request

The `CActive` base class

- Implements a `Cancel()` method, which calls the pure virtual `DoCancel()` method and waits for the request's early completion
- Any implementation of `DoCancel()` should call the appropriate cancellation method on the asynchronous service provider

Canceling an Outstanding Asynchronous Request

`DoCancel()` can also include other processing

- But it should not leave or allocate resources and should not carry out any lengthy operations
- It is a good rule to restrict the method to cancellation, and any necessary cleanup associated with cancellation, rather than implementing any sophisticated functionality
- This is because a destructor should call **`Cancel()`** and may already have cleaned up resources that **`DoCancel()`** might require

It is not necessary to check ...

- Whether a request is outstanding before calling **`Cancel()`**
- It is safe to do so even if it is not currently active i.e. awaiting an event

Error Handling

From Symbian OS v6.0 onwards

- The **`CActive`** provides a virtual **`RunError()`** method which the active scheduler calls if a leave occurs in the **`RunL()`** method
- The method takes the leave code as a parameter and returns an error code to indicate whether the leave has been handled
- The default implementation does not handle the leave and simply returns the leave code passed to it

If the active object can handle any leaves occurring in **`RunL()`**

- It should override the default implementation of **`CActive::RunError()`** to handle the error and return **`KErrNone`**

There is no need to provide an override if no leaves can occur in **`RunL()`**

Error Handling

If **`RunError()`** returns a value other than **`KErrNone`** indicating that the leave has yet to be dealt with

- The active scheduler calls its own **`Error()`** function to handle it

The active scheduler

- Does not have any contextual information about the active object with which to perform error handling
- Thus it is preferable to manage error recovery within the **`RunError()`** method of the associated active object

Active Object Class Destruction

The destructor of a **`CActive`**-derived class should always call **`Cancel()`**

- To terminate any outstanding requests as part of cleanup code
- This should be done before any other resources owned by the active object are destroyed - in case they are used by the service provider or the **`DoCancel()`** method

The destructor code

- Should free all resources owned by the object including any handle to the asynchronous service provider

Active Object Class Destruction

The **`CActive`** base-class destructor is virtual

- Its implementation checks that the active object is not currently active
- It panics if any request is outstanding i.e. if **`Cancel()`** has not been called

The panic catches any programming errors

- Which could lead to the situation where a request completes after the active object to handle it has been destroyed
- This would otherwise result in a "stray signal" where the active scheduler cannot locate an active object to handle the event

Having verified the active object has no issued requests outstanding

- The **`CActive`** destructor removes the active object from the active scheduler

An Example of an Active Object Class

The following example

- Illustrates the use of an active object class to wrap an asynchronous service
- In this case a timer provided by the **`RTimer`** service

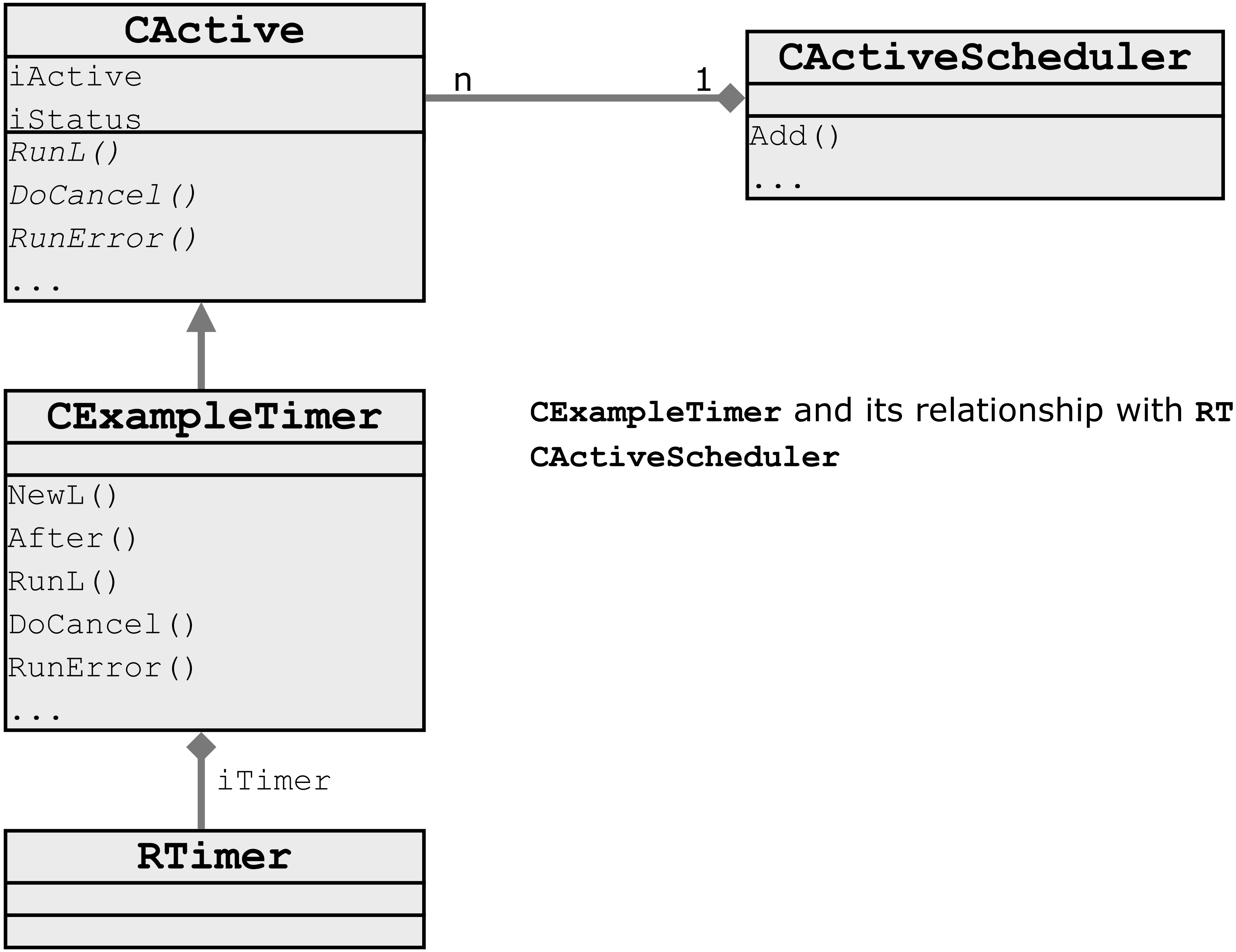
Symbian OS already supplies an abstract active object class **`CTimer`** which wraps **`RTimer`** and can be derived from

- However, the example is used here because it's a straightforward way of showing how to write an active object class.

The following slide

- Shows the classes involved and their relationship with the active scheduler

An Example of an Active Object Class



CActive

iActive

iStatus

RunL()

DoCancel()

RunError()

...

n

1

CActiveScheduler

Add()

...

CExampleTimer

NewL()

After()

RunL()

DoCancel()

RunError()

...

iTimer

RTimer

`CExampleTimer` and its relationship with **`RTimer`**, **`CActive`** and **`CActiveScheduler`**

`CExampleTimer` Class

```
class CExampleTimer : public CActive
  {
public:
  ~CExampleTimer();
  static CExampleTimer* NewL();
  void After(TTimeIntervalMicroSeconds32& aInterval);
protected:
  CExampleTimer();
  void ConstructL();   // Two-phase construction
protected:
  virtual void RunL(); // Inherited from CActive
  virtual void DoCancel();
  virtual TInt RunError(TInt aError);
private:
  RTimer iTimer;
  TTimeIntervalMicroSeconds32 iInterval;
  };
```

CExampleTimer Construction

```
CExampleTimer::CExampleTimer() : CActive(EPriorityStandard)
  { CActiveScheduler::Add(this); }

void CExampleTimer::ConstructL()
  {
  User::LeaveIfError(iTimer.CreateLocal());
  }

CExampleTimer* CExampleTimer::NewL()
  {...}

CExampleTimer::~CExampleTimer()
  {
  Cancel();
  iTimer.Close(); // Close the handle
  }
```

Create the asynchronous service provider

Standard 2-phase construction omitted for clarity

CExampleTimer::After()

Only allow one timer request to be submitted at a time - i.e. it should not already be active.

The caller must call **`Cancel()`** before submitting another

Start the **`RTimer`**

Mark this object active

```
void CExampleTimer::After(TTimeIntervalMicroSeconds32&
aInterval)
{
 if (IsActive())
    {
    _LIT(KExampleTimerPanic, "CExampleTimer");
    User::Panic(KExampleTimerPanic, KErrInUse));
    }

  iInterval = aInterval;
  iTimer.After(iStatus, aInterval);
  SetActive();
  }
```

`RunL()` and `DoCancel()`

Event handler method

If an error occurred leave and deal with the problem in **`RunError()`**

Otherwise, log the timer completion

Resubmit the timer request

Cancel the timer

```
void CExampleTimer::RunL()
  {
   User::LeaveIfError(iStatus.Int());


  _LIT(KTimerExpired, "Timer Expired\n");
  RDebug::Print(KTimerExpired);

  iTimer.After(iStatus, iInterval);
  SetActive();
  }

void CExampleTimer::DoCancel()
  {
   iTimer.Cancel();
  }
```

Error Handling and Event Handling

If no error occurred

- The **`RunL()`** event handler logs the timer completion to debug output using **`RDebug::Print()`**
- **`RunL()`** resubmits the timer request with the stored interval value

Once the timer request has started, it continues to expire and be resubmitted

- Until it is stopped by a call to the **`Cancel()`** method on the active object

Error Handling and Event Handling

The **`RunL()`** event handler checks the active object's **`iStatus`** result

- If **`iStatus`** contains a value other than **`KErrNone`** it leaves so that the **`RunError()`** method can handle the problem

In this case - the error handling is very simple:

- The error returned from the request is logged to debug output
- This could have been performed in the **`RunL()`** method
- But it has been separated into the **`RunError()`** method to demonstrate how to use the active object framework to split error handling from the main logic of the event handler

CExampleTimer::RunError

Called if **`RunL()`** leaves (**`aError`** contains the leave code)

Logs the error

Error has been handled

```
TInt CExampleTimer::RunError(TInt aError)
   {
   _LIT(KErrorLog, "Timer error %d");
   RDebug::Print(KErrorLog, aError);
   return (KErrNone);
   }
```

Active Objects

The Active Scheduler

- Understand the role and characteristics of the active scheduler
- Know that **`CActiveScheduler::Start()`** should only be called after at least one active object has an outstanding request
- Recognize that a typical reason for a thread to fail to handle events may be that the active scheduler has not been started or has been stopped prematurely
- Understand that **`CActiveScheduler`** may be sub-classed, and the reasons for creating a derived active scheduler class

Creating and Installing the Active Scheduler

Most threads have an active scheduler

- Usually created and started implicitly by a framework

  (e.g. CONE for the GUI framework)

- Server code must create and start an active scheduler explicitly before active objects can be used
- Console-based test code must create an active scheduler in its main thread if it depends on components which use active objects

Creating and Installing the Active Scheduler

The code to create and install an active scheduler:

```
CActiveScheduler* scheduler = new(ELeave) CActiveScheduler;
CleanupStack::PushL(scheduler);
CActiveScheduler::Install(scheduler);
```

Starting the Active Scheduler

Once an active scheduler has been created and installed its event-processing wait loop is started by a call to the static **`CActiveScheduler::Start()`** method

- But the call to **`Start()`** enters the event-processing loop and does not return until a corresponding call is made to **`CActiveScheduler::Stop()`**

There <u>must be at least one asynchronous request</u> issued via an active object before the active scheduler is started

- So that the thread's request semaphore is signaled and the call to **`User::WaitForAnyRequest()`** completes
- If <u>no request is outstanding</u> the thread simply enters the wait loop and sleeps indefinitely

The Active Scheduler Wait Loop

When an asynchronous request is complete

- The asynchronous service provider calls **`User::RequestComplete()`** if the service provider and requestor are in the same thread
- If they are in different threads **`RThread::RequestComplete()`** is called

It passes **`RequestComplete()`**

- The **`TRequestStatus`** associated with the request
- A completion result
  - Typically one of the standard error codes such as **`KErrNone`** or **`KErrNotFound`**

`RequestComplete()` sets the value of **`TRequestStatus`** to the given error code

- Generates a completion event in the requesting thread by signaling the thread's request semaphore

The Active Scheduler Wait Loop

While the request is outstanding the requesting thread runs in the active scheduler's event-processing loop

- When it is not handling other completion events the active scheduler suspends the thread by calling **`User::WaitForAnyRequest()`**
- Which waits for a signal to the thread's request semaphore

The Active Scheduler Wait Loop

When a signal is received

- The active scheduler determines which active object should handle it
- It inspects its priority-ordered list of active objects to determine which have outstanding requests

  i.e. those which have their their **`iActive`** Boolean to **`ETrue`** (which is set after by the call to **`CActive::SetActive()`** after the request is submitted)
- The active scheduler inspects the active object's **`TRequestStatus`** member variable to see if it is set to a value other than **`KRequestPending`**
- Indicating that the active object is associated with a request that has completed and that its event handler code should be called

The Active Scheduler Wait Loop

Having found a suitable active object

- The active scheduler clears the active object's **`iActive`** boolean flag and calls its **`RunL()`** event handler

`RunL()` handles the event

- Carrying out any processing as required
  - It may also resubmit a request or generate an event on another object in the system
- Note: While it is running other events may be generated but **`RunL()`** is not pre-empted

`RunL()` completes

- The active scheduler then resumes control
- And determines whether any other requests have completed

The Active Scheduler Wait Loop

Once **`RunL()`** has completed

- The active scheduler re-enters the event processing wait loop by issuing another **`User::WaitForAnyRequest()`** call

`User::WaitForAnyRequest()`

- Checks the thread's request semaphore
  a) If no other requests have completed: Suspends it
  b) If the semaphore indicates that other events were generated while the previous **`RunL()`** was running: Returns immediately and repeats active object lookup and event handling

The Active Scheduler Wait Loop

Event-processing loop pseudo-code

Suspend the thread until an event occurs

1. Thread wakes when the request semaphore is signaled
2. Inspect each active object added to the scheduler, in order of decreasing priority
3. Call the event handler of the first which is active & completed

Get the next active object in the priority queue that is waiting on an event and has **`iStatus!=KRequestPending`**

Found an active object ready to handle an event

Reset the **`iActive`** status to indicate it is not active

Call the active object's <u>event handler</u> in a TRAP

Event handler left, so call **`RunError()`** on the active object

`RunError()` didn't handle the error, call **`CActiveScheduler::Error()`**

Event handled, break out of lookup loop and resume

End of FOREVER loop

```
EventProcessingLoop()
  {
  User::WaitForAnyRequest();


  FOREVER
    {
     if (activeObject->IsActive())
          &&
        (activeObject->iStatus!=KRequestPending)
         {
         activeObject->iActive = EFalse;
         TRAPD(r, activeObject->RunL());
         if (KErrNone!=r)
           {
            r = activeObject->RunError();
            if (KErrNone!=r)
                Error(r);
           }
         break;
         }
    }
  }
```

Stopping the Active Scheduler

The active scheduler is stopped

- By a call to **`CActiveScheduler::Stop()`**, usually made in **`RunL()`**

When the method that calls **`CActiveScheduler::Stop()`** completes i.e. returns

- The outstanding call to **`CActiveScheduler::Start()`** also returns

Stopping the active scheduler

- Breaks off event handling in the thread
- It should only be called by the main active object controlling the thread

  So you are unlikely to do this in a GUI application

Extending the Active Scheduler

`CActiveScheduler` is a concrete class

- It can be used "as is" but it can also be subclassed
- It defines two virtual functions which may be extended: **`Error()`** and **`WaitForAnyRequest()`**

The **`WaitForAnyRequest()`** function by default just calls **`User::WaitForAnyRequest()`**

- But it may be extended e.g. to perform some processing before or after the wait
- If it is overridden it must either call the base-class function or make a call to **`User::WaitForAnyRequest()`** directly

Extending the Active Scheduler

If a leave occurs in a **`RunL()`** event handler

- The active scheduler passes the leave code to **`RunError()`**

If **`RunError()`** cannot handle the leave

- It returns the leave code and the active scheduler passes it to its own **`Error()`** method

By default **`Error()`** raises a panic

- **`E32USER-CBASE 47`**
- But it may be overridden to handle the error
- e.g. by calling an error resolver to obtain the textual description of the error and displaying it to the user or logging it to file

A Word of Caution

If the active object code

- Is dependent upon particular specializations of the active scheduler
- It will not be portable to run in other threads managed by more basic active schedulers.

Furthermore

- Any additional code added to extend the active scheduler should be straightforward
- And must avoid holding up event handling in the entire thread by performing complex or slow processing

Threads Without Active Schedulers

There are a few threads which intentionally do not have an active scheduler and thus cannot use active objects or components that use them

- The Java implementation does not support an active scheduler

  native Java methods may not use active objects.

- The C Standard Library (**`STDLIB`**) thread has no active scheduler, thus standard library code cannot use active objects. Functions provided by the Standard Library may however be used in active object code, for example in an initialization or **`RunL()`** method

- OPL does not provide an active scheduler and C++ extensions to OPL (OPXs) must not use active objects or any component which uses them.

  OPL is an interpreted language generated using an entry-level development tool that enables rapid development of applications.

Active Objects

Canceling an Outstanding Request

- Understand the different paths in code that the active object uses when an asynchronous request completes normally, and as the result of a call to `Cancel()`

`CActive::Cancel()`

`CActive::Cancel()`

- Invokes the derived class's implementation of **`DoCancel()`**
- **`DoCancel()`** should never contain code which can leave or allocate resources as it will be called from within the destructor

Internally the active object

- Must never call the **`DoCancel()`** method directly to cancel a request
- It should call **`CActive::Cancel()`** (to invoke **`DoCancel()`** and handle the resulting cancellation event, as the next slides describe...)

Canceling an Outstanding Request

What happens when **`CActive::Cancel()`** is called?

- First it determines if the active object it has been called on actually has an outstanding request
- It does this by checking whether the **`iActive`** flag is set by calling **`CActive::IsActive()`**

Canceling an Outstanding Request

If the active object does have an outstanding request

- **`CActive::Cancel()`** calls **`DoCancel()`** - a pure virtual method in **`CActive`**
  - Which must be implemented by the derived active object class

When implementing **`DoCancel()`**

- The code does not need to check if there is an outstanding request
- Because if there is no outstanding request - **`DoCancel()`** would not have been called
- **`DoCancel()`** must cancel an outstanding request on the encapsulated asynchronous service provider by calling the cancellation method it provides

Canceling an Outstanding Request

Having called **`DoCancel()`**

- **`CActive::Cancel()`** then calls **`User::WaitForRequest()`** passing in a reference to its **`iStatus`** member variable

`CActive::Cancel()` is a <u>synchronous</u> function

- it does not return until both **`DoCancel()`** has returned and the original asynchronous request has completed. Thus:
  - **`DoCancel()`** should not perform any lengthy operations
  - The thread is blocked until the asynchronous service provider posts a cancellation notification **`KErrCancel`** into **`iStatus`**
- **`CActive::Cancel()`** resets the **`iActive`** member of the active object to reflect that there is no longer an asynchronous request outstanding

Canceling an Outstanding Request

The cancellation event

- Is handled by the **`Cancel()`** method of the active object rather than by the active scheduler
- **`RunL()`** will not be called

The **`CActive::Cancel()`** method performs all the generic cancellation code

A derived active object class

- Only uses **`DoCancel()`** to call the appropriate cancellation function on the asynchronous service provider
- And to perform any cleanup necessary

`DoCancel()` should not call **`User::WaitForRequest()`**

- This will upset the thread semaphore count

Stray Signal Panics

When an active object is about to be destroyed it must ensure that it is not awaiting completion of a pending request

- This is because `CActive`'s destructor removes the active object from the active scheduler list

If any outstanding request were to complete later it would generate an event for which there is no associated active object

- This causes a stray signal panic

`CActive::~CActive()`

To avoid stray signal panics

- The destructor of the **`CActive`** base class checks that there is no outstanding request before removing the object from the active scheduler
- It will raise an **`E32USER-CBASE 40`** panic if there is to highlight the problem

  This panic is easier to trace than a stray signal panic
- For this reason **`Cancel()`** should be called in the destructor of every derived active object class

Active Objects

Background Tasks

- Understand how to use an active object to carry out a long-running (or background) task
- Demonstrate an understanding of how self-completion is implemented

Background Tasks

Besides encapsulating asynchronous service providers

- Active objects can also be used to implement long-running tasks which would otherwise need to run in a lower-priority background thread
- The task must be divisible into multiple short increments

  e.g. preparing data for printing, performing background recalculations and compacting a database

- The increments are performed in the event handler of the active object

  they must be short since `RunL()` cannot be pre-empted once it is running

Background Tasks

The active object should be assigned a low priority

- Such as `CActive::TPriority::EPriorityIdle (=-100)` which determines that a task increment only runs when there are no other events to handle
- Known as idle time

If the task consists of a number of different steps

- The active object must track the progress as a series of states
- Implementing it using a state machine

Background Tasks

The active object

- Drives the task by generating its own events to invoke the event handler
- That is instead of calling an asynchronous service provider it completes itself by calling **`User::RequestComplete()`** on its own **`iStatus`** object
- So the active scheduler calls its event handler
- In this way it continues to resubmit requests until the entire task is complete

Background Tasks

A typical example is shown in the following sample code

- All the relevant methods are shown in the class declarations
- But only the implementations relevant to this discussion are given
- Error handling is also omitted for clarity
- **`StartTask()`**, **`DoTaskStep()`** and **`EndTask()`** perform small, discrete chunks of the task that can be called directly by the **`RunL()`** method of the low-priority active object

Background Tasks: `CLongRunningCalculation`

```
class CLongRunningCalculation : public CBase
  {
public:
  static CLongRunningCalculation* NewL();
  TBool StartTask();
  TBool DoTaskStep();
  void EndTask();
  ...
  };

TBool CLongRunningCalculation::DoTaskStep()
  {
  ...
  ...
  ...
  ...
  }
```

Initialization before starting the task

Performs a short task step

Destroys intermediate data

Do a short task step, returning

`ETrue` if there is more of the task to do

`EFalse` if the task is complete

Omitted for clarity

Background Tasks: `CBackgroundRecalc` Active Object

`NewL()`, destructor etc are omitted for clarity

`iCalc` is the long running task - other active objects have an asynchronous service provider

`iCallerStatus` is to notify the caller on task completion

```
class CBackgroundRecalc : public CActive
  {
public:
  ...
public:
  void PerformRecalculation(TRequestStatus& aStatus);
protected:
  CBackgroundRecalc();
  void ConstructL();
  void Complete();
  virtual void RunL();
  virtual void DoCancel();
private:
  CLongRunningCalculation* iCalc;
  TBool iMoreToDo;
  TRequestStatus* iCallerStatus;
};
```

Construction

Low priority task

```
CBackgroundRecalc::CBackgroundRecalc()
: CActive(EPriorityIdle)
  { CActiveScheduler::Add(this); }
```

Background Tasks: `PerformRecalculation`

`CBackgroundRecalc` is effectively an asynchronous service provider

Debugging house keeping

`iCalc` initializes the task

Self-completion to generate an event

```
void CBackgroundRecalc::PerformRecalculation(TRequestStatus& aStatus)
  {
  iCallerStatus = &aStatus;
  *iCallerStatus = KRequestPending;

  _LIT(KExPanic, "CActiveExample");
  __ASSERT_DEBUG(!IsActive(), User::Panic(KExPanic, KErrInUse));

  iMoreToDo = iCalc->StartTask();
  Complete();
  }
```

`Complete()`

Generates an event on itself by completing on **`iStatus`**

```
void CBackgroundRecalc::Complete()
  {
  TRequestStatus* status = &iStatus;
  User::RequestComplete(status, KErrNone);
  SetActive();
  }
```

Background Tasks: `RunL` & `DoCancel`

Performs the background task in increments

Resubmit request for next increment of the task or stop

No more to do - task is complete

Allow `iCalc` to cleanup any intermediate data

Notify the caller

Do another step and self-complete to generate event

`DoCancel`

Give `iCalc` a chance to perform cleanup

Notify the caller that cancellation has occurred

```
void CBackgroundRecalc::RunL()
  {
  if (!iMoreToDo)
    {
    iCalc->EndTask();
    User::RequestComplete(iCallerStatus,
                          iStatus.Int());
    }
  else
    {
    iMoreToDo = iCalc->DoTaskStep();
    Complete();
    }
  }

void CBackgroundRecalc::DoCancel()
  {
  if (iCalc)
      iCalc->EndTask();

  if (iCallerStatus)
    User::RequestComplete(iCallerStatus, KErrCancel);
  }
```

Active Objects

Common Problems

- Know some of the possible causes of stray signal panics, unresponsive event handling and blocked threads

Stray Signal Panics

The most commonly encountered problem when writing active object code is a “stray signal” panic (**`E32USER-CBASE 46`**)

- It occurs when the active scheduler receives a completion event but <u>cannot find an active object</u> to handle it
- i.e. one that is currently active and has a completed **`iStatus`** result (indicated by a value other than **`KRequestPending`**)

Stray Signal Panics

Stray signals can arise for the following reasons:

- **`CActiveScheduler::Add()`** was not called when the active object was constructed
- **`SetActive()`** was not called following the submission of a request to the asynchronous service provider
- The asynchronous service provider completed the **`TRequestStatus`** of an active object more than once, either:
  a) Because of a programming error in the asynchronous service provider
  b) Because more than one request was submitted simultaneously on the same active object

Unresponsive Event Handling

When using active objects for event handling in, for example, a UI thread

- Event-handler methods must be kept short to keep the UI responsive
- No active object should have a monopoly on the active scheduler that prevents other active objects from handling events

Active objects should be "cooperative" and should not:

- Have lengthy **`RunL()`** or **`DoCancel()`** methods
- Repeatedly resubmit requests that complete rapidly and prevent other active objects from handling events
- Have a higher priority than is necessary

Blocked Thread

A thread can block and thus prevent an application's UI from remaining responsive, for a variety of reasons including the following:

- A call to `User::After()` which blocks a thread until the time specified as a parameter has elapsed
- Incorrect use of the active scheduler

  Before the active scheduler is started, there must be at least one asynchronous request issued, via an active object, so that the thread's request semaphore is signaled and the call to `User::WaitForAnyRequest()` completes

  If no request is outstanding, the thread simply enters the wait loop and sleeps indefinitely
- Use of `User::WaitForRequest()` to wait on an asynchronous request rather than use of the active object framework

Active Objects

- ✓ Event-Driven Multitasking on Symbian OS
- ✓ Class **`CActive`**
- ✓ The Active Scheduler
- ✓ Canceling an Outstanding Request
- ✓ Background Tasks
- ✓ Common Problems