Lecture 17
CS 2506

Symbian OS
Active Objects

Symbian Features

- The main component is the kernel – it manages the system memory and schedules programs for execution. It also allocates shared resources and handles any functionality that requires privileged access to the CPU.

- Other components are *the base libraries* (APIs), used not only by the applications but also by OS components, *application services*, *engines and protocols* that provide access for programs to core application data (e.g., list of contacts, calendar, to-do list) and services, *communications architecture* (APIs that include TCP/IP over cellular radio as well as local communication protocols – Bluetooth, IR, USB) and *middleware services*.

- It is an event-driven multitasking, multi-threaded OS.

- A process always contains a main thread and additional threads if needed.

- The computing model for applications and the OS itself is client/server. A server always runs in a different thread/process; it has no user interface and acts mainly as an engine which provides some functionality to the client, e.g. the file system server execute commands for file creation, read/write.

AOs Concurrency

- As process/thread switching is time consuming, multiple active objects are used within one thread.
- Active objects (AO) represent a model of lightweight, event-driven multitasking. Objects run independently of each other.
- A switch between AOs that run in the same thread incurs a lower overhead than thread context switching.
- An application/server usually is a single, main event-handling thread.

- The thread consists of an active scheduler (event dispatcher) and a set of AOs, each representing a task.
- The active scheduler waits in a loop for an event (at a semaphore), and then invokes the event handler of the AO expecting the event. The active scheduler then waits for the next event,…there is an infinite loop.
- Each AO requests an asynchronous service, waits while it is serviced, handles the request completion event and communicates with other AOs if necessary.
- Once an AO is handling an event, it cannot be preempted by the event handler of another AO.
- Real-time issues are managed by process/thread priority.

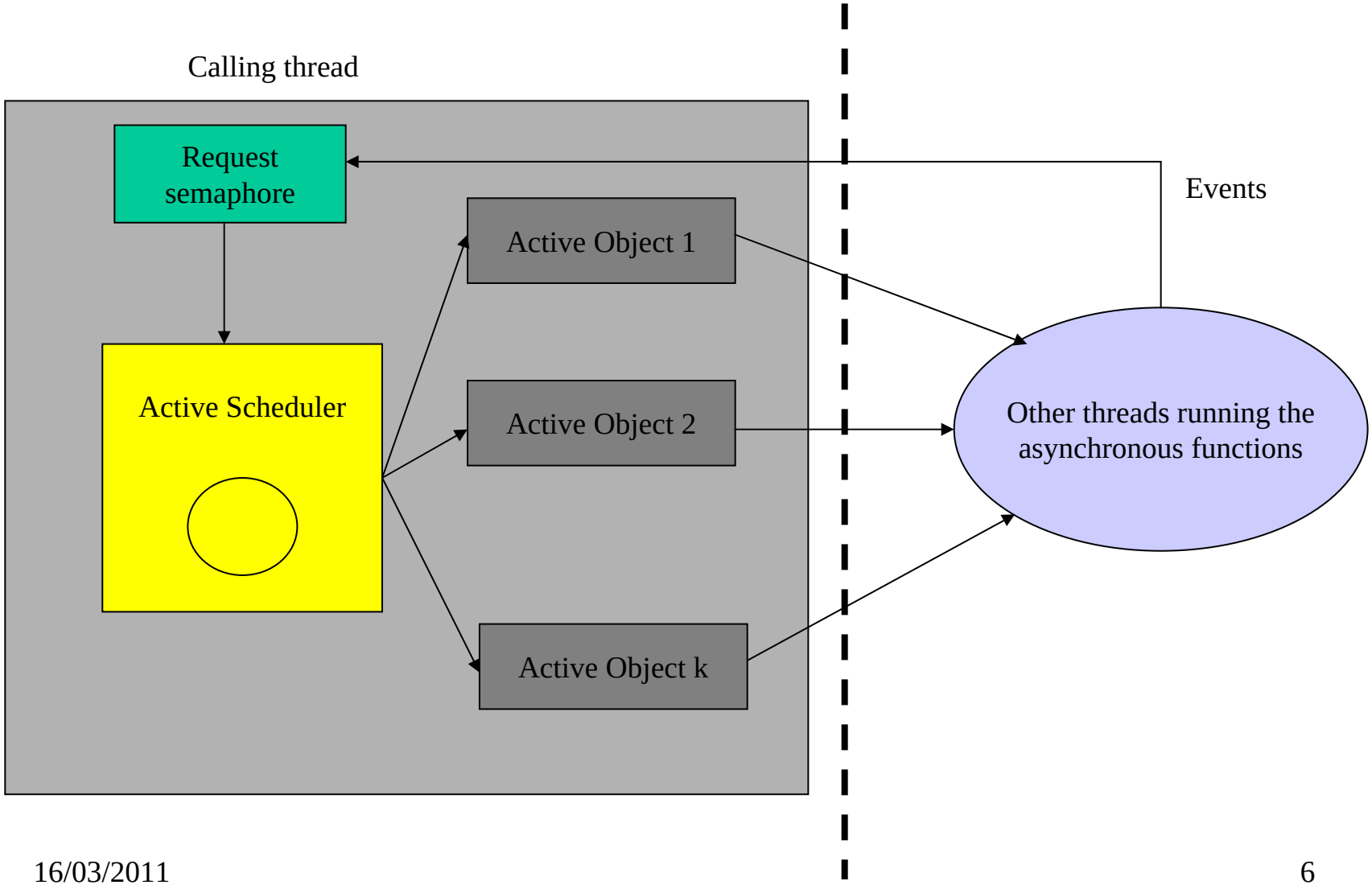
Threads

- Threads are scheduled preemptively by the kernel – time-sharing.
- They have priorities, and when more have the same priority, the strategy is round-robin.
- A thread has an absolute priority (Rthread::SetPriority()) and optionally combined with the priority of the process in which it runs.
- Relative priorities: EPriorityMuchLess, EPriorityNormal, EPriorityMuchMore,…
- Example: TProcessPriority::EPriorityHigh(=450) and SetPriority() is called with TThreadPriority of EPriorityMuchLess →450-20 = 430.
- The thread priority can be independent of the process.
- By default, the priority is set to Normal.

Operations on Threads

- Suspend(), Resume()
- Kill()/Terminate()
- Panic() – stop for highlighting a programming error.
- If the main thread of a process is ended, the process is terminated as well.
- In Kernel EKA2, the security model ensures that a thread is always protected from other threads – a thread cannot stop another one.
- The manner in which a thread was stopped can be determined by calling ExitType(), ExitReason() and ExitCategory().

Scheduler-AO model



AO practice

- An AO class must derive from class CActive.
- An AO has a priority value; classes deriving from CActive must call the protected constructor of the base class and pass a parameter to set the priority of the AO.
- When the asynchronous service associated with the AO completes, it generates an event. The active scheduler detects events, determines the associated AOs and calls the appropriate AO to handle the event.

- If multiple events have occurred before control returns to the scheduler, they are handled sequentially, in the order of their priorities.
- The recommended priority value is EPriorityStandard(=0).
- As part of the construction, the AO code should call a static function on the active scheduler, CActiveScheduler::Add(). This adds the AO to the list of active AO, managed by the active scheduler.
- An AO typically owns an object to which it issues requests that complete asynchronously, generating an event. For example, a timer object of type RTimer.

Submitting Requests

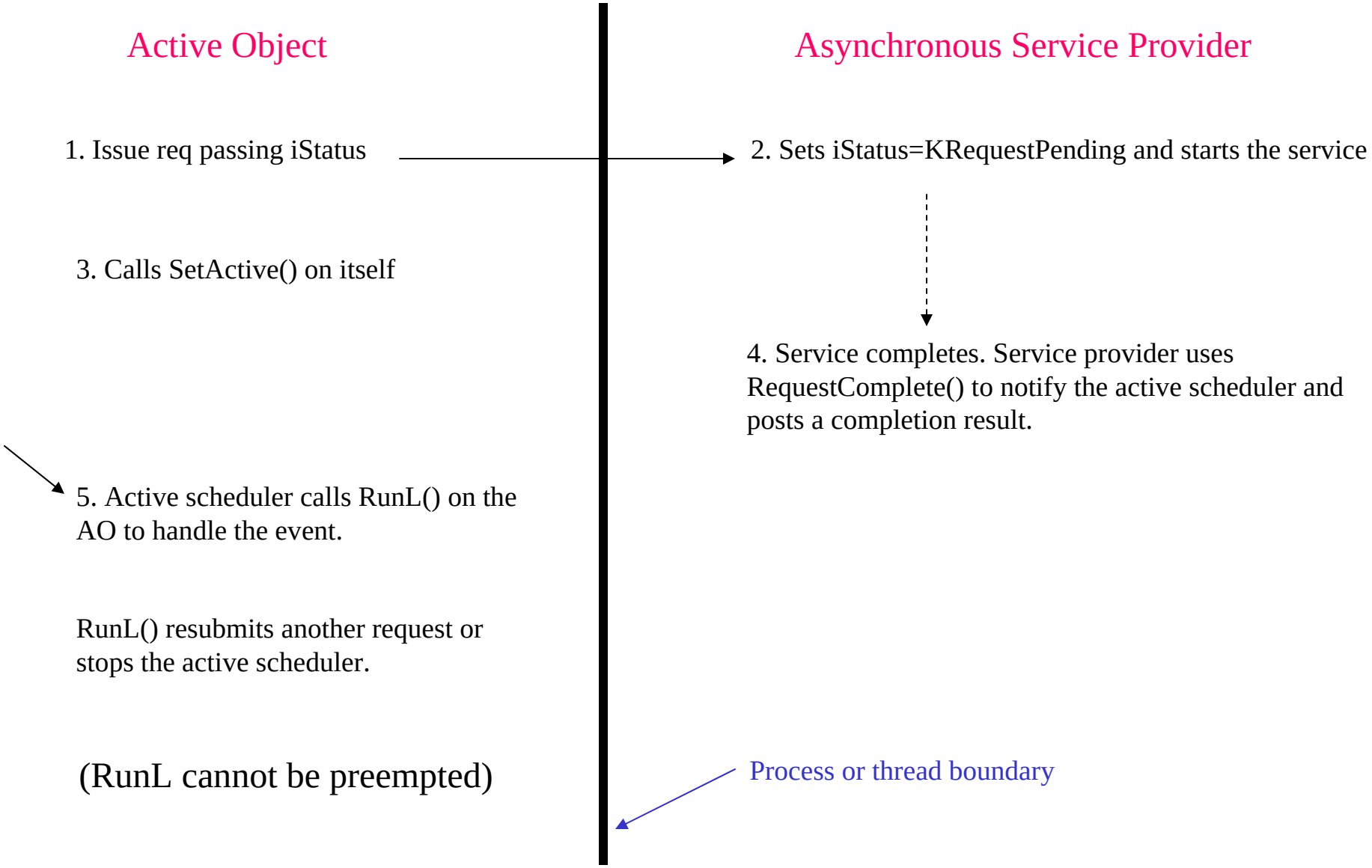
1. The request method should check if there is a request already submitted – each AO can have only one outstanding request. The result of the check depends on the implementation (panic, refuse, cancel/submit).
2. The AO then issues the request to the service provider, passing in its iStatus member variable as the TRequestStatus& parameter.
3. If the request is submitted successfully, the request method then calls the SetActive() method to indicate to the active scheduler that a request is currently outstanding.

Event Handling

- When a completion event occurs (from the associated service provider), the active scheduler calls RunL() on the associated AO.

- Typically, RunL() code determines if the asynchronous request succeeded by inspecting the completion code in the TRequestStatus object – a 32-bit integer value.
- Depending on the result, RunL() either issues another request or notifies other objects in the system.
- As RunL() cannot be preempted, it should complete quickly.

Event-driven Multitasking using AO



Cancellation

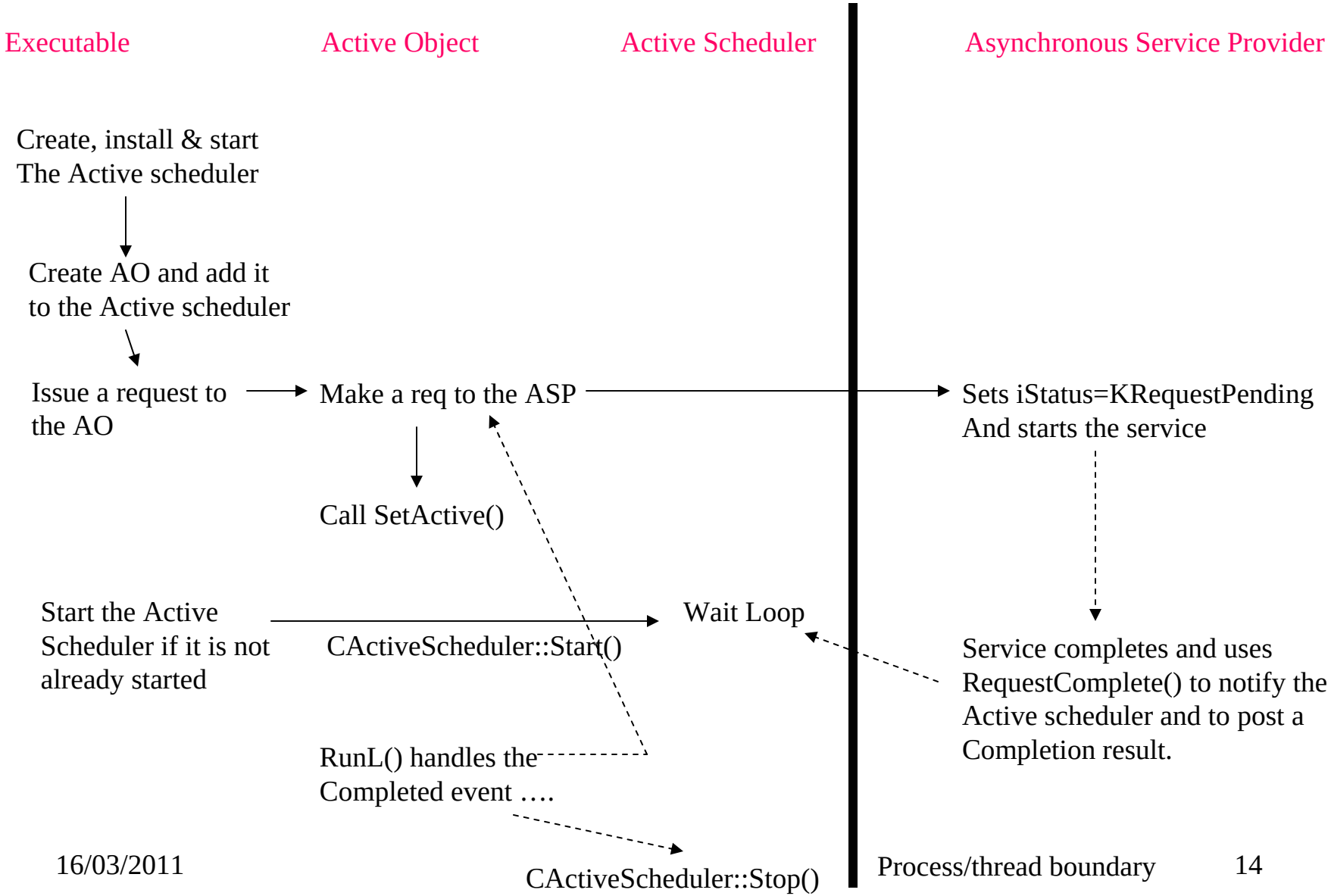
- AO is able to cancel any outstanding requests it has issued.

- The CActive class implements a Cancel() method which calls the pure virtual DoCancel() method (the AO must implement it) and waits for the request's early completion.
- DoCancel() calls the appropriate cancellation method on the service, but can also include other processing. The golden rule is to be short.

Destruction

- As part of the cleanup code, the destructor of a CActive-derived class should always call Cancel() to terminate any outstanding requests.
- This is done before calling DoCancel() method.
- The destructor code should free all resources owned by the object, including any handle to the asynchronous service provider.
- The destructor implementation checks that the AO is not active. It panics if any request is outstanding.

Roles and Actions



AO's Actions

- An AO provides at least one method for clients to initiate requests.
- After submitting a request to an asynchronous service provider, the AO must call SetActive() upon itself. This sets the iActive flag, which indicates an outstanding request. This flag will be used by the active scheduler upon receipt of an event, and by the class upon destruction (to determine if the AO can be removed from the active scheduler).
- An AO must submit only one request at a time.
- The AO implements the RunL() and DoCancel() methods.
- An AO must ensure that it is not awaiting completion of a pending request when it is about to be destroyed.

Asynchronous Service Provider

- Before beginning to process the request, the object must set the incoming TRequestStatus value to KRequestPending to indicate to the active scheduler that a request is ongoing.
- When processing is completed, it must set the TRequestStatus value to a result code other than KRequestPending, by calling RequestComplete().
- The RequestComplete() method generates an event that notifies the completion of the service.
- The service provider must supply a corresponding cancellation method for each request – this should cancel an outstanding request immediately, posting KErrCancel into the TRequestStatus object.

The Active Scheduler

- Suspends the thread by calling User::WaitForAnyRequest(). When an event is generated, it resumes the thread and inspects the list of AOs.
- Ensures that each request is handled only once. It resets the iActive flag of an AO before calling its handler method. This allows the AO to issue a new request from its RunL() event handler…
- Raises panic when the request semaphore has been notified of an event, but the active scheduler cannot find the corresponding AO (iActive set to Etrue and TRequestStatus indicating completion).

Request Completion

1. The normal procedure.
2. The request cannot begin if insufficient resources are available, or invalid parameters are passed. The service provider should include a function that neither leaves nor returns an error code (typically, returns void). The request completes immediately, and an error is posted in TRequestStatus.
3. The req is issued and Cancel() is called before its completion. The AO calls the appropriate cancellation function of the provider.. it terminates and returns KErrCancel as quickly as possible – Cancel() is blocking.
4. The request is issued and Cancel() is called after the request has completed. The provider ignores the call !