Symbian OS

Active Objects and Threads

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Symbian OS courses at the University of Applied Sciences in Hagenberg, Austria (http://www.fh-hagenberg.at/)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- Parts of these materials are based on information from Symbian Press-books published by John Wiley & Sons, Ltd. This document contains copyright materials which are proprietary to Symbian, UIQ, Nokia and SonyEricsson. “S60™” is a trademark of Nokia. “UIQ™” is a trademark of UIQ Technology. Pictures of mobile phones or applications are copyright their respective manufacturers / developers. “Symbian ™”, “Symbian OS ™” and all other Symbian-based marks and logos are trademarks of Symbian Software Limited and are used under license. © Symbian Software Limited 2006.

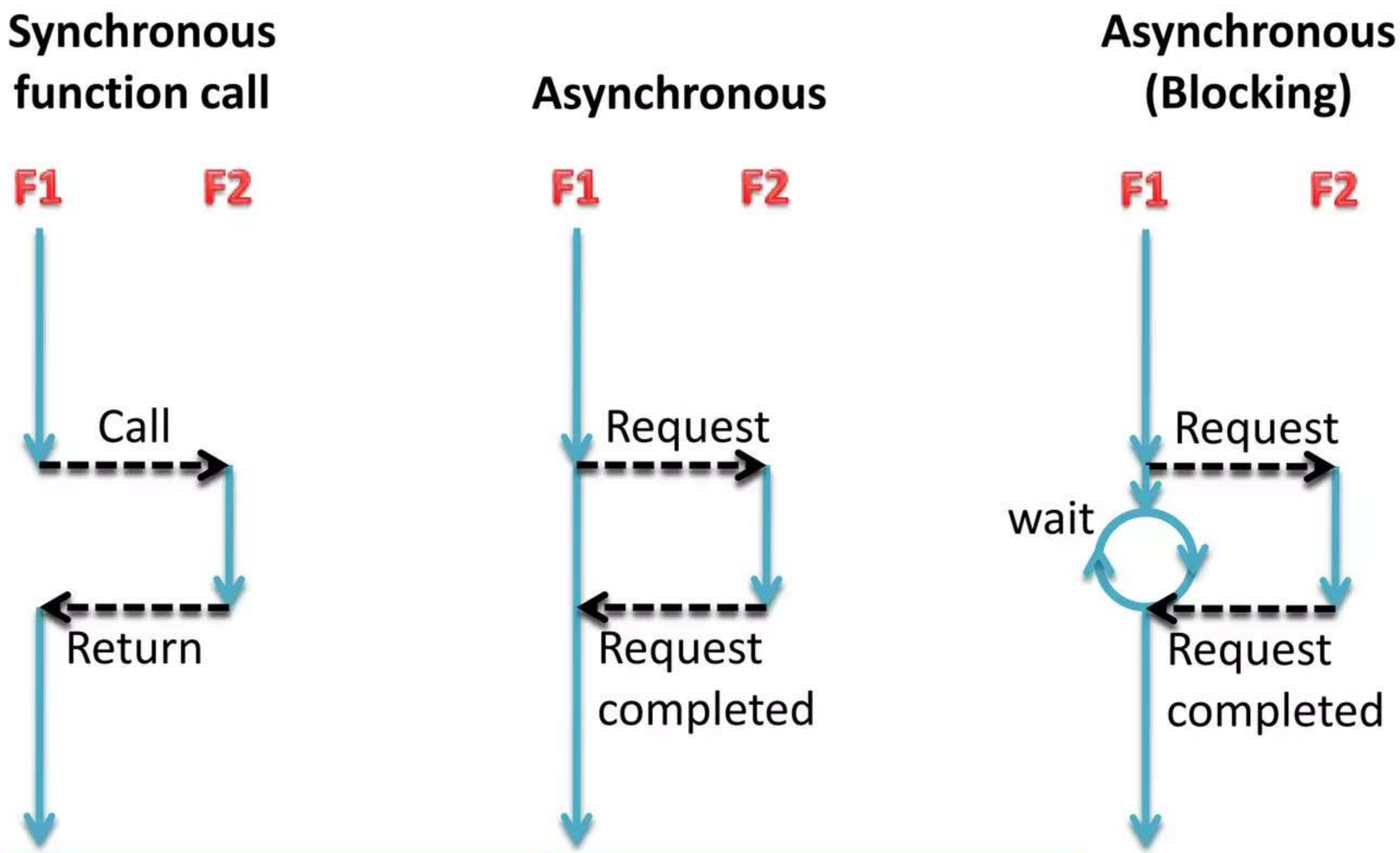
Contents

- Concepts of asynchronous processing
- Threads vs. Active Objects
- Using AOs in Symbian OS
- Long-running background tasks

Overview and motivation

Asynchronous Processing

Synchronous / Asynchronous



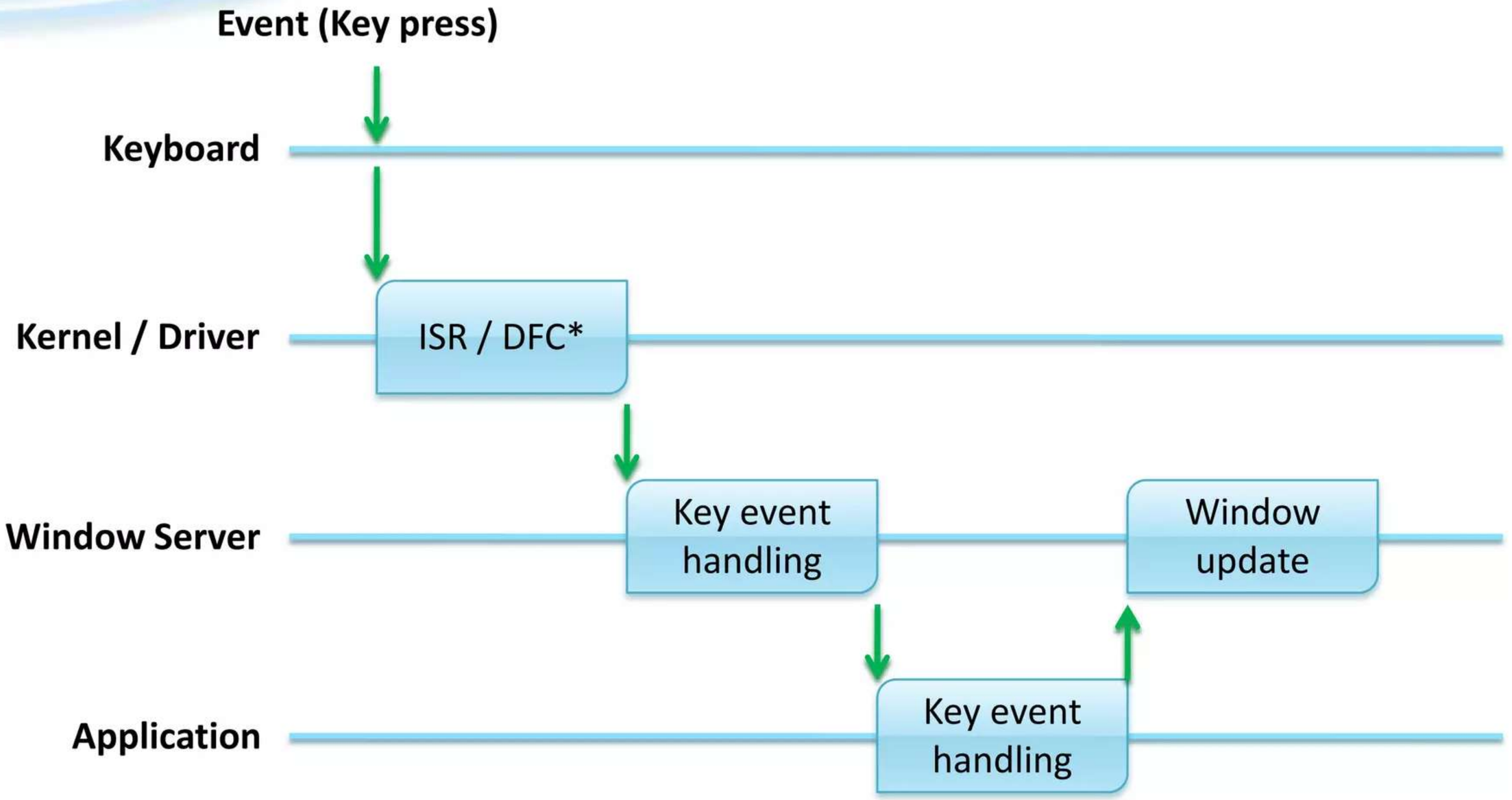
e.g.:

```
RBuf buf;
buf.CreateL(10);
```

e.g.:

```
iSocket.Read( iBuffer, iStatus );
```

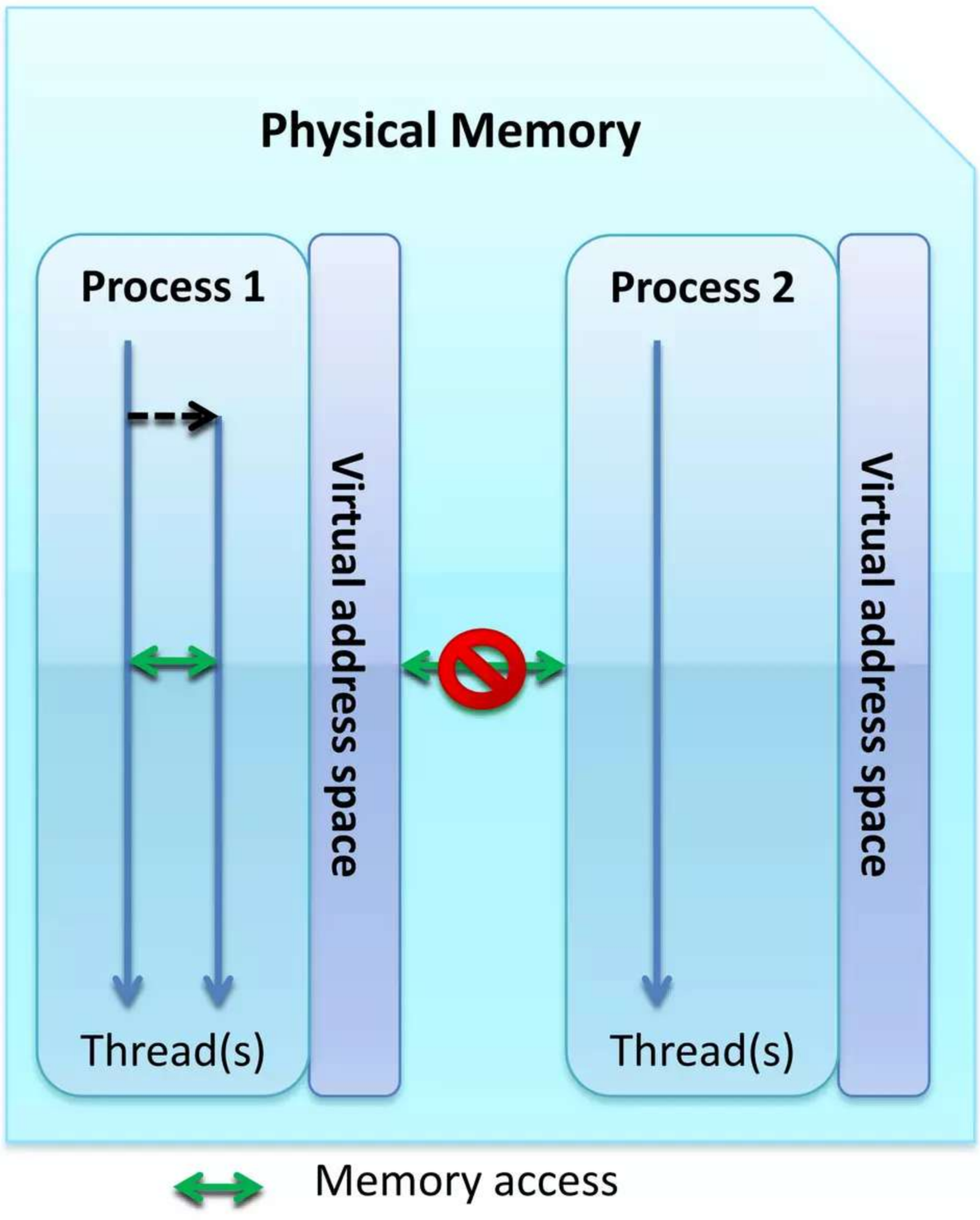
Motivation – Event Handling



ISR = Interrupt Service Handler, DFC = Delayed Function Call

Process

- Contains 1+ threads
- New process: creates 1 primary thread
- **Process:** each has its own virtual memory
- **Thread:** shared memory. Generally a good thing, but negative aspects as well (overwriting data)



Multitasking

- **Pre-emptive:**
  - Threads run as long as they live or until they are interrupted by the scheduler (e.g. because of a thread with higher priority)
  - e.g.: Windows XP, **Symbian OS**
- **Cooperative:**
  - Task runs until it is finished or it gives time to other tasks
  - e.g.: Windows 3.11
- **Non-Preemptive:**
  - Task runs until it is finished and is not interrupted until then (in its own process)
  - e.g.: **Symbian OS** (Active Objects)

Multitasking – Disadvantages

- **Overheads** for a context switch to another process:
  - Runtime overhead (through kernel scheduler)
  - Memory Management Unit
  - Hardware caches
- **Complexity:**
  - Protect shared objects (e.g. mutex, semaphore)
  - Resources (files, …) are owned by only 1 thread (by default) – requires session sharing

Active Objects

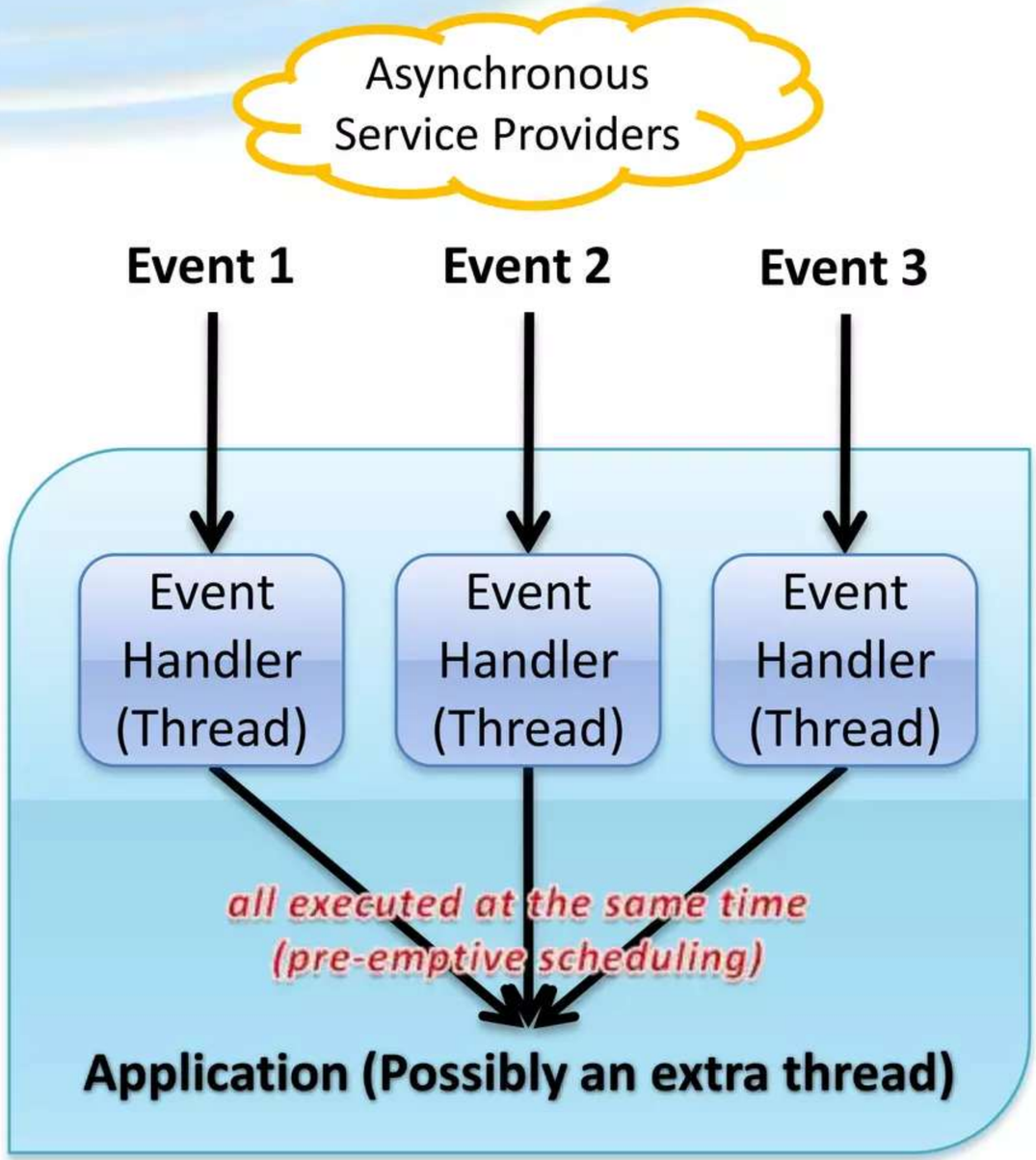
Event Handling

What is an Active Object?

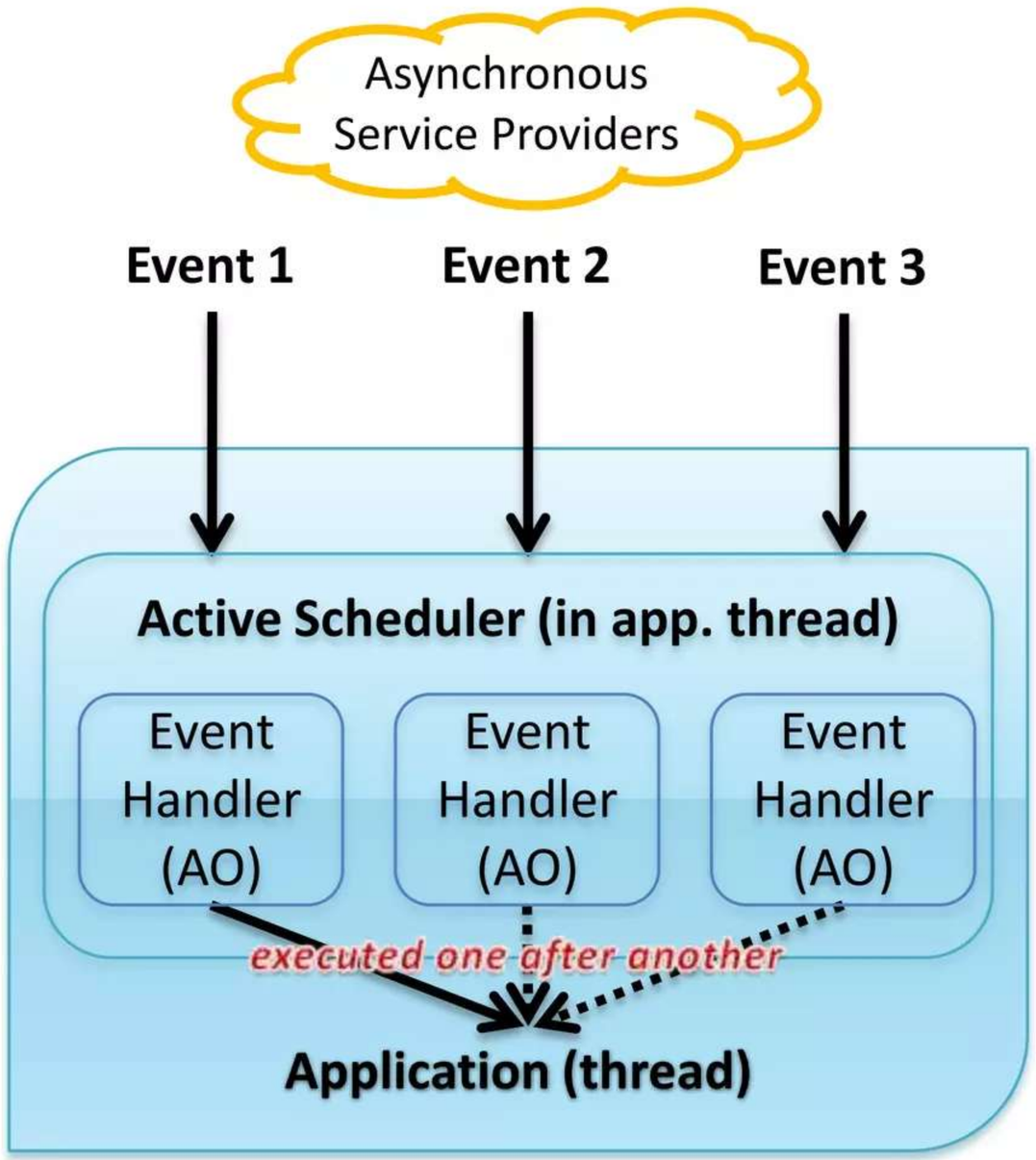
- **Requests asynchronous service**
  - provided by *Asynchronous Service Provider*
- **Receives call-back** when finished / error
  - message handling by *Active Scheduler*

→ *Active Object* ≈ **Listener**!

Event Handling



Traditional Event-Handling



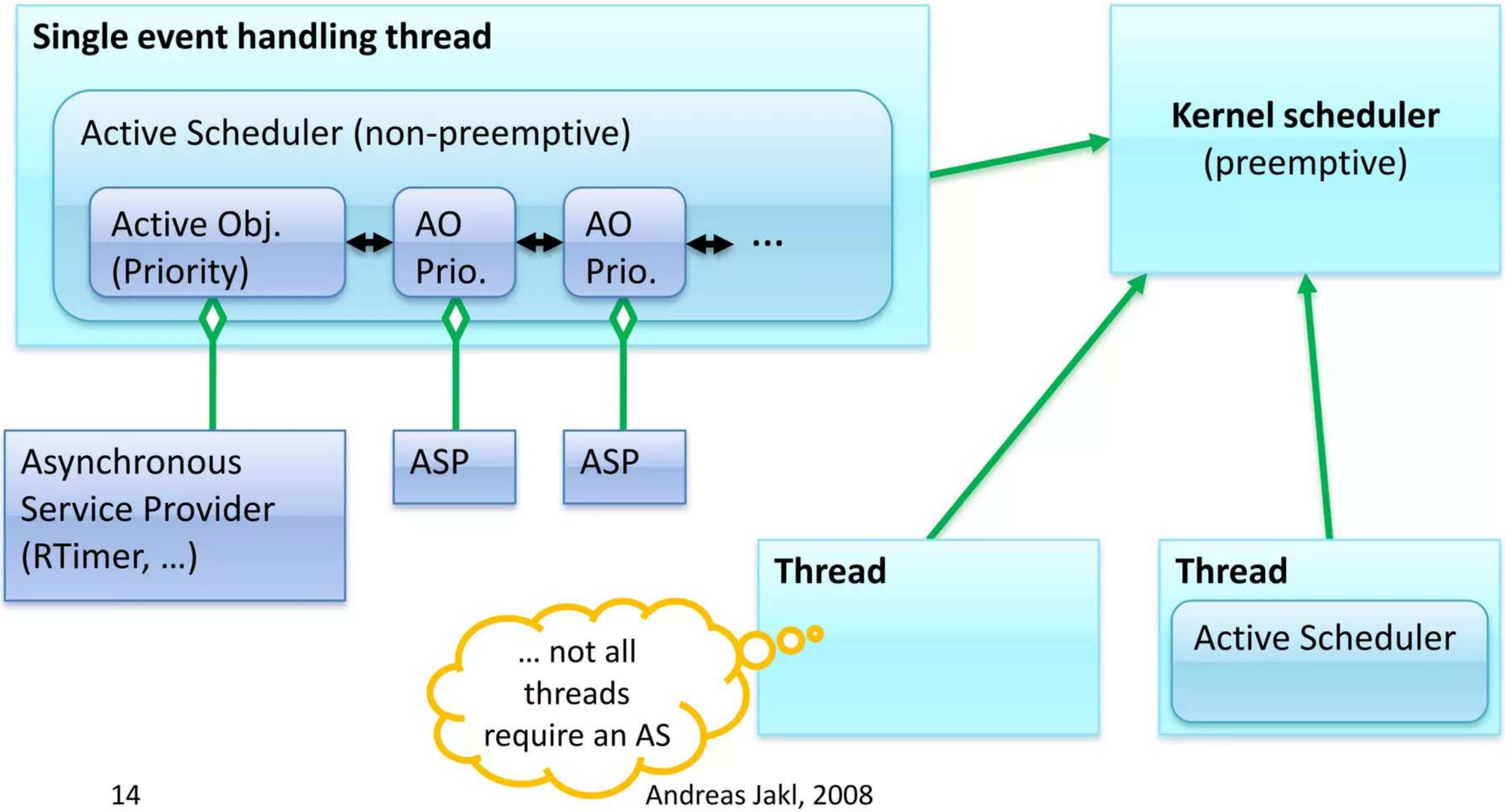
Active Objects
(Symbian OS)

Active Objects

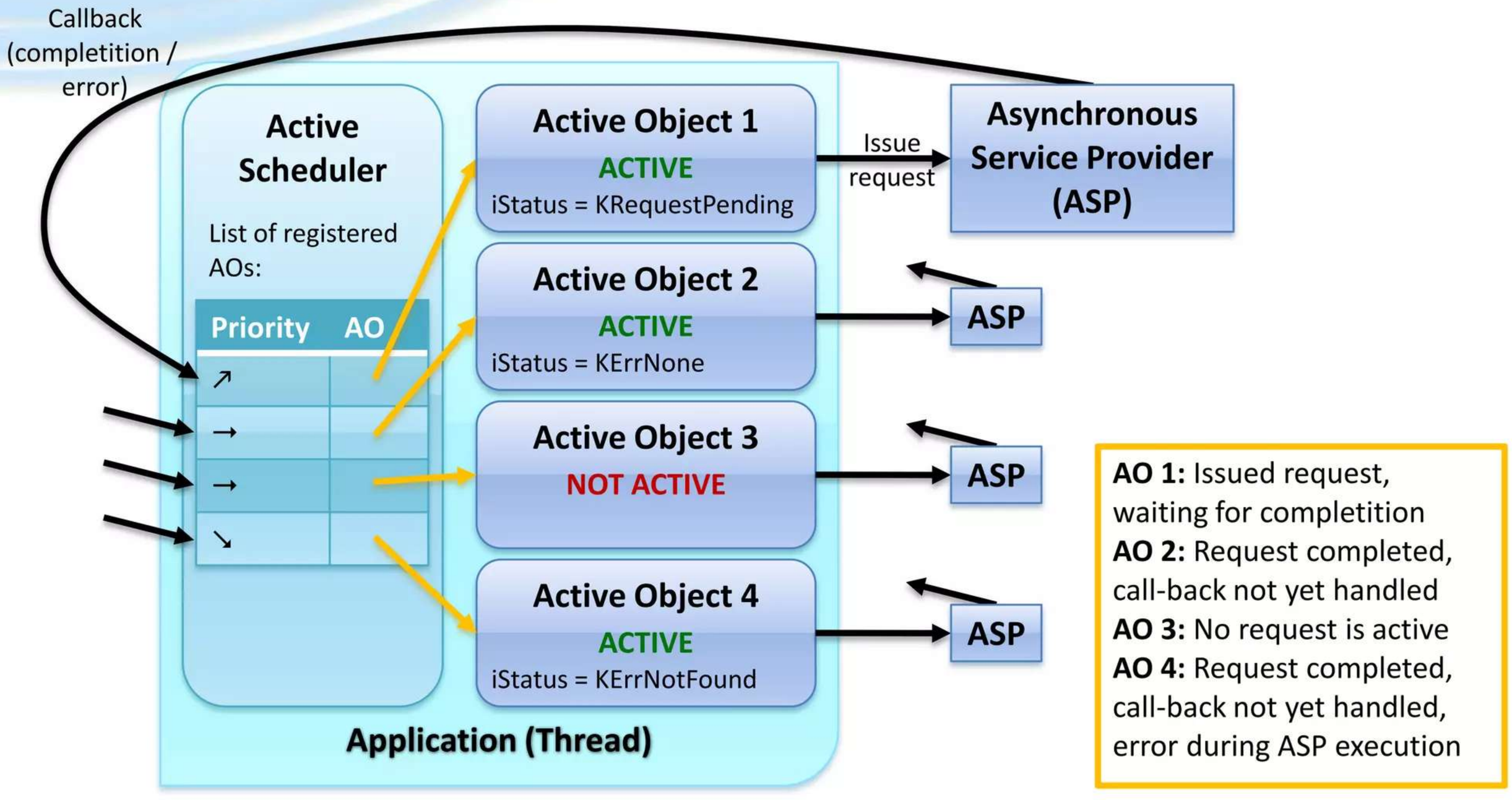
- **Asynchronous Service Provider (ASP)**
  - Executes asynchronous task
  - e.g. timer, socket-related request, take a camera picture, load and prepare a sound file, ...
- **Application-side:**
  - **AOs encapsulate ASP and event handling** (after the request to the ASP has finished)
- Modular separation:
  - **Active Scheduler (AS):** event completion processing for all ASPs
  - **Active Object (AO):** individual event handling

Multiple Applications

- System overview with multiple applications / threads



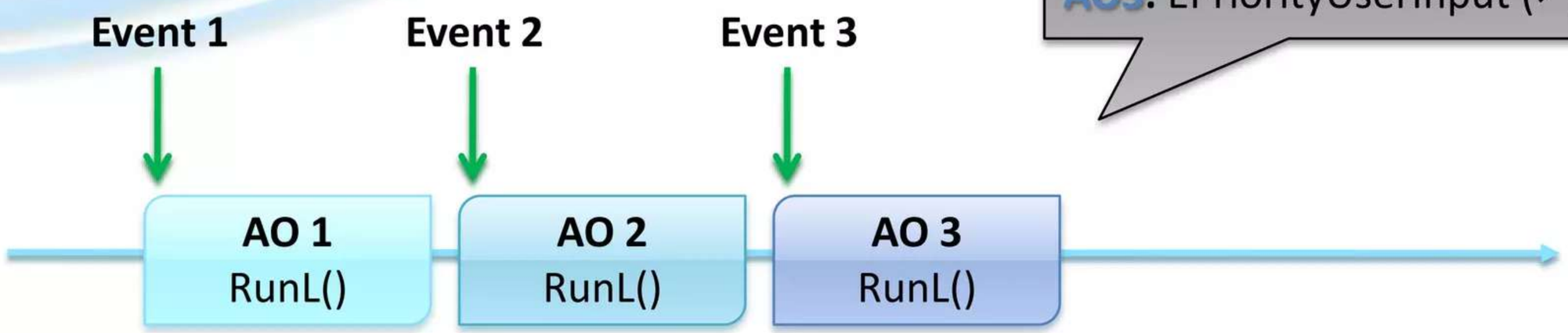
Application Overview



Scheduling

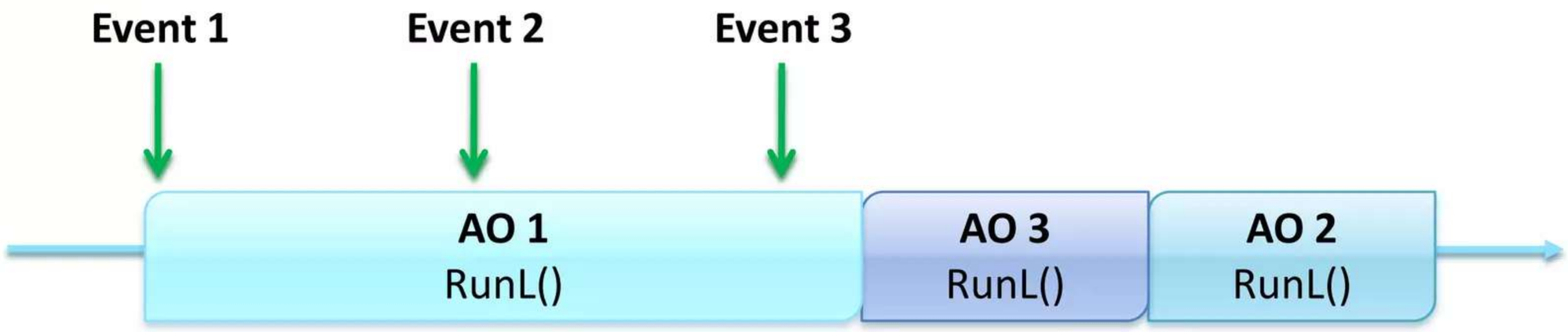
AO1: EPriorityStandard (→)
AO2: EPriorityLow (↘)
AO3: EPriorityUserInput (↗)

1.



2.

3.

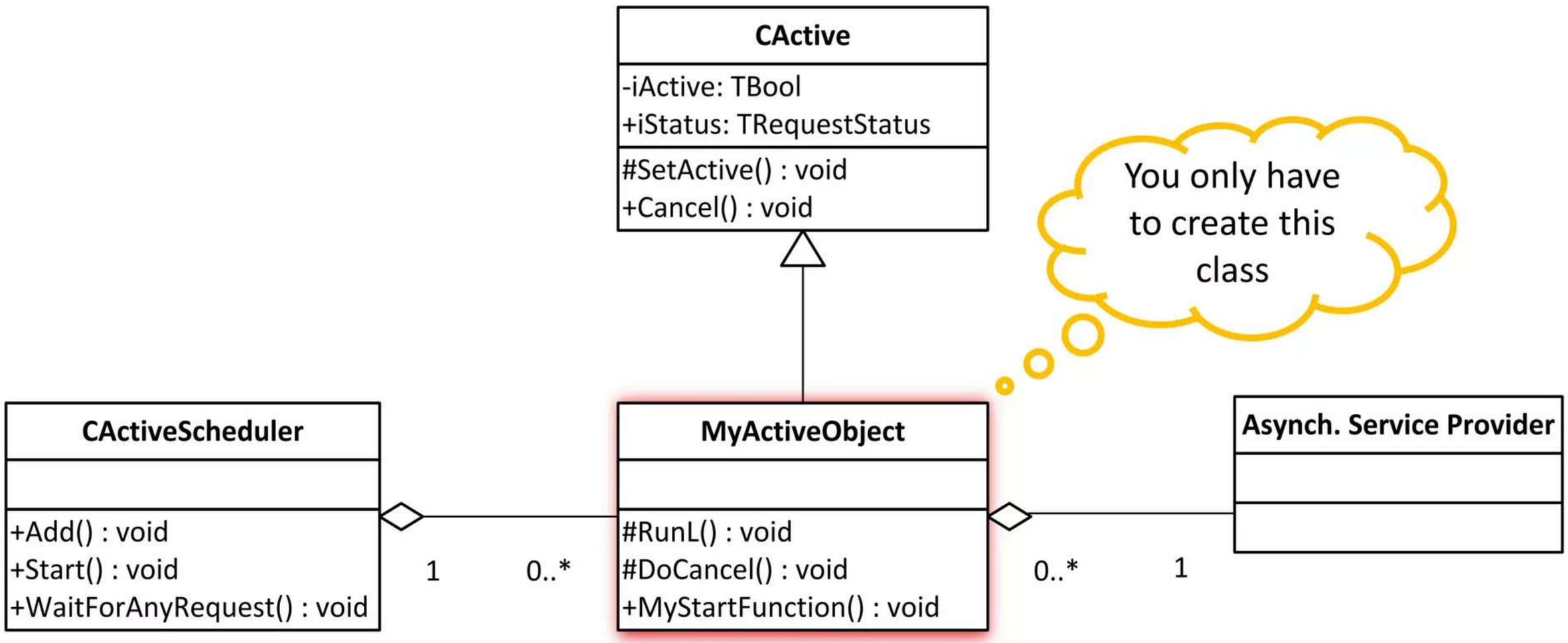


Structure of an AO

Active Object Class

- **Tasks of an Active Object**
  - Request asynchronous service
  - Handle completion event
  - Provide a way to cancel the outstanding request
  - (optional) Error handling
- **To create it**
  - Derive a class from CActive
  - Override two virtual functions: RunL() and DoCancel()

AOs – Classes


CActive
-iActive: TBool
+iStatus: TRequestStatus
#SetActive() : void
+Cancel() : void
You only have to create this class
CActiveScheduler
+Add() : void
+Start() : void
+WaitForAnyRequest() : void
1
0..*
MyActiveObject
#RunL() : void
#DoCancel() : void
+MyStartFunction() : void
0..*
1
Asynch. Service Provider

Example – Overview

- **Timer example**
  - Creates continuous events, writes to RDebug-log
- **Our Active Object: CExampleTimer**
  - Interface to the Asynchronous Service Provider
- **Symbian OS RTimer-object**
  - Asynchronous Service Provider
  - Generates call-back after a specified amount of time
- ***Note:*** *A behaviour like this is actually pre-implemented in Symbian OS (class CTimer)*

CExampleTimer – Definition

```
class CExampleTimer : public CActive {
public:
        ~CExampleTimer();
        // Standard Symbian OS two-phased construction
        // Note: AOs usually created as instance variables -> NewLC() normally not necessary
        static CExampleTimer* NewL();
        // Function called by our own application to initiate regular call-backs
        void After(TTimeIntervalMicroSeconds32& aInterval);
protected:
        CExampleTimer();
        void ConstructL();
protected:      // Inherited from CActive …
        virtual void RunL();                    // Handle the timer event
        virtual void DoCancel();                // Cancel the timer
        virtual TInt RunError(TInt aError); // Leave occurred during RunL()
private:
        RTimer iTimer;                          // Symbian OS Asynchronous Service Provider
        TTimeIntervalMicroSeconds32 iInterval;      // Save interval for regular call-backs
};
```

Create an AO

1. `CActive`-derived class has to call constructor of base class with desired priority
2. Add to the *ActiveSchedulerer*

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

CExampleTimer::ConstructL() {
      // Create the timer object (= Asynch. Service Provider)
      User::LeaveIfError(iTimer.CreateLocal());
}
```

Send out a new Request

1. Test if a **previous request is still active** (only 1 Request / *AO*).
   If yes, either:
   a) Trigger a Panic if this can only happen because of a programming error
   b) Decline the new request (if allowed by the logical structure)
   c) Cancel and discard currently active request, send new request
2. **Send request to ASP**, pass AO's own `iStatus` as `TRequestStatus&`-parameter
   – ASP will set iStatus to KErrNone when the request was finished successfully, otherwise to an error code
3. Call **`SetActive()`** of the `CActive`-Base class
   – … to inform the Active Scheduler that we're waiting for a request

New Request – Example

```
void CExampleTimer::After(TTimeIntervalMicroSeconds32& aInterval)
{
      // Only allow 1 active timer request at a time
      // Here: caller (= app.) has to cancel the previous request himself – would also be
      // be possible to cancel the request here or to ignore the new one
      if (IsActive()) {
            _LIT(KExampleTimerPanic, “CExampleTimer”);
            User::Panic(KExampleTimerPanic, KErrInUse));
      }
      iInterval = aInterval;       // Save the interval for regular callbacks
      // Submit the request to the Asynchronous Service Provider → start the timer!
      iTimer.After(iStatus, aInterval);
      // Mark this object active, so that the Active Scheduler knows we expect a callback
      SetActive();
}
```

Event Handling

- AO has to implement **`RunL()`** – called by ASP when request has been processed
  - What RunL() usually does:
    checks completion code (**`iStatus`**)
  - Depending on `iStatus`:
    - Send out another request
    - Notify other objects in the system / application
  - Can **not** be **pre-empted by other AOs** → make the processing time as short as possible!

Event Handling – Example

```
void CExampleTimer::RunL()
{
	// If an error occurred, deal with the problem in RunError()
	// (not very likely in the case of an RTimer)
	User::LeaveIfError(iStatus.Int());

	// No error: Log the timer completion (or do any processing you like)
	__LIT(KTimerExpired, “Timer Expired\n”);
	RDebug::Print(KTimerExpired);

	// Resubmit a new timer request, to provide continuous call-backs
	iTimer.After(iStatus, iInterval);
	SetActive();
}
```

Cancelling

- AO must be able to **cancel every asynch. process**
  - e.g. when application is closed
- Implement **`DoCancel()`**
  - Call ASP cancellation
  - Do necessary Cleanup
  - Must not cause a leave (no **L**!)
- Application should **only call `Cancel()`** from `CActive`, not `DoCancel()` directly!
  (Cancel() checks if the AO is currently active at all and only calls DoCancel() if necessary)

Cancelling – Example

```
void CExampleTimer::DoCancel()
{
        // CActive::Cancel() checks if the AO is active and only calls DoCancel() if it is.
        // Therefore, we don't have to check if the AO is active before cancelling the ASP.
        iTimer.Cancel();
}
```

Error handling

- If there's a leave in RunL( ) → Active Scheduler calls RunError( ) of your AO
- **Parameter:** Leave code.
  Return: Taken care of leave?
  - **Yes:** return KErrNone
  - **No:** ActiveSchedulerer calls own Error( ) → no context information is available anymore → problematic

Error Handling – Example

```
TInt CExampleTimer::RunError(TInt aError)
{
	// Called by the Active Scheduler if RunL() leaves
	// The parameter aError contains the error code
	_LIT(KErrorLog, “Timer Error %d”);
	// Log the error
	RDebug::Print(KErrorLog, aError);
	// Tell the Active Scheduler that the error has been handled
	return KErrNone;
}
```

Destructor

- Call `Cancel()` in the AO-destructor!
  - Cancel active requests
    - Remember: Cancel() only calls DoCancel() if the AO is active!
  - Release resources of the object
- Deleting AO without cancelling request:
  - causes a „**stray signal**“ (E32USER-CBASE 46)

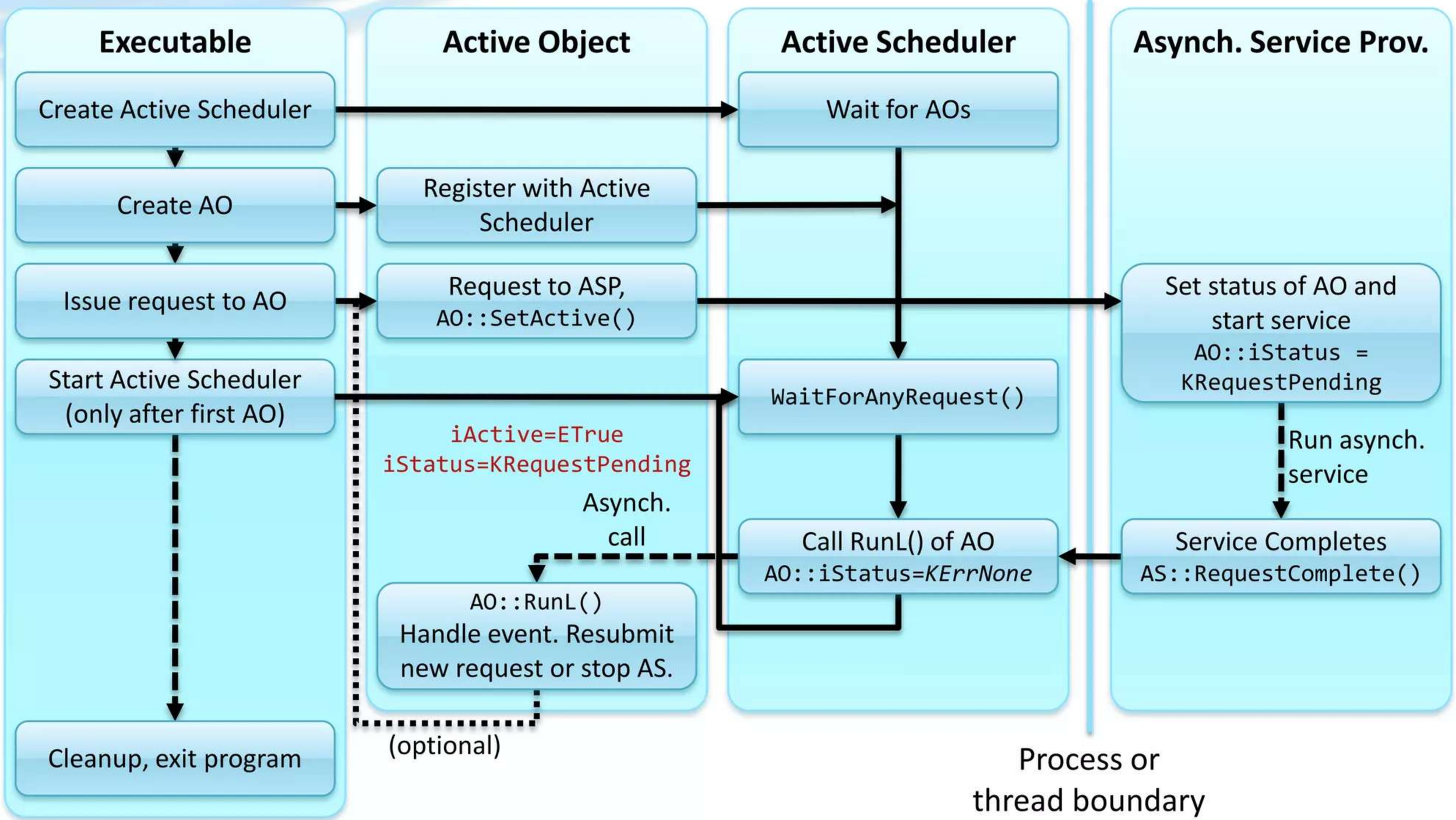
Destructor – Example

```
CExampleTimer::~CExampleTimer()
{
    // Don't call DoCancel() directly! AObject::Cancel() will only call DoCancel()
    // if the AO is active and waits for the ASP to actually cancel the request
    // to avoid stray signals
    Cancel();
    // Do normal cleanup after cancelling the AO!
    iTimer.Close();
}
```

Stray Signals

- Active Scheduler receives completion event, but can't find corresponding Active Object to handle it
- → **"Stray Signal"-panic (E32USER-CBASE 46)**
- **Possible reasons:**
  - **CActiveScheduler::Add()** missing in AO-construction
  - **SetActive()** not called after submitting request
  - More than 1 call-back from ASP (due to programming error or **multiple requests** from the same AO)

AOs – Events and Sequence



Active Objects vs. Threads

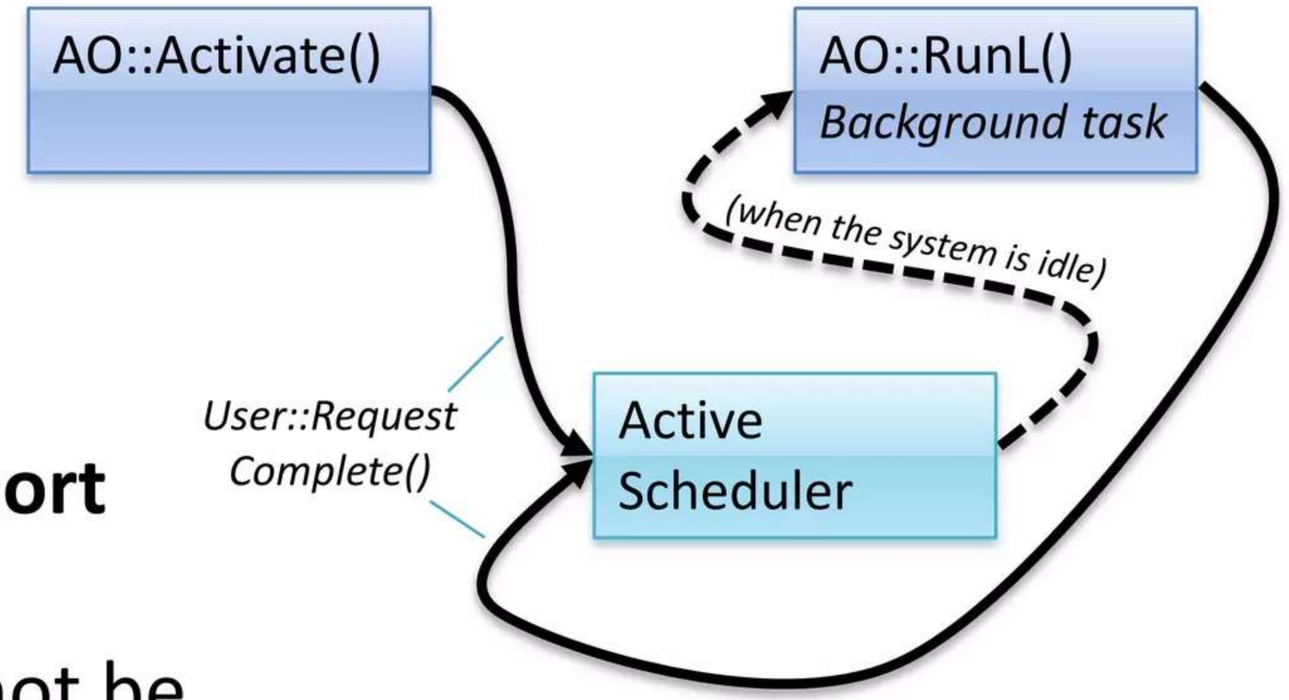
- **Active Objects:**
  - Less runtime overhead
  - Easier to implement: no mutexes, semaphores, ...
  - You have to split up long processes into smaller, individual steps
- **Threads:**
  - Large overhead for a context switch
  - If you need them:
    use RThread → Thread with pre-emptive multitasking

Long running

Background Tasks

Background Task

- **Usual solution:**
  - low-priority thread
- **If you want to use AOs:**
  - Split task into **multiple short increments**
    (→ short RunL() as it cannot be pre-empted)
  - Assign **low priority** (EPriorityIdle)
  - Instead of using a timer,
    the AO **instantly completes** itself



Instant Completition

1. AO tells Active Scheduler that its request has finished
2. AS calls RunL() only when the system is idle due to the low priority of the AO

Call this code when initiating the background processing and after every increment of the RunL()-function:

```
TRequestStatus* status = &iStatus;
// Generates event on itself
User::RequestComplete(status, KErrNone);
// Important to set the AO as active so that the AS can call our RunL()
SetActive();
```

... Symbian's wrapper class CIdle can also do that for you!

ASD-like Question – Easy

- **Which of the following statements about active objects are incorrect?**
  - **A.** SetActive() should be called on an active object by the calling client, after it has called the function which submits a request to an asynchronous service provider.
  - **B.** An active object class should implement RunL() and DoCancel() methods.
  - **C.** The active object framework allows an active object to have multiple outstanding asynchronous requests.
  - **D.** An active object should always derive from CActive.
  - **E.** The active object should implement RunError() if a leave can occur in the RunL() function.

Solution

- **A.** Incorrect. The Active Object sets itself active right after it has submitted the request to the Asynchronous Service Provider (ASP).
- **B.** Correct.
- **C.** Incorrect. Each Active Object (AO) can only have one pending request at the same time. If a second one should be issued, the AO can either Panic, ignore the new request or stop the previous request first.
- **D.** Correct.
- **E.** Correct.

ASD-like Question – Medium

- **Which of the following statements regarding an active object used for a long-running background task are correct?**
  - **A.** It should have a high priority to ensure that it gets a chance to run to completion.
  - **B.** It should maintain an internal state machine to track the different stages of the task.
  - **C.** It should split the long-running task into small processing “slices”, with each execution of RunL() performing one such slice only.
  - **D.** It should contain no accesses to data that would cause problems if the RunL() was preempted.
  - **E.** It should use a timer to generate events that invoke RunL().

Solution

- **A.** Incorrect. Long running background tasks should have a low priority, so that high priority events (e.g. user input) can be handled instantly.
- **B.** Correct.
- **C.** Correct.
- **D.** Incorrect. Active Objects are not pre-empted when they are in RunL(), therefore the RunL()-method should be rather short.
- **E.** Incorrect. Background-tasks instantly set themselves to be complete, so that the Active Scheduler calls them again right away if no higher priority AO is waiting.

ASD-like Question – Hard

- **The active scheduler causes a "stray signal" panic (E32USER-CBASE 46) when the scheduler receives a request completion and cannot find an active object associated with it.**
  Which of the following errors in the implementation of an active object class could cause this problem?
  - **A.** Not calling SetActive() after submitting a request to an asynchronous service provider.
  - **B.** Not checking whether there is an outstanding request in the implementation of DoCancel().
  - **C.** Forgetting to add an active object to the active scheduler.
  - **D.** Implementing RunL() so that it performs a lot of time-consuming processing before returning.
  - **E.** Setting iStatus to KRequestPending before submitting it in a request to an asynchronous service provider.

Solution

- **A.** Yes. The Active Scheduler can only send completion events to Active Objects that are active and therefore waiting for completion events.
- **B.** No. DoCancel() is only called by the Active Object base class after Cancel() checks that a request is currently active
- **C.** Yes. The Active Scheduler can't send the event to the Active Object if it doesn't know about it.
- **D.** No. This is of course bad practice and makes your app unresponsive, but doesn't cause a panic.
- **E.** No. Automatically done by the Asynchronous Service Provider. Therefore, it doesn't matter to what you set it before submitting the request.

That’s it!

Thanks for your attention