symbian Academy

System Structure

Part One

System Structure

This Lecture Examines

- DLLs
- Memory management
- Threads and processes

System Structure

A brief high-level overview of Symbian OS:

- It is a multi-tasking operating system based on open standards for advanced mobile phones

The phones

- Have a sophisticated graphical user interface (GUI) and a number of built-in applications which use it
  - For example, messaging and calendar

Is said to be an "open" platform because, in addition to the applications built in by the manufacturer

- A user may install others such as games, enterprise applications (for example push e-mail) or utilities

System Structure

Symbian OS is licensed to the world's leading handset manufacturers

- Arima, BenQ, Fujitsu, Lenovo, LG Electronics, Motorola, Mitsubishi, Nokia, Panasonic, Samsung, Sharp and Sony Ericsson

Symbian OS has a flexible architecture

- Allowing different user interfaces to run on top of the core operating system

Symbian OS UIs include

- Nokia's S60 and Series 80 platforms, NTT DoCoMo's FOMA user interface and UIQ

System Structure

EKA1 and EKA2

- Refer to different versions of the Symbian OS kernel
- The EKA stands for "EPOC Kernel Architecture"

  (Symbian OS was previously known as "EPOC", and earlier still "EPOC32")
- EKA1 is the 32-bit kernel released originally in the Psion Series 5 in 1997
- EKA2 is discussed in the next slide

System Structure

EKA2

- Was first introduced in Symbian OS version 8.0b
- But first shipped in a phone product until version 8.1b
- Is found in the Japanese MOAP 2.0 FOMA 902i series phones
- It is the second iteration of Symbian's 32-bit kernel
- Very different internally to EKA1
- Offers hard real-time guarantees to kernel and user-mode threads

DLLs in Symbian OS

- Know and understand the characteristics of polymorphic interface and shared library (static) DLLs
- Know that `UID2` values are used to distinguish between static and polymorphic DLLs, and between plug-in types
- For a shared library, understand which functions must be exported if other binary components are to be able to access them
- Know that Symbian OS does not allow library lookup by name but only by ordinal

Shared Library and Polymorphic Interface DLLs

Dynamic link libraries

- DLLs are libraries of compiled C++ code
- That may be loaded into a running process
- In the context of an existing thread

There are two main types of DLL

- Shared library (static-interface) DLLs
- Polymorphic interface (plug-in) DLLs

Shared Library DLLs

A shared library DLL

- Implements library code that may be used by other libraries or EXEs
- The filename extension of a shared library is .dll

Examples of this type are

- The user library **`EUser.dll`**
- The file system library **`EFsrv.dll`**

A shared library

- Exports API functions according to a module definition (**`.def`**) file
- It may have any number of exported functions
- Each is an entry point into the DLL

Shared Library DLLs

A shared library releases

- A header file (`.h`) for other components to compile against
- An import library (`.lib`) to link against in order to resolve the exported functions

When executable code that uses the library runs

- The Symbian OS loader loads any shared library DLLs that it links to, and any further DLLs the shared library DLLs require
- This is done recursively until all shared code needed by the executable is loaded

Polymorphic Interface DLLs

A polymorphic interface DLL

- Implements an abstract interface which is defined separately

  For example by a framework

It may have a **`.dll`** filename extension

- But it often uses a different extension to identify the nature of the DLL further

For example

- **`.fsy`** for a file system plug-in
- **`.prt`** for a protocol module plug-in
- File systems and Sockets are discussed later

Polymorphic Interface DLLs

Polymorphic Interface DLLs are used as plug-ins

They have a single entry-point "gate" or "factory" function

- Which instantiates the concrete class that implements the interface

They are used

- To provide a range of different implementations (plug-ins) of a single consistent interface

They are loaded dynamically

- Typically by a framework

ECOM Plug-ins

From Symbian OS v7.0 onward

- The most common type of plug-ins are ECOM plug-ins

ECOM is a generic framework for specifying interfaces

- And for finding and loading those plug-ins which implement them

Many Symbian OS frameworks require their plug-ins to be written as ECOM plug-ins

- Rather than as a “proprietary” type of framework which loads polymorphic interface DLL plug-ins
- For example, the recognizer framework

ECOM Plug-ins

Using ECOM allows each framework to delegate the finding and loading of suitable plug-ins to ECOM

- Rather than performing that task itself

Thus making easier to design and implement new services or features

- ECOM provide a consistent design pattern

UIDs used by DLLs

UIDs are used to identify a file type

- For running executable code (including DLLs)
- And for associating data files with the appropriate application

A UID is a 32-bit globally unique identifier value

Symbian OS uses a combination of up to three UIDs to uniquely identify a binary executable

- The three UID values used by DLLs are as follows ...

UID1

UID1 is a system-level identifier

- Distinguishes between EXEs and DLLs
- This value is never stated explicitly
- It is determined by the Symbian build tools from the **`targettype`** specified in the MMP file

For shared libraries

- The **`targettype`** specified should be DLL
- **`UID1 = KDynamicLibraryUid = 0x10000079`**

UID1 Continued

For polymorphic ECOM plug-in DLLs

- The **`targettype`** is **`PLUGIN`**
- or **`ECOMIIC`** for versions of Symbian OS earlier than v9.0

Other polymorphic non-ECOM plug-in DLL target types

- **`FSY`** (file system plug-in)
- **`PRT`** (protocol module plug-ins).
- The **`targettype`** keyword and the build tools are discussed in later lectures

UID2

UID2 distinguishes between shared library DLLs and polymorphic interface DLLs

- Shared libraries are always **`KSharedLibraryUid`** (**`0x1000008d`**)

Polymorphic interface DLLs have UID2 values specific to their type

- For example the socket server protocol module UID2 value is **`0x1000004A`**

UID3

UID3 is used to identify a component uniquely

Symbian manages UID allocation through a central database

- Ensuring the UID is a genuinely unique value

Developers must be registered with Symbian Signed to request UIDs

- More on Symbian Signed later

EXEs

The UID1 value is set by the **`targettype EXE`** statement to (**`KExecutableImageUid=0x1000007a`**)

UID2 is not relevant for an EXE

- It can be left unspecified
- Or set explicitly to **`KNullUid`** (=0)

UID3

- on Symbian OS v9 and beyond UID3 should usually be set to a unique value to act as the secure identifier for the binary
- Pre-Symbian OS v9 it can be left unspecified

symbian Academy

Exporting Functions from a DLL

A shared library DLL provides access to its APIs by exporting its functions

- Used by another DLL or by EXE code compiled into a separate binary component
- Exporting makes the functions "public" to other modules by creating a `.lib` file
- Libs contain the export table to be linked against by the calling code

Exporting Functions from a DLL

Functions to be exported

- Should be marked in the class definition in the header file with the macro **`IMPORT_C`**

The client code will include the header file

- effectively "importing" each function into their code module
- When they call it

The corresponding function implementation

- Should be prefixed with the **`EXPORT_C`** macro in the .cpp file which implements it

Exporting Functions from a DLL

Use of **IMPORT_C** and **EXPORT_C**:

```
class CMyExample : public CSomeBase
    {
public:
    IMPORT_C static CMyExample* NewL();
public:
    IMPORT_C void Foo();
    ...
    };

EXPORT_C CMyExample* CMyExample::NewL()
    {...}
EXPORT_C void CMyExample::Foo()
    {...}
```

Exporting Functions from a DLL

The rules as to which functions should be exported are as follows:

Inline functions must never be exported because there is no need to do so

This is why:

- The **`IMPORT_C`** and **`EXPORT_C`** macros add functions to the export table to make them accessible to components linking against the library
- But the code of an inline function is already accessible to callers because it is declared within the header file
- So the compiler interprets the **`inline`** directive by adding the code directly into the client code wherever it calls it. There is no need to export it.

Exporting Functions from a DLL

Only functions that are to be used outside a DLL should be exported by using of **`IMPORT_C`** and **`EXPORT_C`**

If the function is private to the class

- It can never be accessed by client code

Exporting it adds it to the export table in the module definition file (**`.def`**) unnecessarily

Exporting Functions from a DLL

All virtual functions should be exported

- Whether public, protected or private
- Since they may be re-implemented by a derived class in another code module

Any class which has virtual functions

- Must also export a constructor
- Even if it is empty

So that the virtual function table

- Can be correctly generated by access to the base-class constructor

Lookup by Ordinal and by Name

The size of DLL program code is optimized

- To save ROM and RAM space

In most operating systems to load a dynamic library the entry points of a DLL can either be:

- Identified by string-matching their name - lookup by name
- Or by the order in which they are exported in the module definition file - lookup by ordinal

Symbian OS does not offer lookup by name

- As it adds an overhead to the size of the DLL
- Storing the names of all the functions exported from the library is wasteful of limited ROM and RAM space

Lookup by Ordinal and by Name

Symbian OS only uses link by ordinal

- This has significant implications for binary compatibility
- Ordinals must not be changed between one release of a DLL and another

For example

- Code which links against a library and uses an exported function with a specific ordinal number in an early version of the library
- Will not be able to call that function in a newer version of the library if the ordinal number is changed

Binary compatibility is discussed further in a later lecture

Note

The one type of virtual function which should NOT be exported from a DLL is a pure virtual function

- As there is generally no implementation code for a pure virtual function
- So there is no code to export

Writable Static Data

- Recognize that writable static data is not allowed in DLLs on EKA1 and discouraged on EKA2
- Know the basic porting strategies for removing writable static data from DLLs

Support for Writable Static Data

Symbian OS supports global writable static data in EXEs

- On all versions and handsets

In versions of Symbian OS which contain EKA1 (Symbian OS versions 8.1a, 8.0a or earlier)

- Writable static data cannot be used in DLLs
- This is because DLLs have separate areas for program code and read-only data

  But do not have an area for writable data

Versions of Symbian OS which Support Writable Static Data in DLLs

Versions of Symbian OS which contain EKA2 (Symbian OS versions 8.0b, 8.1b, 9.0 and beyond)

- Now support the use of writable static data in DLLs

But it is still not recommended

- As it is expensive in terms of memory usage
- And has limited support in the Symbian OS Emulator

Symbian recommends that it only be used as a last resort

- e.g. when porting code written for other platforms which uses writable static data heavily

Writable Static Data in GUI applications

On EKA1

- All GUI applications were built as DLLs
- No application code could use writable static or global data

On EKA2

- Applications are now built as EXEs, so this is no longer an issue
- Modifiable global or static data has always been allowed in EXEs

Versions of Symbian OS which Support Writable Static Data in DLLs

Symbian OS platform version	Writable static data in DLLs built for hardware	Application binary type
v6.1 — v8.0a (inclusive), v8.1a (EKA1)	Not supported on hardware builds (compilation will fail)	DLL — no writable static data allowed
v8.0b, v8.1b, v9.0 and beyond (EKA2(	Supported but not recommended — limited emulator support and inefficient in terms of memory usage	EXE — writable static data can be used

How to Enable Writable Static Data in DLLs

In order to enable global writable static data on EKA2

- The **`EPOCALLOWDLLDATA`** keyword must be added to the MMP file of a DLL
- Where this is not used and on EKA1 versions of the Symbian OS
- The tool chain will return an error when the DLL code is built for the phone hardware

Workarounds to Avoid Writable Static Data in DLLs

1. Thread-local storage

- One workaround used to replace writable static data is called thread-local storage (TLS)

This can be accessed through

- Class **`Dll`** on pre-8.1b versions of Symbian OS
- Class **`UserSvr`** for version 8.1b and version 9.0.

Thread-local storage is a 32-bit pointer

- Specific to each thread that can be used to refer to an object which simulates global writable static data
- All the global data must be grouped within this single object
- And allocated on the heap when the thread is created

Thread-Local Storage

Functions `Dll::SetTls()` or `UserSvr::DllSetTls()`

- Are used to save the pointer to the object
- To the thread-local storage pointer

Functions `Dll::Tls()` or `UserSvr::DllTls()`

- Are used to access the global data

On destruction of the thread

- The data is destroyed too

Workarounds to Avoid Writable Static Data in DLLs

2. Client–server framework

- Symbian OS supports writable global static data in EXEs

A common porting strategy is to wrap the code in a Symbian server

- Which is an EXE
- Exposing its API as a client interface

Workarounds to Avoid Writable Static Data in DLLs

3. Embed global variables into classes

- With small amounts of code it may be possible to move most global data inside classes
- The data can then be passed as function parameters between objects and functions

Writable Static Data Defined

Global writable static data is any per-process modifiable variable

- Which exists for the lifetime of the process

In practice this means any globally scoped data declared outside of

- A function
- A struct or class
- Function-scoped static variables

Writable Static Data Defined

The only global data that can be used within DLLs is

- constant global data of the built-in types
- Or of a class with no constructor

So these definitions are acceptable:

```
static const TUid KUidFooDll = { 0xF000C001 };

static const TInt KMinimumPasswordLength = 6;
```

Writable Static Data Defined

The following definitions cannot be used because they have non-trivial class constructors

- That is, the objects must be constructed at run-time

```
static const TPoint KGlobalStartingPoint(50, 50);
static const TChar KExclamation('!');
// The following literal type is deprecated
static const TPtrC KDefaultInput =_L("");
```

Writable Static Data Defined

The memory for the object is pre-allocated in code but it does not actually become initialized and constant

- until after the constructor has run

Thus at build time, each constitutes a non-constant global object

- Causes the build to fail for phone hardware, unless the **`EPOCALLOWDLLDATA`** keyword has been added to the MMP file of the DLL

Writable Static Data Defined

The following object is also non-constant

- Although the data pointed to by ptr is constant
- The pointer itself is not constant:

```
// Writable static data!

static const TText* ptr = (const TText*)"data";
```

- This can be corrected by making the pointer constant

```
static const TText* const ptr = (const TText*)"data";
```

Note

On EKA1

- The emulator can use the underlying Windows DLL mechanism to provide per-process DLL data
- If non-constant global data is used inadvertently - it will go undetected in emulator builds
- It will only fail when the PETRAN tool encounters it in the hardware platform build

Executables in ROM and RAM

- Recognize the correctness of basic statements about Symbian OS execution of DLLs and EXEs in ROM and RAM

EXEs in ROM and RAM

On target hardware

- Executable code can either be built onto the phone in read-only memory (ROM) when the phone is in the factory
- Or can be later installed on the phone either into the phone's internal memory or onto removable storage media such as a memory stick or MMC

ROM-based EXEs

- Can be thought of as executing directly in place from the ROM
- This means that program code and read-only data (such as literal descriptors) are read directly from the ROM
- The component is only allocated a separate data area in RAM for its read/write data.

EXEs in ROM and RAM

If an EXE is installed (rather than built into the ROM)

- It executes entirely from RAM
- It has an area allocated for program code and read-only static data
- A separate area for read/write static data

If a second copy of the EXE is launched

- The read-only area is shared
- A new area of read/write data is allocated.

DLLs in ROM and RAM

DLLs in ROM

- Are not loaded into memory
- Execute in place in ROM at their fixed address

DLLs running from RAM

- Are loaded at a particular address
- The address is determined only at load time

Reference counting is used

- Allowing the DLLs to be unloaded only when they are no longer being used by any component

DLLs in ROM and RAM

Loading a DLL from RAM

- Is different from simply storing it on the internal (RAM) drive

Symbian OS

- Copies it into the area of RAM reserved for program code
- Preparing it for execution by fixing up the relocation information

DLLs in ROM and RAM

DLLs that execute from ROM are fixed at an address

- Thus do not need to be relocated

To compact the DLL

- In order to occupy less ROM space
- Symbian OS tools strip the relocation information out when a ROM is built

The lack of relocation information means that a DLL <u>cannot be copied from the ROM</u>

- then stored and executed from RAM

DLLs in ROM and RAM

For both types of DLL (shared library and polymorphic interface plug-in)

- The code section is shared

If multiple threads or processes use a DLL simultaneously

- The same copy of program code is accessed
- At the same location in memory

Subsequently loaded processes or libraries that wish to use the DLL

- Are fixed up by the DLL loader to use the same copy

Threads and Processes

- Recognize the correctness of basic statements about threads and processes on Symbian OS
- Recognize the role and the characteristics of the synchronization primitives **`RMutex`**, **`RCriticalSection`** and **`RSemaphore`**

Threads

Threads

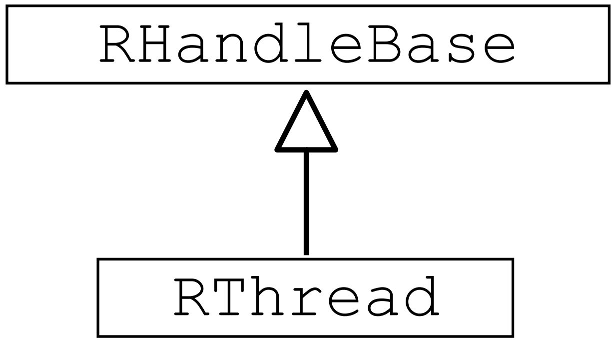
- Are the basic unit of execution
- Form the basis of multitasking - allowing multiple sequences of code to execute simultaneously (or appear to do so)

It is possible to create multiple threads in a Symbian OS application for parallel execution

But in many cases

- It is more appropriate to use active objects
- Since these are optimized for event-driven multi-tasking on Symbian OS

Threads



The class used to manipulate threads is **`RThread`**

- An object of type **`RThread`** represents a handle to a thread
- The thread itself is a kernel object

Threads

The base class of **`RThread`** is **`RHandleBase`**

- Which encapsulates the behavior of a generic handle
- **`RHandleBase`** used as a base class throughout Symbian OS
- To identify a handle to another object
- Often a kernel object

Class **`RThread`** defines several functions for thread creation

Threads are not contained in separate executable files

- But execute within a parent process executable
- Each thread has an independent execution stream

The **`RThread::Create()`** Function

This is one **`RThread::Create()`** method (there are a number of overloads):

```
TInt Create(const TDesC &aName,
            TThreadFunction aFunction,
            TInt aStackSize,
            TInt aHeapMinSize,
            TInt aHeapMaxSize,
            TAny *aPtr,
            TOwnerType aType=EOwnerProcess)
```

Each thread-creation function

- Takes a descriptor representing a unique name for the new thread
- A pointer to a function in which thread execution starts
- A pointer to data to be passed to that function
- A value for the stack size of the thread, which defaults to 8 KB.

Thread Creation

A thread is created

- In the suspended state
- Its execution started by a call to `RThread::Resume()`

The `Create()` function

- Is overloaded to offer various options associated with the thread heap

Such as

- Its maximum and minimum size
- Whether it shares the creating thread's heap or uses a specific heap within the process in which it runs

Thread Heaps

By default, each Symbian OS thread

- Has its own independent heap as well as its own stack
- The size of the stack is limited to the size set in **`RThread::Create()`**

The heap can grow from its minimum size up to a maximum size

When the thread has its own heap

- The stack and the heap are located in the same chunk of memory

Thread Identification

When the thread is created the system assigns it a unique thread identity

- Returned by the `Id()` function of `RThread` as a `TThreadId` object

If the `TThreadId` value of an existing thread is known

- It can be passed to `RThread::Open()`
- To open a handle to that thread

Alternatively

- The unique name of a thread can be passed to open a handle to it

Thread Scheduling

Threads are pre-emptively scheduled

- The currently running thread is the highest-priority thread ready to run

If there are two or more threads with equal priority

- They are time-sliced on a round-robin basis

The priority of a thread is a number

- The higher the value - the higher the priority

A running thread can be removed

- By a call to **`Suspend()`** on the thread handle
- Can be scheduled to run again by another call to **`Resume()`**

Thread Termination

A thread can be ended permanently

- By a call to `Kill(TInt aReason)` or `Terminate(TInt aReason)`
- `aReason` represents the exit reason
- These methods should be used to stop a thread normally
- For stopping the thread to highlight a programming error `Panic()` is used

On EKA1

- A thread must call `SetProtected()`
- To prevent other process threads from acquiring a handle to it
- And killing it by making a call to `Suspend()`, `Panic()`, `Kill()` or `Terminate()`

Thread Security

On EKA2 the security model ensures

- The thread is always protected from threads running in other processes
- The redundant `SetProtected()` method has been removed
- A thread cannot stop another thread in a different process

The functions

- `Suspend()`, `Terminate()`, `Kill()` or `Panic()`
- Are still retained in EKA2
- A thread can still use these functions on itself
- Or other threads in the same process

  but not on threads in a different process

It is also still possible

- For a server to panic a misbehaving client thread
- By calling `RMessagePtr2::Panic()`

Thread Termination

If the main thread in a process

- Is ended by any of the termination methods
- **`Suspend()`**, **`Terminate()`**, **`Kill()`** or **`Panic()`**
- The process also terminates

If a secondary thread

- That is created by a call to **`RThread::Create()`** from with in the process
- The thread terminates
- The process itself does not stop running

Thread Death Notification

To receive notification when a thread dies

- Submit a request for notification of thread termination
- By a call to `RThread::Logon(TRequestStatus &aStatus)`
- The `TRequestStatus` is a completion semaphore

The request completes when the thread terminates

- `aStatus` contains the value with which the thread ended

If the notification request was cancelled

- By a call to `RThread::LogonCancel()`
- `aStatus` will contain `KErrCancel`

Thread Termination

The thread handle class also provides functions to give full details of the associated thread's end state

`TExitType RThread::ExitType()`

- Allows the caller to distinguish between normal termination and a panic

`TInt RThread::ExitReason()`

- Gets the specific reason associated with the end of this thread

`TExitCategoryName RThread::ExitCategory()`

- Gets the name of the category associated with the end of the thread

Thread Notification

A thread rendezvous request can also be created

- To allow correct order synchronization e.g. data manipulation
- By calling the asynchronous **`RThread::Rendezvous()`**

The request completes in any of the following ways:

- When the thread next calls **`RThread::Rendezvous(TInt aReason)`**
- If the outstanding request is cancelled by a call to **`RThread::RendezvousCancel()`**
- If the thread exits or panics

Kernel Objects for Synchronization

Besides the use of **`RThread::Rendezvous()`**, Symbian OS provides several classes representing kernel objects for thread synchronization

- A semaphore
- A mutex
- A critical section

Semaphores

A semaphore

- Can be used either for sending a signal from one thread to another
- Or for protecting a shared resource from being accessed by multiple threads at the same time

A semaphore is created and accessed

- with a handle class called **`RSemaphore`**

A global semaphore

- Can be created, opened and used by any process in the system

A local semaphore

- Can be restricted to all threads within <u>a single process</u>

Semaphore

Semaphores can be used

- To limit concurrent access to a shared resource
- Either to a single thread at a time
- Or multiple accesses up to a specified limit

Mutexes

A mutex

- Is used to protect a shared resource
- So that it can only be accessed by one thread at a time
- The **`RMutex`** class is used to create and access global and local mutexes

Critical Sections

A critical section

- Is a region of code that should not be entered simultaneously by multiple threads

An example is code that manipulates global static data

- Since it could cause problems if multiple threads change the data simultaneously

The **`RCriticalSection`** class

The **`RCriticalSection`** class

- Allows only one thread within the process into the controlled section
- Forces other threads attempting to gain access to that critical section to wait until the first thread has exited from the critical section

`RCriticalSection` objects

- Are always local to a process

A critical section cannot be used to control access to a resource shared by threads across different processes

- A mutex or semaphore should be used instead

Processes

Note

A note on User-side or User-mode operations

- User-side operations dealt with by **`EUser.dll`**
- Which calls system or kernel functions for the user-side component
- Kernel functions sometimes referred privileged mode

For mode in-depth information on the Symbian OS kernel and memory management please see:

- Smartphone Operating System Concepts with Symbian OS
- By Michael J. Jipping
- ISBN 978-0-470-03449-1

Processes

A Symbian OS process

- Is an executable that has its own data area, stack and heap
- By default a process is given 8 KB of stack and 1 MB of heap
- Sometimes referred to as a unit of protection

Processes

Many processes can be active on Symbian OS at once

- Including multiple instances of the same process
- Processes have private address spaces
- A user-side process cannot directly access memory belonging to another user-side process

By default

- A process contains a single execution thread - the main thread
- Additional threads can be created as described above

Processes

A context switch occurs when switching from one thread to another

- Context switches occur whenever a thread is scheduled to run and becomes active

Switching between threads

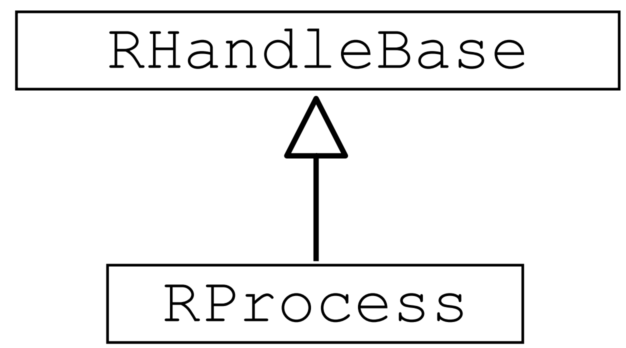
- in different processes is more "expensive" than switching between threads within the same process

A process context switch

- Requires that the data areas of the two processes be remapped by the memory management unit (MMU).

Processes

The class used to manipulate processes is **`RProcess`**



The **`RProcess::Create()`** function

- Can be used to start a new named process

The **`RProcess::Open()`** function

- Can be used to open a handle to a process
- Identified by name or process identity (**`TProcessId`**)

symbian Academy

Processes

There are assorted functions to stop the process

- Similar to **`RThread`**

The **`Resume()`** function

- Marks the first thread in the process as eligible for execution

Note that there is no **`RProcess::Suspend()`** function

- As processes are not scheduled
- Threads form the basic unit of execution and run inside the protected address space of a process

Processes

On Windows

- The emulator runs within a single Win32 process called **`EPOC.exe`**
- Each Symbian OS process runs as a separate thread inside it

On EKA1

- The emulation of processes on Windows is incomplete
- **`RProcess::Create()`** returns **`KErrNotFound`**

On EKA2

- This has been removed
- Symbian OS still runs in a single process
- But the emulation is enhanced ...
- **`RProcess::Create()`** translates to creation of a new **`Win32`** thread

System Structure: Part One

- ✓ DLLs in Symbian OS
- ✓ Writable Static Data
- ✓ Executables in ROM and RAM
- ✓ Threads and Processes

System Structure

Part Two

System Structure

This Lecture Examines

- Inter-process communication (IPC)
- Recognizers
- Panics and assertions

symbian Academy

Inter-Process Communication (IPC)

- Recognize the preferred mechanisms for IPC on Symbian OS (client–server, publish and subscribe and message queues), and demonstrate awareness of which mechanism is most appropriate for given scenarios
- Understand the use of publish and subscribe to retrieve and subscribe to changes in system-wide properties, including the role of platform security in protecting properties against malicious manipulation

Client-Server

The Client–Server framework

- Is a common form of inter-process communication (IPC) on Symbian OS
- The client–server framework will be discussed in detail in a later lecture

Clients connect to servers

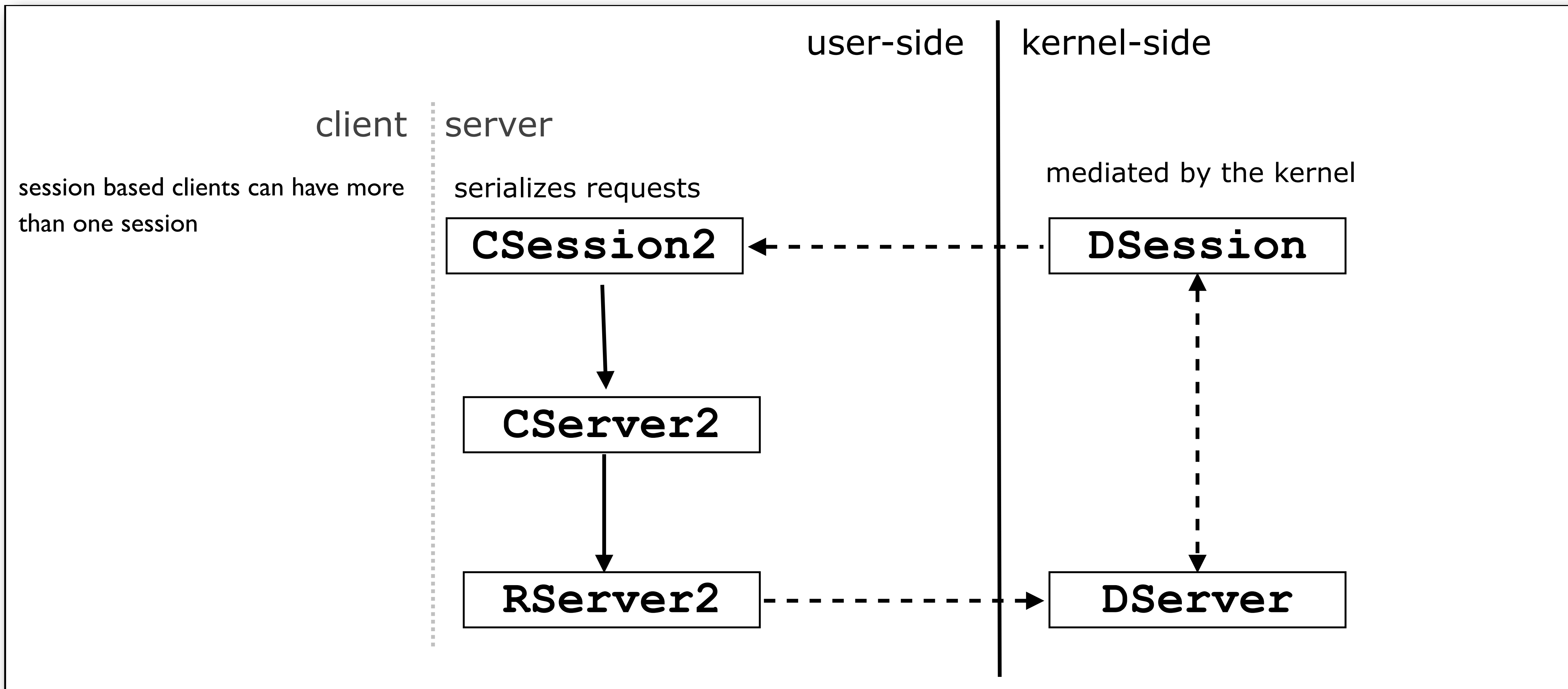
- To establish a session for all further communication
- A session consists of client requests and server responses mediated by the kernel

Session-based communication

- Ensures that all clients will be notified in the case of an error or shutdown of a server
- All server resources will be cleaned up if an error occur
- Or when a client disconnects or dies

Client-Server

Do not worry about the details now as client-server architecture shall be examined in a later lecture



Client-Server

The client-server communication paradigm

- Is used for many clients requiring reliable concurrent access to a service or shared resource
- The server serializes and mediates access to the service accordingly

There are some limitations:

- Clients must know which server provides the service they need
- A permanent session must be maintained between client and server
- It is not really suitable for event multicasting

  (Server-initiated "broadcast" to multiple clients)

Inter-Process Communication (IPC)

In order to overcome such limitations

- Symbian OS version 8.0 was extended

To offer additional IPC mechanisms:

- Publish and subscribe
- Message queues
- Shared buffer I/O

Publish and subscribe and message queues

- Are described in this lecture

Inter-Process Communication (IPC)

Shared buffer I/O

- Is not discussed because it is intended primarily for device driver developers

It is used

- To allow a device driver and its clients to access the same memory area
- Without copying even during interrupt handling

Publish and Subscribe

The publish and subscribe mechanism

- Provides asynchronous multicast event notification
- Connectionless communication between threads

Publish and subscribe

- Provides a means to define and publish changes to system-wide global variables known as "properties"

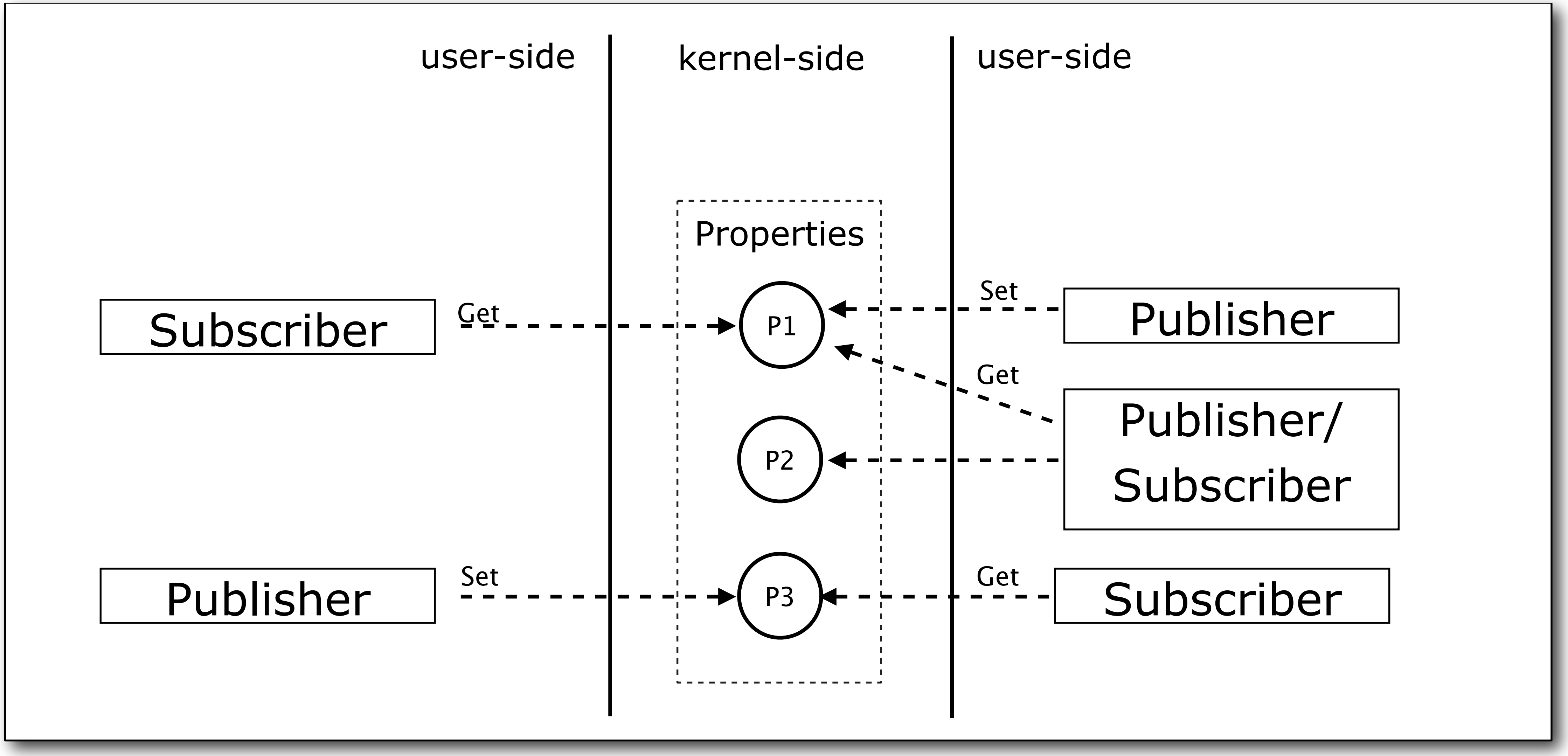
Changes to the properties

- Can be communicated ("published") to more than one interested ("subscribed") peer asynchronously

Publishers and subscribers

- Can dynamically join and leave without any connection set-up or tear-down

Publish & Subscribe



Publish and Subscribe

Properties are data values

- Uniquely identified by a 64-bit integer
- Which is the only information that must be shared between a publisher and a subscriber (typically through a common header file)
- There is no need to provide interface classes or functions for a property

Subscribers

- Do not need to know which component is publishing to a property

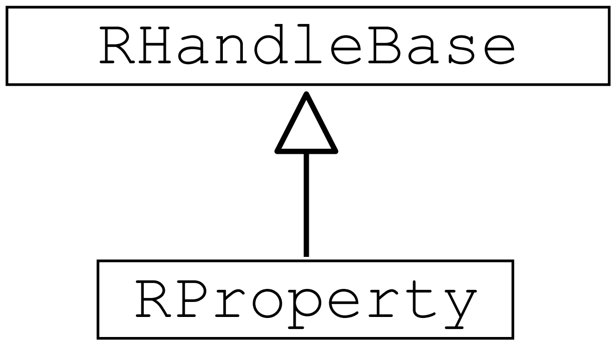
They only need to know:

- About the publish and subscribe API
- The identity of the property of interest to them

Publish and Subscribe

The publish and subscribe API

- Is supplied by the **`RProperty`** class



The identity of a property is composed of two parts:

- A category - defined by a standard UID which specifies the category to which the property belongs
- A key - which uniquely identifies a property within a particular category
- Its value depends on how keys within the category are enumerated

Publish and Subscribe

A property holds a single data variable which may be either

- A 32-bit integer

A byte array (a descriptor) of

- Up to 512 bytes in length
- Unicode text (also up to 512 bytes in size)
- Or even large byte arrays of up to 65 536 bytes.

A thread may take the role

- Of either the publisher or the subscriber

Publish and Subscribe

Any thread can define a property

- By calling **`RProperty::Define()`** to create the variable
- And specify its type and access controls

Once a property has been defined

- It will persist in the kernel until it is deleted explicitly or the system reboots
- The property's lifetime is not linked to that of the defining thread or process

symbian Academy

Publish and Subscribe

Properties can be published or retrieved

- Using a previously attached handle
- Or by specifying the property's identity for each call

On EKA2

- The benefit of attaching to an existing handle is that it has a deterministic bounded execution time
- This makes it suitable for high-priority real-time tasks

Publish and Subscribe

A property is published

- By calling `RProperty::Set()`
- This writes a new value "atomically" to the property
- Ensuring that access by multiple threads is handled correctly

When a property is published

- All outstanding subscriptions are completed
- Even if the value is actually unchanged
- This allows the property to be used as a simple broadcast notification

Publish and Subscribe

To subscribe to a property

- A client must register interest by attaching to it
- Calling the asynchronous **`RProperty::Subscribe()`** method

Publish and Subscribe

Notification happens in the following stages:

1. A client registers its interest in the property

   By attaching to it **`RProperty::Attach()`** - and calling **`Subscribe()`** on the resulting handle passing in a **`TRequestStatus`** reference

2. Upon publication of a new value the client gets notified via a signal to the **`TRequestStatus`** object to complete the **`Subscribe()`** request

3. The client retrieves the value of the updated property by calling **`RProperty::Get()`**

4. The client can re-submit a request for notification of changes to the property by calling **`Subscribe()`** again

Publish and Subscribe

It is not necessary for a property to be defined

- Before it is accessed
- This is known as “lazy definition”

It is not a programming error

- for a property to be published before it has been defined
- This is known as “speculative publishing”

    Attaching to an undefined property is not necessarily an error

Publish and Subscribe

A **`Subscribe()`** request on an undefined property will not complete until either:

- The property is defined and published
- Or the subscriber unsubscribes by canceling the request using **`RProperty::Cancel()`**

Publish and Subscribe

Publish and subscribe is used when a component needs to supply or consume timely and transient information

- To or from an unknown number and type of interested parties
- While remaining decoupled from them

A typical example

- Is the notification of a change to the device's radio states

For example

- Flight-mode
- Bluetooth radio on/off
- WiFi on/off

Publish and Subscribe and Platform Security

On the secure platform of Symbian OS v9

- To ensure that processes are partitioned so that one process cannot interfere with the property of another process
- The category UID of the property should match the secure identifier of the defining process

Alternatively

- The process calling `RProperty::Define()` must have `WriteDeviceData` capability

Properties must also be defined

- With security policies using `TSecurityPolicy` objects

Publish and Subscribe and Platform Security

For processes to publish the property value, the following are required

- The capabilities
- (And/or) vendor identifier
- (And/or) secure identifier

For processes to subscribe to the property the following are required

- The capabilities
- (And/or) vendor identifier
- (And/or) secure identifier

More on Platform Security and capabilities in a later lecture

Publish and Subscribe and Platform Security

For example

- Before accepting a subscription to a property
- The security policy defined when the property was created is checked
- The subscription request completes with **`KErrPermissionDenied`** if the check fails

Message Queues

In contrast to the connection-oriented nature of client–server IPC

- Message queues (`RMsgQueue`) offer a peer-to-peer, many-to-many communication mechanism

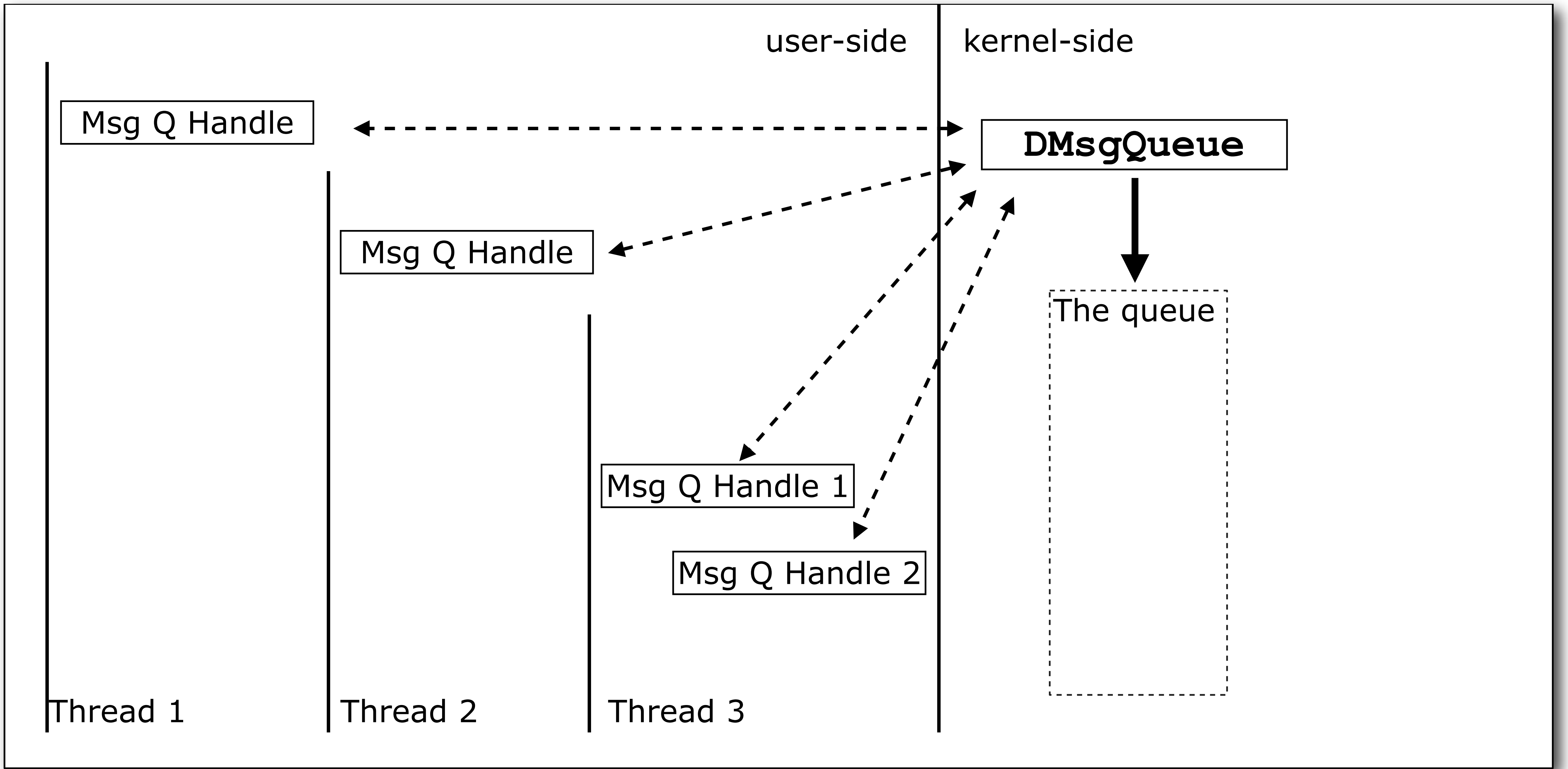
Message queues

- Provide a way to send data (messages) to interested parties
- Without needing to know whether any thread is listening
- Or the identity of a recipient

Messages are sent

- To the queue rather than to any specific recipient
- A single queue can be shared by many readers and writers

Message Queues



Message Queues

A message

- Is an object that is placed into a queue for delivery to recipients
- A queue is normally created for messages of a given type

A queue

- Is created to deal with messages of a defined (fixed) length
- Which must be a multiple of four bytes

Message Queues

The size of a queue

- i.e. the maximum number of messages or slots it can contain
- Is fixed when the queue is created

The maximum size of the message and of the queue

- Are limited only by system resources

Message Queues

A message queue

- Allows two or more threads to communicate without setting up a connection to each other

A message queue is a mechanism for passing data:

- Between threads that run in separate processes (using a global queue which is named and visible to other processes)
- Between threads within a process using a local queue which is not visible to other processes ...

Within a process

- The messages can point to memory mapped to that process and can be used for passing descriptors and pointers between threads

Message Queues

Message queues allow

- For "fire-and-forget" IPC from senders to recipients
- Lend themselves well to event notification

Publish and subscribe

- Is good for notification of state changes which are inherently transient

Message queues

- Are useful for allowing information to be communicated beyond the lifetime of the sender

Message Queues

An a good example of using message queues:

- A central logging subsystem can use a message queue to receive messages from numerous threads
- That may or may not still be running at the point the messages are read and processed

However

- Neither messages nor queues are persistent
- They are cleaned up when the last handle to the queue is closed

Recognizers

- Recognize correct statements about the role of recognizers in Symbian OS

Recognizers

Recognizers

- Are a good example of the use of framework plug-in DLLs
- The framework which loads the recognizers is provided by the application architecture server (Apparc)

Up to Symbian OS v9.1

- Apparc implemented its own custom loading of recognizer plug-ins

In later releases it has been modified to use ECOM

Recognizers

When a file in the file system needs to be associated with an application

- Apparc <u>opens the file</u> and reads some data from the start of it into a buffer
- It then calls `DoRecognizeL()` on each recognizer in the system in turn
- Passing in the data it read into the buffer
- If a plug-in "recognizes" it it returns its data type (MIME type)

Recognizers <u>do not handle the data</u>

- They just try to identify its type
- So that the data can be passed to the application that can best use it

Recognizers

The plug-in recognizer architecture

- Allows developers to create additional data recognizers
- Adding them to the system by installing them

All data recognizers

- Must implement the polymorphic interface defined by **`CApaDataRecognizerType`**
- Which has three virtual functions ...

`DoRecognizeL()`

`DoRecognizeL()`

- Performs data recognition
- This function is not pure virtual but must be implemented

Each implementation

- Should set a value to indicate the MIME type it considers the data to belong to

And a value to indicate a level of confidence ranging from:

- **`ECertain`** - the data is definitely of a specific data type
- **`ENotRecognized`** - the data is not recognized

`SupportedDataTypeL()`

`SupportedDataTypeL()`

- Returns the MIME types that the recognizer is capable of recognizing
- This pure virtual function must be implemented by all recognizer plug-ins

Each recognizer's implementation of **`SupportedDataTypeL()`**

- Is called by the recognizer framework
- After all the recognizers in the system have been loaded
- To build up a list of all the types the system can recognize

`PreferredBufSize()`

`PreferredBufSize()`

- Specifies the size in bytes of the buffer passed to **`DoRecognizeL()`**
- That the recognizer needs to work with
- This function is not pure virtual but must be implemented

Panics and Assertions

- Know the type of parameters to pass to **`User::Panic()`** and understand how to make them meaningful
- Understand the use of **`__ASSERT_DEBUG`** statements to detect programming errors in debug code by breaking the flow of code execution using a panic
- Recognize that **`__ASSERT_ALWAYS`** should be used more sparingly because it will test statements in released code too and cause code to panic if the assertion fails

Panics

When a thread is panicked

- It stops running

Panics are used

- To highlight a programming error in the most noticeable way
- By stopping the thread to ensure that the code is fixed
- Rather than potentially causing serious problems by continuing to run

There is no recovery from a panic

- Unlike a leave - a panic can’t be trapped
- A panic is terminal

Panics

If a panic occurs in the main thread of a process

- The entire process in which the thread runs will terminate

If a panic occurs in a secondary thread

- It is only that thread which closes

If a thread is deemed to be a system thread

- That is - essential for the system to run
- A panic in that thread will reboot the phone

This is very rare

- Since the code running in system threads on Symbian OS is mature and well-tested

symbian Academy

Panics

On phone hardware

- And in release builds on the Windows emulator
- The end result of a panic is either a reboot or an “Application closed” message box

In debug emulator builds

- A panic can be set to break into the debugger
- Known as “just-in-time” debugging

The developer can use the debugger

- To look through the call stack to see where the panic arose
- Thus to examine the state of appropriate objects and variables

Panics

A call to the static function `User::Panic()`

- Panics the currently running thread

On EKA2

- A thread may panic any other thread in the same process
- By acquiring an `RThread` handle and using it to call `RThread::Panic()`

On EKA1

- This function could be used to panic any unprotected thread in any process
- This was deemed insecure for EKA2

Panics

The only occasion for EKA2 ...

- Where a thread running inside a user process can panic another thread in a different process

Is for a server thread to panic a badly-behaved client

- By using the **`RMessagePtr2::Panic()`** method

Panics

`User::Panic()` and **`RThread::Panic()`** take two parameters:

- A panic category string
- An integer error code which can be any value, positive, zero or negative.

Without breaking into the debugger

- These values should still be sufficient for a developer to determine the cause of a panic

Panics

The panic string

- Should be short and descriptive for a programmer rather than for a user - since the user should never see them

Panics should only be used as a means to eliminate programming errors during the development cycle

- For example by using them in assertion statements
- Panicking cannot be seen as useful functionality for properly debugged software

Panics

The following is a very bad example of the use of a panic to indicate a problem to a user:

```
_LIT(KTryDifferentMMC, "File was not found, try selecting another");
User::Panic(KTryDifferentMMC, KErrNotFound); // Not helpful!
```

The following is a good example of the use of a panic

- To highlight a programming error to a developer calling a function in class Bar of the Foo library, and passing in invalid arguments
- The developer can determine which method is called incorrectly and fix the problem:

```
_LIT(KFooDllBarAPI, "Foo.dll, Bar::ConstructL")
User::Panic(KFooDllBarAPI, KErrArgument);
```

Panics

Symbian OS

- Has a series of well-documented panic categories for example:
- `KERN-EXEC`
- `E32USER-CBASE`
- `ALLOC`
- `USER`
- And associated error values

The details of which can be found in the Symbian OS Library which accompanies each SDK

Assertions

Assertions are used

- To check that assumptions made about code are correct
- For example - the states of objects, function parameters or return values are as expected

Typically

- An assertion evaluates a statement
- If it is false it halts execution of the code

There is an assertion macro for debug builds only

- `__ASSERT_DEBUG`

For both debug and release builds

- `__ASSERT_ALWAYS`

Assertions

The assertion macro tests a statement

- If it evaluates to false it calls the method specified in the second parameter passed to the macro

The method is not hard-coded to be a panic

- But rather than return an error or leave it should always terminate the running code and flag up the failure
- Panics are the best choice

Assertions

Assertions help the detection

- Of invalid states or bad program logic so that code can be fixed

It makes sense to stop the code at the point of error

- Rather than return an error as it is easier to track down the bug

Assertions

The use of assertions in release builds of code should be considered carefully

- Assertion statements have a cost in terms of size and speed
- If the assertion fails - it will cause code to terminate with a panic
- Resulting in a poor user experience

Assertions

This is one example of how to use the debug assertion macro:

```
void CTestClass::EatPies(TInt aCount)
    {
    #ifdef _DEBUG
    _LIT(KMyPanicDescriptor, "CTestClass::EatPies");
    #endif
    __ASSERT_DEBUG((aCount>=0),
            User::Panic(KMyPanicDescriptor, KErrArgument));
    ... // Use aCount
    }
```

Assertions

It is more common for a class or code module to define:

- A panic function
- A panic category string
- A set of specific panic enumerators

For example

- The following enumeration could be added to **`CTestClass`**
- So as not to pollute the global namespace

```
enum TTestClassPanic
    {
    EEatPiesInvalidArgument, // Invalid argument passed to EatPies()
    ...                      // Enum values for assertions
                             // in other CTestClass methods
    };
```

Assertions

A panic function is defined either

- As a member of the class
- Or as a static function within the file containing the implementation of the class:

```
static void CTestClass::Panic(TInt aCategory)
    {
    _LIT(KTestClassPanic, "CTestClass");
    User::Panic(KTestClassPanic, aCategory);
    }
```

- The assertion in **`EatPies()`** can then be written as follows:

```
void CTestClass::EatPies(TInt aCount)
    {
    __ASSERT_DEBUG((aCount>=0), Panic(EEatPiesInvalidArgument));
    ... // Use aCount
    }
```

Assertions

The advantage of using an identifiable panic descriptor and enumerated values for different assertion conditions

- Is traceability

This is particularly useful for calling code using a given library

- The developer may not have access to the library source code
- Only access to the header files

Assertions

If the panic string is clear and unique

- A developer should be able to locate the class which raised the panic
- Use the panic category enumeration to find the associated failure
- Which is named and documented to explain clearly why the assertion failed

Assertions

Code with side effects

- Should not be called within assertion statements

The code may well behave as expected in debug mode

- But in release builds the assertion statements are removed by the preprocessor
- With them potentially vital steps in the programming logic

Assertions

Rather than use the "condensed" statements

```
// Bad use of assertions!
__ASSERT_DEBUG(FunctionReturningTrue(), Panic(EUnexpectedReturnValue));
__ASSERT_DEBUG(++index<=KMaxValue, Panic(EInvalidIndex));
```

- Statements should be evaluated independently
- With their returned values then passed into the assertion macros

Panics, Assertions and Leaves

Leaves

- May legitimately occur under exceptional conditions

Such as:

- Out of memory
- Insufficient disk space
- Or the absence of a communications link

It is not possible

- To stop a leave from occurring
- Code should implement a graceful recovery strategy
- Always catch leaves using TRAP statements

symbian Academy

Panics, Assertions and Leaves

Programming errors ("bugs") can be caused by:

- Contradictory assumptions
- Unexpected design errors
- Genuine implementation errors

These are persistent and unrecoverable errors

- Which should be detected and corrected by the programmer
- Rather than handled at run-time

The mechanism to do this

- Is to use assertion statements

These terminate the flow of execution of code if an error is detected, using a panic

- Panics cannot be caught and handled gracefully
- The programmer has to fix the problem

System Structure : Part Two

- ✓ Inter-Process Communication (IPC)
- ✓ Recognizers
- ✓ Panics and Assertions