Process collocation and core affinity deployment

Document WINNF-15-R-0015

Version V1.0.0

29 May 2015

Terms and Conditions

This document has been prepared by the SCA 4.1 Draft Adjudication Work Group to assist The Software Defined Radio Forum Inc. (or its successors or assigns, hereafter "the Forum"). It may be amended or withdrawn at a later time and it is not binding on any member of the Forum or of the SCA 4.1 Draft Adjudication Work Group.

Contributors to this document that have submitted copyrighted materials (the Submission) to the Forum for use in this document retain copyright ownership of their original work, while at the same time granting the Forum a non-exclusive, irrevocable, worldwide, perpetual, royalty-free license under the Submitter's copyrights in the Submission to reproduce, distribute, publish, display, perform, and create derivative works of the Submission based on that original work for the purpose of developing this document under the Forum's own copyright.

Permission is granted to the Forum's participants to copy any portion of this document for legitimate purposes of the Forum. Copying for monetary gain or for other non-Forum related purposes is prohibited.

Intellectual Property Rights

Use this chart in all contributions

THIS DOCUMENT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY USE OF THIS SPECIFICATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE FORUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER, DIRECTLY OR INDIRECTLY, ARISING FROM THE USE OF THIS DOCUMENT.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the specification set forth in this document, and to provide supporting documentation.

Proposal

This document contains a proposal to change the Draft SCAv4.1 specification to provide the capability to support dynamic threading, when intended, for Executable Device Component OS process address space and also for a separate OS process address space. The proposal also intends to allow application threads to be mixed with platform component threads in the same OS process address space. Furthermore, this proposal adds support for multi-core devices deployment via core affinity requirements in a SAD and DCD supported by an executable device component that manages a multi-core processor.

Proposal authors:

- Jerry Bickle, Raytheon
- François Lévesque, NordiaSoft
- Steve Bernier, NordiaSoft

Proposal reviewers:

- Chuck Linn, Harris
- Sarah Miller, Rockwell Collins
- Christoper J. Hagen, Rockwell Collins
- Eric Nicollet, Thales

Recommendation

SCA v4.1 Process collocation and core affinity deployment

Topics

Description of the Issue

Summary of the Proposal

Detailed Proposal

Specifications Changes

- Main Specification Changes
- IDL Specification Changes
- Appendix A: Glossary
- Appendix D-1: PSM - Document Type Definition (DTD) Files Specification Changes
- SCA User Guide

Description of the Issue

POSIX Operating Systems support dynamic loading of libraries and dynamic creation of threads within an OS process address space, thus allowing the capability of a thread to be dynamically added to a OS process. Furthermore, operating systems today support multi-core processors and different techniques to run processes/threads across different cores.

Currently, one can collocate platform components or application components in the same OS process address space using a Factory Component but not together. The use of a Factory Component is static configuration on the types of components that Factory Component can create up. Also, the SCA offers limited support for multi-core devices deployment. Indeed, it uses an executable device component per core or an executable device component for all cores, which results in letting the OS make the decision on what processor core to deploy executables.

At the moment, an Executable Device Component implementation could support OS process creation and/or OS thread creation within its address space. However, there is no standard approach to provide to an ExecutableDeviceComponent the desired deployment strategy on single-core nor on multi-core processors.

Rationale

Applications across all device categories continue to require better performances. Two main trends are driving the embedded device market today:

1. smaller form factors
2. improved performance per watt

However, traditional method of achieving better performances via higher clock frequency leads to increased thermal dissipation and energy requirements

Multicore technology improves performance per watt ratios. It also reduces board real-estate requirements

Rationale

Multi Core computing has gained widespread acceptance in embedded systems.

1. **Symmetric Multi Processing (SMP):** One operating system controls more than one identical processor/core. In SMP, all processors/cores must be able to access the same memory and the same I/O devices
   - Interactions between tasks can be done via memory access
   - Interactions can also be done using IPCs

2. **Asymmetric Multi Processing (AMP):** Multiple operating systems are used; one operating system for each processor/core. Operating systems do not need to be the same. Processors/cores do not need to be identical. Homogeneous AMP vs Heterogeneous AMP
   - Interactions between tasks cannot be done via memory access
   - Interactions must be done using IPCs

Rationale

AMP is good when:

- Communication speed between cores is not critical
- When more than one operating system is needed (legacy code, Security requirements, etc.)

SMP is good when:

- Communication speed between cores is critical
- Workload needs to be distributed across processors/cores dynamically

AMP is mostly used with multiple processors

SMP is mostly used with multicore processors

Rationale

Concurrency

- Having two or more tasks in progress at the same time (time slicing).

Parallelism

- Having two or more tasks executing at the same time.

With AMP, parallelism is achieved executing several tasks on different processors

With SMP, parallelism is achieved executing several tasks on different cores of a single processor

Rationale

Most common Operating Systems all support SMP. Some support AMP.

SMP is a feature that is supported by the OS scheduler which needs to allocate each tasks to a core. Different OSs use different techniques to decide choose a target core.

By default, to avoid overloading a single core, all operating systems use a form of load balancing algorithm that can move tasks to under-utilized cores.

Real-time operating system offer the possibility to influence the scheduling of time-critical tasks. This is generally offered via the concept of Core Affinity. The following operating systems support Core Affinity:

VxWorks® WIND RIVER

Green Hills SOFTWARE INTEGRITY

BlackBerry QNX

LynxOS FROM LYNUXWORKS

Powered By GNU/Linux

Windows

Rationale

```
Cpuset_t affinity;
CPUSET_ZERO(affinity)
CPUSET_SET(affinity, 1);
CPUSET_SET(affinity, 3);
taskCpuAffinitySet(taskId, affinity);
```

VxWorks®
WIND RIVER

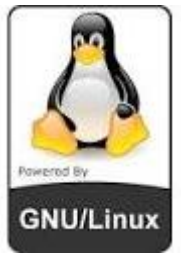
```
my_data = malloc(...);
memset(my_data, ...);
rmask = ((int *)my_data) + 1;
RMSK_SET(1, rmask);
ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHEIRT, my_data);
```

BlackBerry
QNX

```
SetTaskProcessorBinding(TaskId, TRUE, 1);
SetTaskProcessorBinding(TaskId, TRUE, 2);
```

Green Hills SOFTWARE
INTEGRITY

```
cpu_set_t  mask;
CPU_ZERO(&mask);
CPU_SET(0, &mask);
CPU_SET(2, &mask);
sched_setaffinity(threadID, sizeof(mask), &mask);
```



```
SetThreadAffinityMask(::GetCurrentThread(), threadAffinityMask);
```

Windows

Rationale

Some Operating Systems offer more sophisticated scheduling schemes. For instance, VxWorks supports the concept of "Core Reservation" which means that a Task can reserve a Core. This prevents other tasks from running on the reserved Core.

Core reservation can also be done with Core Affinity by setting the affinity of every task to not use a specific Core and by allowing a single task to have an affinity for the reserved core

This proposal adds Core Affinity support to the SCA. Affinity is the most basic SMP scheduling technique and is widely supported by embedded operating systems.

The current proposal does not add support for the more advanced scheduling techniques since they vary significantly from on RTOS to another

Rationale

The new feature will allow a developer to specify a Core Affinity for the deployment of SCA components. When a preference is specified (it is optional), it will be stored in Assembly Descriptors (SAD and DCD) at the component instantiation level

The Core Framework will be required to feed the Core Affinity to the ExecutableDevice for each component that contains an Affinity

If the ExecutableDevice implements support for Core Affinity, it will be responsible for mapping the Core Affinity requirements to the underlying operating system

Summary of the Proposal

- Add an optional sub-element to SAD and DCD component instantiation to specify a core affinity.
- Define a new options parameter for the ExecutableInterface::execute operation to provide a core affinity value that will be used to indicate a preference for a specific core to execute a component.
- Add a new parameter for the ApplicationFactory::create operation to specify core affinity assignments.
- Add an optional attribute to SAD and DCD component instantiation to specify a process collocation.
- Define a new options parameter for the ExecutableInterface::execute operation to provide a process collocation value that will be used to execute a component within a specific address space.
- Define a new option parameter for the ExecutableInterface::execute operation to provide the entry point for a function to execute from a shared library.
- Remove InvalidFunction exception from being thrown by ExecutableInterface::execute operation.

Detailed Proposal

In SAD XML file, the recommendation is to add a processcollocation attribute and a coreaffinity sub-element to componentinstantiation element, and add a executionaffinityassignments element to assemblyinstantiation element.

D-1.10.1.3.1.2 componentinstantiation

The *componentinstantiation* element's optional processcollocation attribute indicates a specific logical process in which the component instance must be executed. The *processcollocation attribute* is used as part of the options parameter for the *ExecutableInterface execute* operation.

```
<!ELEMENT componentinstantiation
( usagename?
, componentproperties?
, coreaffinity*
, deploymentdependencies?
, findcomponent?
)>
<!ATTLIST componentinstantiation
id ID #REQUIRED
stringifiedobjectref CDATA #IMPLIED
processcollocation CDATA #IMPLIED>
```

Update Figure 22.

Detailed Proposal

D-1.10.1.3.1.2.x coreaffinity

The optional *coreaffinity* element is used to indicate preference for execution of a component instance on specific a processor core. The *coreaffinity* is used as part of the options parameter for the *ExecutableInterface execute* operation.

Data type for the value of this option is unsigned long, with values being platform dependents.

<!ELEMENT coreaffinity (#PCDATA)>

Detailed Proposal

D-1.10.1.3.3.1 assemblyinstantiation

```
<!ELEMENT assemblyinstantiation
( usagename?
, componentproperties?
, deviceassignments?
, executionaffinityassignments?
, deploymentdependencies?
)>
<!ATTLIST assemblyinstantiation
id ID #REQUIRED>
```

Update Figure 27.

Add the following section.

D-1.10.1.3.3.1.4 executionaffinityassignments

The optional executionaffinity*assignments* element provides a list of *executionaffinityassignment* elements which are used when deploying the sub-application's components.

Detailed Proposal

In an *executionaffinityassignment* element, the *componentid* attribute refers to the *componentinstantiationref* within the scope of the sub-application being created. The optional processcollocation attribute indicates a specific logical process in which the component instance of the sub-application must be executed. The *processcollocation attribute* is used as part of the options parameter for the *ExecutableInterface execute* operation. The optional *coreaffinity* element is used to indicate preference for execution of component instances of the sub-application on a specific processor core. The *coreaffinity* is used as part of the options parameter for the *ExecutableInterface execute* operation.

```
<!ELEMENT executionaffinityassignments
( executionaffinityassignment+
)>

<!ELEMENT executionaffinityassignment
( coreaffinity*
)>
<!ATTLIST executionaffinityassignment
componentid CDATA #REQUIRED
processcollocation CDATA #IMPLIED>
```

Detailed Proposal

In DCD XML file, the recommendation is to add a processcollocation attribute and a coreaffinity sub-element to componentinstantiation element.

D-1.11.1.4.1.5 componentinstantiation

The *componentinstantiation* element's optional processcollocation attribute indicates a specific logical process in which the component instance must be executed. The *processcollocation attribute* is used as part of the options parameter for the *ExecutableInterface execute* operation.

```
<!ELEMENT componentinstantiation
( usagename?
,componentproperties?
,coreaffinity*
,componentfactoryref?
)>
<!ATTLIST componentinstantiation
id ID #REQUIRED
stringifiedobjectref CDATA #IMPLIED
processcollocation CDATA #IMPLIED>
```

Update Figure 38.

Detailed Proposal

D-1.11.1.4.1.5.x coreaffinity

The optional *coreaffinity* element is used to indicate preference for execution of a component instance on specific a processor core. The *coreaffinity* is used as part of the options parameter for the *ExecutableInterface execute* operation.

Data type for the value of this option is unsigned long, with values being platform dependents.

<!ELEMENT coreaffinity (#PCDATA)>

Detailed Proposal

D-1.6.1.6.3 code

The code element (see Figure 4) is used to indicate the local filename of the code that is described by the softpkg element, for a specific implementation of the software component. Options parameters stacksize and priority are used by the ExecutableInterface execute operation. Data types for the values of ~~these~~ stacksize and priority options are unsigned long. The stacksize element provides the means to specify a stack size for the process/thread being created. The priority element provides the means to specify the scheduling priority for the process/thread being created. ~~The type attribute for the code element will also indicate the type of file being delivered to the system.~~ Options parameter entrypoint is used by the ExecutableInterface execute operation. The entrypoint element provides the means for providing the name of the entry point of the component being delivered. The data type for the value of entrypoint option is string. The type attribute of the code element indicates the type of file being delivered to the system.

Detailed Proposal

3.1.3.4.1.6.3.1 InvalidProcess

The InvalidProcess exception indicates that a process or thread, as identified by the ~~processId~~executionId parameter, does not exist on this device.

Remove section 3.1.3.4.1.6.3.2

~~**3.1.3.4.1.6.3.2 InvalidFunction**~~

~~The InvalidFunction exception indicates that a function, as identified by the input name parameter, hasn't been loaded on this device.~~

~~exception InvalidFunction{};~~

Detailed Proposal

3.1.3.4.1.6.3.3 ~~Process~~ExecutionID_Type

The ~~Process~~ExecutionID_Type contains information for a process and a thread ~~number~~ id within the system. The ~~process number~~ ExecutionID_Type is unique to the processor operating system that created the process and thread. The threadId field is the thread id provided by the operating system when a thread is created to execute a function as specified in the entry point options parameter. The processId field is the process identifier provided by the operating system when a process is created either to execute a file or to execute a function as specified in the entry point options parameter. The processCollocation field is the value of the process collocation options parameter when specified. The cores field is the value of the processor cores used to execute the process.

~~typedef long ProcessId_Type;~~

```
struct ExecutionID_Type
{
  unsigned long long threadId;
  unsigned long long processId;
  string processCollocation;
  CF::ULongSeq cores;
};
```

Detailed Proposal

3.1.3.4.1.6.3.6 STACK_SIZE_ID

~~The~~ STACK_SIZE_ID is the identifier for the *execute* operation options parameter. STACK_SIZE_ID is used to set the operating system's process/thread stack size. ~~SCA277~~ The value for a stack size ~~shall be~~ is an unsigned long.

const string STACK_SIZE_ID = "STACK_SIZE";

3.1.3.4.1.6.3.7 PRIORITY_ID

~~The~~ PRIORITY_ID is the identifier for the *execute* operation options parameter. PRIORITY_ID is used to set the operating system's process/thread priority. ~~SCA278~~ The value for a priority ~~shall be~~ is unsigned long.

const string PRIORITY_ID = "PRIORITY";

Detailed Proposal

Add new constants for ExecutableInterface options parameters.

3.1.3.4.1.6.3.x EXEC_DEVICE_PROCESS_SPACE

~~The~~ EXEC_DEVICE_PROCESS_SPACE is the constant value known for the execute operation PROCESS_COLLOCATION_ID option parameter.

const string EXEC_DEVICE_PROCESS_SPACE = “DEVICE";

3.1.3.4.1.6.3.x PROCESS_COLLOCATION_ID

The PROCESS_COLLOCATION_ID is the identifier for the execute operation options parameter. PROCESS_COLLOCATION_ID is used to select the process from within which the entry point function must be invoked. A PROCESS_COLLOCATION_ID value of EXEC_DEVICE_PROCESS_SPACE means the entry point is invoked from within the process of the Executable Device Component. A PROCESS_COLLOCATION_ID empty value means a new process is created to invoke the entry point. A PROCESS_COLLOCATION_ID of any other value means the entry point is invoked from within a process associated with that logical process value. The value for a process collocation is a string.

Detailed Proposal

const string PROCESS_COLLOCATION_ID = “PROCESS_COLLOCATION";

3.1.3.4.1.6.3.x ENTRY_POINT_ID

~~The~~ ENTRY_POINT_ID is the identifier for the execute operation options parameter. ENTRY_POINT_ID is used to identify the name of the entry point function that must be invoked. The value for a entry point is a string.

const string ENTRY_POINT_ID = "ENTRY_POINT";

3.1.3.4.1.6.3.x CORE_AFFINITY_ID

~~The~~ CORE_AFFINITY_ID is the identifier for the execute operation options parameter. CORE_AFFINITY_ID is used to identify the processor core where to execute a process. The value for a core affinity is a CF::ULongSeq.

const string CORE_AFFINITY_ID = "CORE_AFFINITY";

Detailed Proposal

3.1.3.4.1.6.5.1.2 Synopsis

~~Process~~ExecutionID_Type execute (in string filename, in Properties options, in Properties parameters) raises (InvalidState, InvalidFunction, InvalidParameters, InvalidOptions, InvalidFileName, ExecuteFail);

3.1.3.4.1.6.5.1.3 Behavior

SCA279 The *execute* operation shall execute the ~~function or~~ file identified by the input filename parameter using the input parameters and options parameters. ~~Whether the input name parameter is a function or a file name is device-implementation-specific.~~

SCA280 The *execute* operation shall map the input parameters (id/value string pairs) parameter as an argument to the operating system "execute/thread" function. The argument (e.g. argv) is an array of character pointers to null-terminated strings where the last member is a null pointer and the first element is the input filename parameter. Thereafter the second element is mapped to the input parameters[0] id, the

Detailed Proposal

third element is mapped to the input parameters[0] value and so forth until the contents of the input parameters parameter are exhausted.

The execute operation input options parameters are STACK_SIZE_ID, ~~and~~ PRIORITY_ID, PROCESS_COLLOCATION_ID, ENTRY_POINT_ID, and CORE_AFFINITY_ID.

~~SCA281 The *execute* operation shall use these options, when specified, to set the operating system's process/thread stack size and priority, for the executable image of the given input name parameter.~~

3.1.3.4.1.6.5.1.4 Returns

SCA282 The *execute* operation shall return a unique ~~process~~ ExecutionID_Type for the process/thread that it created. The threadId is zero when no ENTRY_POINT_ID is specified.

Detailed Proposal

3.1.3.4.1.6.5.1.5 Exceptions/Errors

~~SCA284 The *execute* operation shall raise the InvalidFunction exception when the function indicated by the input name parameter does not exist for the device to be executed.~~

SCA285 The *execute* operation shall raise the CF InvalidFileName exception when the file name indicated by the input file name parameter does not exist for the device to be executed.

SCA286 The *execute* operation shall raise the InvalidParameters exception when the input parameter ID or value attributes are not valid strings.

SCA287 The execute operation shall raise the InvalidOptions exception when the input options parameter does not comply with sections 3.1.3.4.1.6.3.6 STACK_SIZE_ID, ~~and~~ 3.1.3.4.1.6.3.7 PRIORITY_ID, 3.1.3.4.1.6.3.x PROCESS_COLLOCATION_ID, 3.1.3.4.1.6.3.x ENTRY_POINT_ID, and 3.1.3.4.1.6.3.x CORE_AFFINITY_ID.

SCA288 The *execute* operation shall raise the ExecuteFail exception when the operating system "execute/thread" function is not successful.

Detailed Proposal

3.1.3.4.1.6.5.2 terminate

3.1.3.4.1.6.5.2.1 Brief Rationale

The *terminate* operation provides the mechanism for terminating the execution of a process/thread on a specific device that was started up with the *execute* operation. The terminate operation may terminate a process when all threads within that process have been terminated.

3.1.3.4.1.6.5.2.2 Synopsis

void terminate (in ~~Process~~ExecutionID_Type ~~process~~executionId) raises (InvalidProcess, InvalidState);

3.1.3.4.1.6.5.2.3 Behavior

SCA289 The *terminate* operation shall terminate the execution of the process/thread designated by the ~~process~~executionId input parameter on the device ~~to be executed~~. When threadId is 0, the specified process will be terminated, including all threads it contains. When a specific threadId is provided, only that thread will be terminated.

Detailed Proposal

3.1.3.4.1.6.5.2.4 Returns

This operation does not return a value.

3.1.3.4.1.6.5.2.5 Exceptions/Errors

SCA291 The *terminate* operation shall raise the InvalidProcess exception when the ~~process~~ executionId does not exist for the device.

Detailed Proposal

Update 3.1.3.1.3.29 SpecializedInfo Identifiers

Since the type returned by ExecubleInterface::execute is ExecutionID_Type instead of ProcessID_Type

Remove PROCESS_ID

~~This string constant is the identifier for ExecutableInterface::ProcessID_Type value within a ComponentType's specializedInfo.~~

~~const string PROCESS_ID = "PROCESS_ID";~~

Add EXECUTION_ID

This string constant is the identifier for ExecutableInterface::ExecutionID_Type value within a ComponentType's specializedInfo.

const string EXECUTION_ID = "EXECUTION_ID";

Update Appendix C and SpecializedInfo IDL file.

Detailed Proposal

Add the possibility to perform core affinity assignment when an application is instantiated.

3.1.3.1.3.x ExecutionAffinityType

The CF ExecutionAffinityType defines a structure that associates a component with a process collocation and/or processor cores on which it is executed. Only processCollocation and coreAffinity values that are specified (non-empty) are used.

```
struct ExecutionAffinityType
{
  string componentId;
  string processCollocation;
  CF::ULongSeq coreAffinities;
};
```

Detailed Proposal

3.1.3.1.3.x ExecutionAffinitySequence

The IDL sequence, CF ExecutionAffinitySequence, provides an unbounded sequence of CF ExecutionAffinityTypes.

typedef sequence <ExecutionAffinityType> ExecutionAffinitySequence;

3.1.3.3.1.3.5.1.2 Synopsis

ApplicationManager create (in string name, in Properties initConfiguration, in DeviceAssignmentSequence deviceAssignments, in Properties deploymentDependencies, in ExecutionAffinitySequence executionAffinityAssignments) raises (CreateApplicationError, CreateApplicationRequestError, InvalidInitConfiguration);

3.1.3.3.1.3.5.1.3 Behavior

SCAXXX The *create* operation shall use the values contained in the input executionAffinityAssignments parameter. These values have precedence over the ApplicationFactoryComponent profile's *processcollocation* attribute and/or *coreaffinity* elements.

Detailed Proposal

In section Appendix A, add definitions

A.2 Definitions

Logical process

A process that this associated with a specific name within an ExecutableDeviceComponent and that can be identified via a PROCESS_COLLOCATION_ID option parameter.

Process

A process is composed of one region in runtime memory where execution happens (called an address space) and of one-to-many threads. A terminology other than process may be used in some operating systems.

Thread

The smallest sequence of programmed instructions that can be managed independently by an operating system.

Detailed Proposal

In User guide, add a section to explain and clarify that process collocation and core affinity requirements are applied once an ExecutableDevice has been selected using the regular deployment rules (allocation properties, device assignments, deployment channels, etc.) already described in the SCA specification.