The Safe and Secure Virtual Machine - Sense-VM architecture

Author: Bo Joel Svensson, Abhiroop Sarkar

WORK IN PROGRESS - DRAFT

Sense-VM is a virtual machine for IoT and embedded applications in general. Sense-VM is a bytecode virtual machine (think Java-VM, not an operating system hypervisor) that provides a base level of safety, robustness and security.

Sense-VM targets 32/64Bit microcontroller based systems with at least about 128Kb of Read Write Memory (RWM).

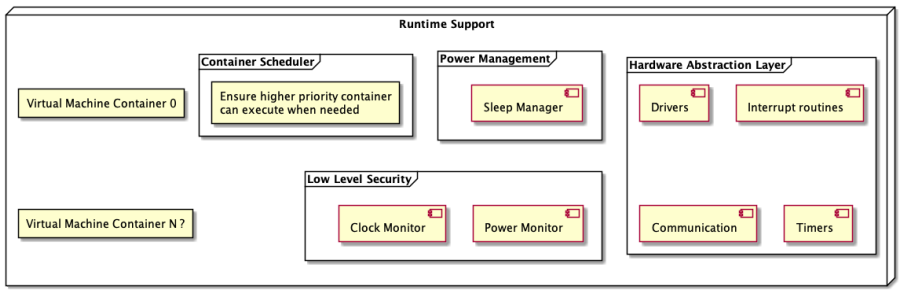
Sense-VM consists of a runtime-system for execution of compiled linearly-stored bytecode within isolated containers. The bytecode programs cannot mutate arbitrary memory addresses and all accesses to underlying hardware goes via the runtime system.

Sense-VM can concurrently run a number of isolated virtual exection environments, called Virtual Machine Containers. Each virtual machine container contains the resources needed to execute one program and consist of registers and memory that is private to that environment. A program running within a virtual machine container could potentially be concurrent itself, running a cooperative scheduler. A criticality level (priorities) can be associated with each virtual machine container. Peripherals and external hardware resources are assigned to a single virtual machine container to rule out competition for resources between applications of different criticality (priority).

Sense-VM is implemented in standard C for portability and static analysers, infer (from Facebook) and scan-build (from the Clang framework) are used to detect problematic code.

Sense-VM specifies an interface downwards towards hardware functionality and peripherals, but implemnetation of that interface from below is not considered a part of Sense-VM per se. Microcontroller platforms come in very different forms so it makes sense to leave the low-level implementation to the specific use case. We do however provide implementations based on ChibiOS (Or contikiOs or ST HAL, or so on..) (TODO: Such interfaces downwards towards the hardware are not yet specified) (TODO: No such “ref” implementation is implemented or specified)

The Sense-VM Virtual Machine



Hardware Abstraction Layer

The hardware abstraction layer (HAL) will probably be implemented by building upon an existing HAL system such as ChibiOS, ContikiOs or ZephyrOS. Other HALs could also be candidates: CMSIS, HAL from ST for STM32 platforms for example. We should try to keep as much of the code as possible portable so that it can be ported to HALs that parhaps have more desirable features. The HAL will be a collection of functions (and perhaps routines running on a timer interrupt periodically if needed) that can be called from the runtime support system.

Sleep Manager

If there is no processing going on (reported from the various schedulers) the sleep manager puts the system to sleep for a period of time such that it wakes up when it is time for the context/container with the earliest "wake-up" time to start executing.

The sleep manager may also be prompted to wake the system up by for example a sensor driver when the sensor has new samples to process.

It is not unthinkable that there are applications that will just sleep indefinitely unless there is outside stimulus. It may still be desirable that these applications wake up periodically and just let the monitoring system know that it is still alive and ok (a heartbeat).

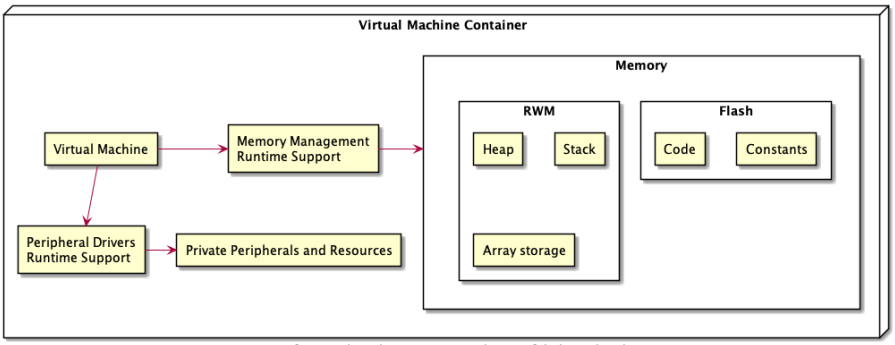
Container Scheduler and Virtual Machine Containers

A virtual machine container is an isolated executing environment for a virtual machine.

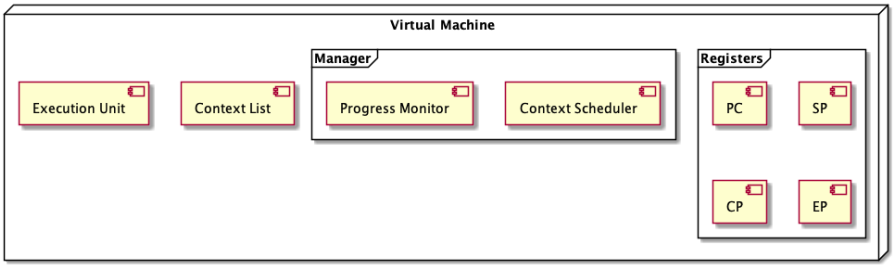
The container scheduler ensures that each container gets a time slot to execute. Possibly, these containers could be implemented using a "thread" feature of an underlying OS/RTOS/HAL or be given a certain number of iterations of some main loop.

Containers should be isolated from eachother to allow different level of criticality to different containers. High criticality containers should be prioritised over low

criticality containers.



A VM container consist of a virtual machine instance and a set of dedicated and private resources.



Virtual Machine

The virtual machine is based upon the Categorical Abstract Machine (CAM) and consists of number of registers and and an execution unit. (More details later)

Execution Unit

The execution unit reads instructions from code memory at location stored in PC register and evaluates each instruction over the state.

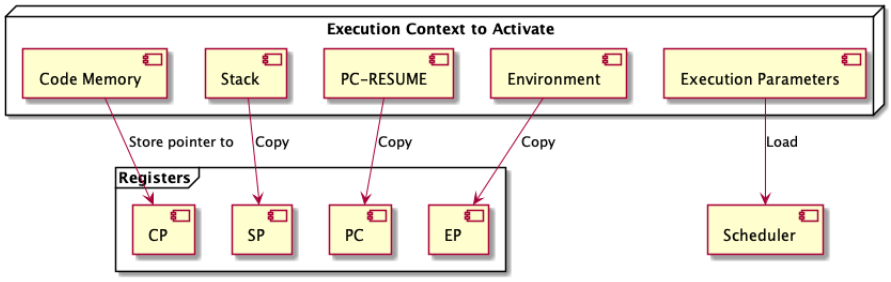
Context Scheduler and Context List

There is a list of contexts (separate instances of code and state) that can execute in a time-shared fashion on the execution unit.

(TODO: Maybe cooperative scheduling at this level? This would require "go to sleep" operations in the bytecode.)

The contexts that are executed on one VM are sharing the memory resources of the Virtual Machine Container.

Context Switching



The context list consists of some number of "Execution Context to Activate". A context is activated by copying the stored register state into the VM. Putting a context to sleep works in the reversed way.

Scheduling

Virtual Machine Container Scheduling

Priority based scheduling with pre-emption between virtual machine containers. The VMC scheduling is implemented on top of threading functionality of the underlying HAL. If there are no threading abstractions in the HAL, the scheduling can be implemented using a periodic timer interrupt or explicitly allocating a number of iterations of a main-loop to each VMC.

Cooperative scheduling within a Virtual Machine

A program may consist of a number of cooperating tasks that are never pre-empted. The running task gives up the Execution Unit by executing `sleep ns` instruction.

The Categorical Abstract Machine

Instruction set

To save space in the instr-set, constants are stored in a constants pool (Constants memory). Instructions such as `LOADI` (that takes stores an integer value into the value register) takes and index into an array of integers stored in the constants memory.

OpCode Mnemonic	Value	Arguments	Size (Bytes - including arrguments)
FST	0x00	0	1
SND	0x01	0	1
ACC <n>	0x02	1	2
REST <n>	0x03	1	2

OpCode Mnemonic	Value	Arguments	Size (Bytes - including arrguments)
PUSH	0x04	0	1
SWAP	0x05	0	1
LOADI <i>	0x06	1	3
LOADB <b>	0x07	1	2
CLEAR	0x08	0	1
CONS	0x09	0	1
CUR <l>	0x0A	1	2
PACK <t>	0x0B	1	3
SKIP	0x0C	0	1
STOP	0x0D	0	1
APP	0x0E	0	1
RETURN	0x0F	0	1
CALL <l>	0x10	1	2
GOTO <l>	ox11	1	2
GOTOFALSE <l>	0x12	1	2
SWITCH <n> <t_1> <l_1> .. <t_n> <l_n>	0x13	1 + 2n	2 + 3n (max n = 256)
ABS	0x14	0	1
NEG	0x15	0	1
NOT	0x16	0	1
DEC	0x17	0	1
ADDI	0x18	0	1
MULI	0x19	0	1
MINI	0x1A	0	1
ADDF	0x1B	0	1
MULF	0x1C	0	1
MINF	0x1D	0	1
GT	0x1E	0	1
LT	0x1F	0	1
EQ	0x20	0	1
GE	0x21	0	1
LE	0x22	0	1

- <n> - Positive ints - 1 byte long
- <l> - Positive ints for label numbers - 1 byte long
- <b> - Boolean 1 byte long; 7 bits wasted
- <t> - Tag for a constructor - 2 bytes long
- <i> - index from int pool - max_index_size = 65536. The int itself can be upto 4 bytes long

FIXME: Currently we use the string pool for tags and not some fixed size 2 bytes tag

Bytecode format

TODO: We should also make room in the bytecode format for checksums

Bytecode file contents	Explanation
FE ED CA FE	*Magic Number* - 4 bytes
FF	*Version of bytecode* - 1 byte;
00 03	*Int Pool count* - 2 bytes - 65536 ints possible
00 0E ED 24	Example integer `978212`, size upto 4 bytes
00 00 CE 35	Example integer `52789`
00 01 C4 4D	Example integer `115789`
...	
00 0A	*String Pool count* - Each byte is indexed unlike Int Pool where every 4 bytes is an index.
48 65 6c 6c 6f	Example string "Hello"
57 6f 72 6c 64	Example string "World"
..	
00 00	*Native Pool count* - 2 bytes - index for native functions
..	
00 00 00 FF	Code Length - Max size of 4 bytes Code
.	
.	

Security Measures

TODO: This needs to be thought out parhaps after doing some kind of a Threat Analysis

Fault Tolerance and Reliability

TODO: Try to build in a base set of functionality for fault tolerance and reliability

Communication

Between contexts within a Virtual Machine Container

Contexts executing within a virtual machine container share a heap. Data can easily be exchanged between tasks if both tasks know of a common name

referencing a heap structure. Since the Heap is private to the container and contexts never pre-empt eachother, no locking mechanism should be needed for accesses to these shared structures.

Between Virtual Machine Containers

Low criticality containers should not be able to influence the execution of a higher criticality container in any way. (TODO: What mechanisms for communication do we need to add for Container -> Container communication?)

Thoughts

Splitting concurrency up between internal to VM container via contexts and external between containers open up to running VM containers on different cores in parallel.

The Concurrency between different containers may allow mixed criticality applications sense each application will execute in an isolated memory and with access only to private resources.

Management of other resources is also important. Communication interfaces, sensors etc. I think it would be beneficial to assign such resources to a VM Container and never allow the same interface to be connected to more than one VM Container.