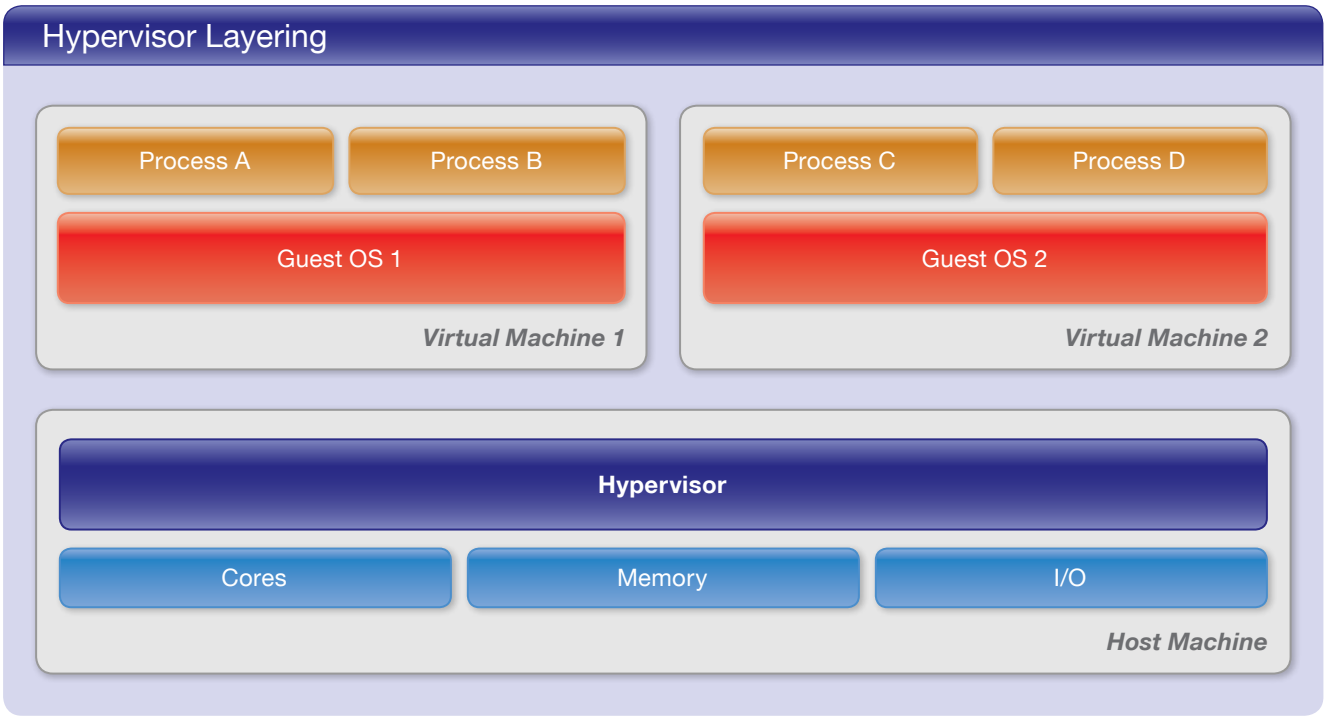
Hypervisor Debugging



In April 2017, Lauterbach will provide the high performance capabilities of its new hypervisor support. This article presents a reference implementation on which a Xen hypervisor with two Linux guests is running on a HiKey board from LeMaker (Cortex-A53).

Virtualization in Embedded Systems

The virtualization concept allows multiple operating systems to be run in parallel on a single hardware platform. Currently, virtualization is being used more and more in embedded systems. For example, in the cockpit of a car, real-time applications that are monitored by an AUTOSAR operating system run on the same hardware platform parallel to Android based user interfaces. A hypervisor, which is the core of virtualization, ensures that everything works reliably and efficiently.

The hypervisor, which is also referred to as a virtual machine monitor, is a software layer fulfilling two tasks:

1. Starting and managing the virtual machines (VMs).
2. Virtualizing the physical hardware resources for the VMs.

An operating system running on a VM is referred to as a guest OS. All accesses by the guests to the virtualized hardware resources are mapped to the physical resources by the hypervisor.

CPU virtualization is important for debugging. Every virtual machine is assigned one or more virtual CPUs (vCPUs). The number of vCPUs does not necessarily have to be the same as the number of CPU cores available on the hardware platform.

Memory virtualization is equally important. The VMs do not see the actual physical memory but see the guest physical memory as virtualized memory. The hypervisor manages a separate page table for each VM to control access to physical memory. Since the application processes, at least on operating systems like Linux, work with virtual addresses anyway, the debugger has to deal with a two stage address translation:

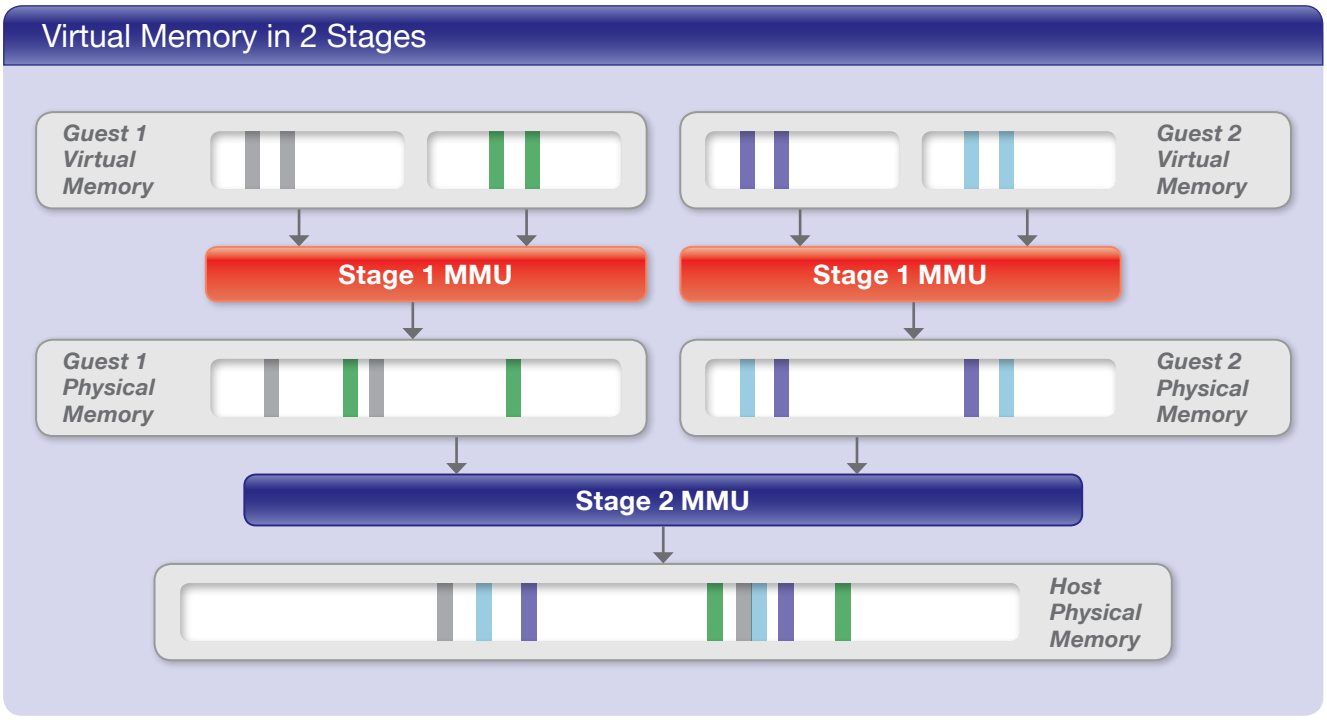
- Guest virtual memory to guest physical memory
- Guest physical memory to host physical memory

See the diagram “Virtual Memory in 2 Stages” on the opposite page:

- The Stage 1 MMUs mapping information is handled by the page table of their guest OS.
- The Stage 2 MMU uses the page tables of the hypervisor.

Extended Debugging Concepts

TRACE32 was systematically extended in 2016 by Lauterbach to provide its customers unlimited



debugging capability with a hypervisor. The following extensions were added:

- A machine ID was added to the TRACE32 command syntax. The machine ID allows the debugger to access the context of the active VM as well as the context of all inactive VMs. A virtual machine is considered active when a core has been allocated to it for execution.
- Using the new hypervisor-awareness, the debugger detects and visualizes the VMs of the hypervisor.
- Instead of only being able to debug a single operating system, it is now possible to debug several operating systems at the same time.
- Instead of only being able to access the OS page tables of the active guests as before, the debugger can now also use the page tables of all inactive guests.

The most important objective for all extensions was seamless debugging of the overall system. This means that when the system has stopped at a breakpoint, you can check and change the current state of every single process, all VMs, plus the current state of the hypervisor and of the real hardware platform. In addition, you can set a program breakpoint at any location in the code.

The unlimited debugging capability that Lauterbach has been offering for almost 20 years now, for operating systems like Linux, formed the starting point for all of these implemented extensions. Therefore, what follows is a brief summary of the most important OS debugging concepts:

Processes run on operating systems in a private virtual address space. The TRACE32 OS-awareness and the TRACE32 MMU support allow users to debug seamlessly across process boundaries:

- With the help of the space ID, it is possible to directly access the virtual address space of each process.
- With the help of the TASK option, it is possible to display the current register set and the stack frame for every single process.

Machine ID

How does this concept need to be extended if the operating systems are running on virtual machines?

1. First, it is necessary to uniquely identify each virtual machine. For this purpose, TRACE32 assigns each VM a number, the machine ID. The machine ID of the hypervisor is 0. Just as the space ID is used to identify the virtual address space of a process, the machine ID is used to identify the private address space of a VM.
2. To show the register set and the stack frame of any process, the debugger must know on which VM and on which guest OS the process is running. The MACHINE option was introduced for this purpose.

TRACE32 Commands

Traditional OS-Aware Debugging	`Data.dump`	`<space _ id>:<virtual _ address>`
	`Data.LOAD.Elf <file>`	`<space _ id>:<virtual _ address>`
	`Register.view`	`/TASK <process _ name>`
	`Frame.view`	`/TASK <process _ name>`
< NEW > Hypervisor Debugging	`Data.dump`	**`<machine _ id>:::`**`<space _ id>::<virtual _ address>`
	`Data.LOAD.Elf <file>`	**`<machine _ id>:::`**`<space _ id>::<virtual _ address>`
	`Register.view`	**`/MACHINE <machine _ id>`** `/TASK <process _ name>`
	`Frame.view`	**`/MACHINE <machine _ id>`** `/TASK <process _ name>`

These two extensions are sufficient to allow the debugger to access all information across process boundaries. The “TRACE32 Commands” overview above provides a comparison of the extended TRACE32 command syntax for hypervisor debugging to the traditional syntax used for OS-aware debugging.

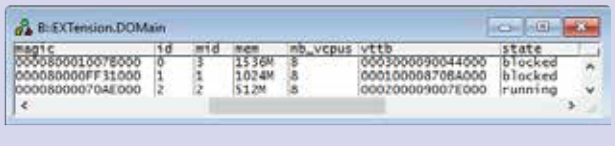
Hypervisor Awareness

Like the OS-awareness functionality, there is now a hypervisor-awareness functionality. This functionality provides the debugger with all information on the hypervisor running on the hardware platform. However, hypervisor-awareness requires the debug symbols for the hypervisor to be loaded. The debugger can then create an overview of all guests. The “Guest List” screenshot for our reference implementation — Xen, Cortex-A53 — shows the following information:

- VM IDs and VM states, number of vCPUs per VM
- Start addresses of the stage 2 page tables (vttb)

The awareness for the particular hypervisor is created by Lauterbach and provided to its customers. An overview of all currently supported hypervisors is shown in the table “Currently Supported Hypervisors” on page 4.

Guest List

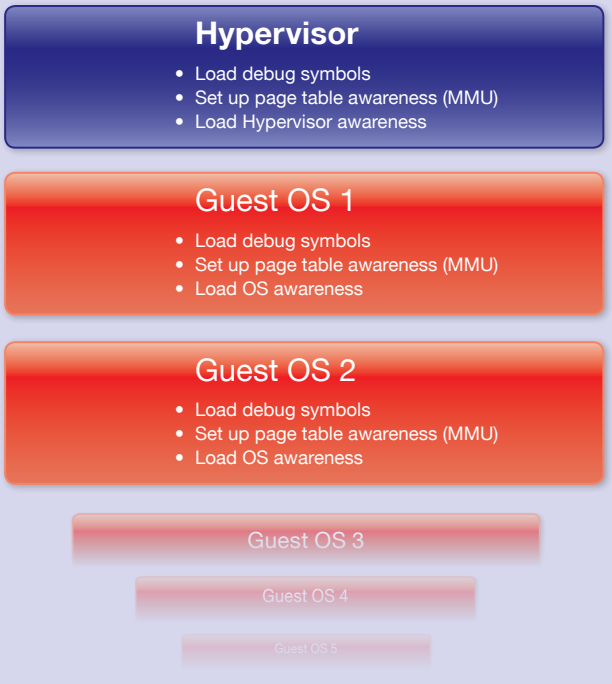


Debugger Configuration

How do the extended debug concepts effect debugging with TRACE32 now? Let‘s first look at the configuration. The following steps are necessary to configure the hypervisor as well as every single guest OS:

1. Load the debug symbols
2. Set up page table awareness (MMU)
3. Load the TRACE32 hypervisor-awareness respectively the TRACE32 OS-awareness

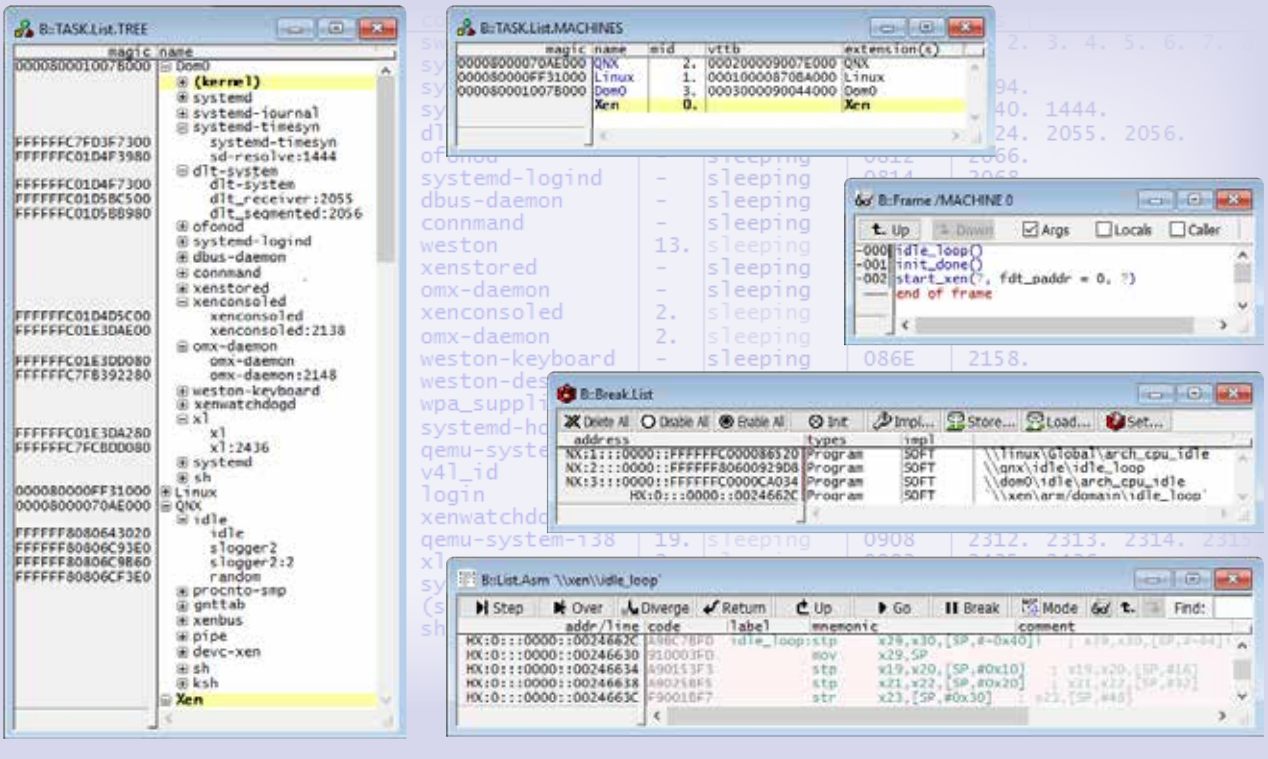
Debugger Configuration



Xen Hypervisor on Cortex-A53

Global Task List

Virtual Machine List



The “Debugger Configuration” diagram shows an overview of the individual configuration steps.

Debug Process

The operation of a debugger must often resolve contradicting requirements. One user group wants simple and intuitive operation while another group demands maximum flexibility and full scripting capabilities. Let‘s first take a look at the intuitive operation. The basic idea is actually very simple: if the debugger stops at a breakpoint, then the GUI visualizes the application process that triggered the breakpoint.

If you are interested in a different application process, then you simply open the TRACE32 global task list. All tasks executing on the overall system are listed there. You can select the task you want to display in the GUI by double-clicking on the task. The global task list also offers a simple way to set program breakpoints for a specific task. Since the debug symbols are associated with a machine ID and a space ID when the .elf file is loaded, functions and variables can be addressed by name as per usual when debugging.

Maximum flexibility and full scripting capabilities can be obtained using the TRACE32 commands. The extended syntax for these commands are presented above.

Summary

Since Lauterbach has systematically extended the well-known concepts for OS-aware debugging to hypervisor debugging, it will be easy for TRACE32 users to get started with just a little practice.

Currently Supported Hypervisors

KVM	Wind River Hypervisor 2.x
VxWorks 653 3.x	Xen

(more to follow)