Hello everyone!

Welcome to the first part of a multi-part series of tutorials called “Hypervisor From Scratch”. As the name implies, this course contains technical details to create a basic Virtual Machine based on hardware virtualization. If you follow the course, you’ll be able to create your own virtual environment and you’ll get an understanding of how VMWare, VirtualBox, KVM and other virtualization softwares use processors’ facilities to create a virtual environment.

Introduction

Both Intel and AMD support virtualization in their modern CPUs. Intel introduced (VT-x technology) that previously codenamed “Vanderpool” on November 13, 2005, in Pentium 4 series. The CPU flag for VT-x capability is “vmx” which stands for Virtual Machine eXtension.

AMD, on the other hand, developed its first generation of virtualization extensions under the codename “Pacifica”, and initially published them as AMD Secure Virtual Machine (SVM), but later marketed them under the trademark AMD Virtualization, abbreviated AMD-V.

There two types of the hypervisor. The hypervisor type 1 called “bare metal hypervisor” or “native” because it runs directly on a bare metal physical server, a type 1 hypervisor has direct access to the hardware. With a type 1 hypervisor, there is no operating system to load as the hypervisor.

Contrary to a type 1 hypervisor, a type 2 hypervisor loads inside an operating system, just like any other application. Because the type 2 hypervisor has to go through the operating system and is managed by the OS, the type 2 hypervisor (and its virtual machines) will run less efficiently (slower) than a type 1 hypervisor.

Even more of the concepts about Virtualization is the same, but you need different considerations in VT-x and AMD-V. The rest of these tutorials mainly focus on VT-x because Intel CPUs are more popular and more widely used. In my opinion, AMD describes virtualization more clearly in its manuals but Intel somehow makes the readers confused especially in Virtualization documentation.

Hypervisor and Platform
These concepts are platform independent, I mean you can easily run the same code routine in both Linux or Windows and expect the same behavior from CPU but I prefer to use Windows as its more easily debuggable (at least for me.) but I try to give some examples for Linux systems whenever needed. Personally, as Linux kernel manages faults like #GP and other exceptions and tries to avoid kernel panic and keep the system up so it's better for testing something like hypervisor or any CPU-related affair. On the other hand, Windows never tries to manage any unexpected exception and it leads to a blue screen of death whenever you didn't manage your exceptions, thus you might get lots of BSODs. By the way, you'd better test it on both platforms (and other platforms too.).

At last, I might (and definitely) make mistakes like wrong implementation or misinformation or forget about mentioning some important description so I should say sorry in advance if I make any faults and I'll be glad for every comment that tells me my mistakes in the technical information or misinformation.

That's enough, Let's get started!

**The Tools you'll need**

You should have a Visual Studio with WDK installed. you can get Windows Driver Kit (WDK) [here](#).

The best way to debug Windows and any kernel mode affair is using **Windbg** which is available in Windows SDK [here](#). (If you installed WDK with default installing options then you probably install WDK and SDK together so you can skip this step.)

You should be able to debug your OS (in this case Windows) using Windbg, more information [here](#).

**Hex-rays IDA Pro** is an important part of this tutorial.

**OSR Driver Loader** which can be downloaded [here](#), we use this tools in order to load our drivers into the Windows machine.

**SysInternals DebugView** for printing the **DbgPrint()** results.
Almost all of the codes in this tutorial have to run in kernel level and you must set up either a Linux Kernel Module or Windows Driver Kit (WDK) module. As configuring VMM involves lots of assembly code, you should know how to run inline assembly within your kernel project. In Linux, you shouldn’t do anything special but in the case of Windows, WDK no longer supports inline assembly in an x64 environment so if you didn’t work on this problem previously then you might have struggle creating a simple x64 inline project but don’t worry in one of my post I explained it step by step so I highly recommend seeing this topic to solve the problem before continuing the rest of this part.

Now its time to create a driver!

There is a good article here if you want to start with Windows Driver Kit (WDK).

The whole driver is this:

```c
#include <ntddk.h>
#include <wdf.h>
#include <ntddk.h>

extern void inline AssemblyFunc1(void);
extern void inline AssemblyFunc2(void);

VOID DrvUnload(PDRIVER_OBJECT DriverObject);
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath);

#pragma alloc_text(PAGE, Example_Unload)
#pragma alloc_text(INIT, DriverEntry)

NTSTATUS
VOID
extern
extern
#include <wdm.h>
#include <wdf.h>
#include <ntddk.h>

if
NtStatus
RtlInitUnicodeString
DbgPrint
UNICODE_STRING
PDEVICE_OBJECT
UINT64
NTSTATUS
uiIndex = 0;
PODEVICE_OBJECT pDeviceObject = NULL;
UNICODE_STRING usDriverName, usDosDeviceName;

DbgPrint("DriverEntry Called.");
RtlInitUnicodeString(&usDriverName, L"\Device\MyHypervisor");
RtlInitUnicodeString(&usDosDeviceName, L"\DosDevices\MyHypervisor");

NtStatus = IoCreateDevice(pDriverObject, 0, &usDriverName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPE

if (NtStatus == STATUS_SUCCESS)
{
   pDriverObject->DriverUnload = DrvUnload;
pDeviceObject->Flags |= IO_TYPE_DEVICE;
pDeviceObject->Flags &= (~DO_DEVICE_INITIALIZING);
   IoCreateSymbolicLink(&usDosDeviceName, &usDriverName);
}
return NtStatus;

VOID DrvUnload(PDRIVER_OBJECT DriverObject)
{
    UNICODE_STRING usDosDeviceName;
    DbgPrint("DrvUnload Called rn");
    RtlInitUnicodeString(&usDosDeviceName, L"\DosDevices\MyHypervisor");
    IoDeleteSymbolicLink(&usDosDeviceName);
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

AssemblyFunc1 and AssemblyFunc2 are two external functions that defined as inline x64 assembly code.

Our driver needs to register a device so that we can communicate with our virtual environment from User-Mode code, on the hand, I defined DrvUnload which use PnP Windows driver feature and you can easily unload your driver and remove device then reload and create a new device.

The following code is responsible for creating a new device:

```c
RtlInitUnicodeString(&usDriverName, L"\Device\MyHypervisor");
RtlInitUnicodeString(&usDosDeviceName, L"\DosDevices\MyHypervisor");

NtStatus = IoCreateDevice(pDriverObject, 0, &usDriverName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPE

if (NtStatus == STATUS_SUCCESS)
{
   pDriverObject->DriverUnload = DrvUnload;
pDeviceObject->Flags |= IO_TYPE_DEVICE;
pDeviceObject->Flags &= (~DO_DEVICE_INITIALIZING);
   IoCreateSymbolicLink(&usDosDeviceName, &usDriverName);
}
```
If you use Windows, then you should disable Driver Signature Enforcement to load your driver, that’s because Microsoft prevents any not verified code to run in Windows Kernel (Ring 0).

To do this, press and hold the shift key and restart your computer. You should see a new Window, then

1. Click **Advanced options**.
2. On the new Window Click **Startup Settings**.
3. Click on **Restart**.
4. On the Startup Settings screen press 7 or F7 to disable driver signature enforcement.

![Startup Settings](image)

Press F10 for more options
Press Enter to return to your operating system

The latest thing I remember is enabling Windows Debugging messages through registry, in this way you can get `DbgPrint()` results through `SysInternals DebugView`.

Just perform the following steps:

In regedit, add a key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter
```

Under that, add a DWORD value named `IHVDRIVER` with a value of `0xFFFF`

Reboot the machine and you’ll good to go.

**Some thoughts before the start**

There are some keywords that will be frequently used in the rest of these series and you should know about them (Most of the definitions derived from *Intel software developer's manual, volume 3C*).
**Virtual Machine Monitor (VMM):** VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.

**Guest Software:** Each virtual machine (VM) is a guest software environment.

**VMX Root Operation and VMX Non-root Operation:** A VMM will run in VMX root operation and guest software will run in VMX non-root operation.

**VMX transitions:** Transitions between VMX root operation and VMX non-root operation.

**VM entries:** Transitions into VMX non-root operation.

**Extended Page Table (EPT):** A modern mechanism which uses a second layer for converting the guest physical address to host physical address.

**VM exits:** Transitions from VMX non-root operation to VMX root operation.

**Virtual machine control structure (VMCS):** is a data structure in memory that exists exactly once per VM, while it is managed by the VMM. With every change of the execution context between different VMs, the VMCS is restored for the current VM, defining the state of the VM's virtual processor and VMM control Guest software using VMCS.

The VMCS consists of six logical groups:

- Guest-state area: Processor state saved into the guest state area on VM exits and loaded on VM entries.
- Host-state area: Processor state loaded from the host state area on VM exits.
- VM-execution control fields: Fields controlling processor operation in VMX non-root operation.
- VM-exit control fields: Fields that control VM exits.
- VM-entry control fields: Fields that control VM entries.
- VM-exit information fields: Read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

I found a great work which illustrates the VMCS, The PDF version is also available [here](#).
### GUEST STATE AREA

<table>
<thead>
<tr>
<th>CR0</th>
<th>CR3</th>
<th>CR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSP</td>
<td>RIP</td>
<td>RFLAGS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selector</th>
<th>Base Address</th>
<th>Segment Limit</th>
<th>Access Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDTR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDTR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDTR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IA32_DEBUGCTL</th>
<th>IA32_SYSENTER_CS</th>
<th>IA32_SYSENTER_ESP</th>
<th>IA32_SYSENTER_EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td>IA32_PAT</td>
<td>IA32_EFER</td>
<td>IA32_BNDCFGs</td>
</tr>
</tbody>
</table>

### HOST STATE AREA

<table>
<thead>
<tr>
<th>CR0</th>
<th>CR3</th>
<th>CR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSP</td>
<td>RIP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selector</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>LDTR</td>
<td></td>
</tr>
<tr>
<td>TR</td>
<td></td>
</tr>
<tr>
<td>GDTR</td>
<td></td>
</tr>
<tr>
<td>IDTR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IA32_SYSENTER_CS</th>
<th>IA32_SYSENTER_ESP</th>
<th>IA32_SYSENTER_EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td>IA32_PAT</td>
<td>IA32_EFER</td>
</tr>
</tbody>
</table>
Don’t worry about the fields, I’ll explain most of them clearly in the later parts, just remember VMCS Structure varies between different version of a processor.

**VMX Instructions**

VMX introduces the following new instructions.

<table>
<thead>
<tr>
<th>Intel/AMD Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVEPT</td>
<td>Invalidate Translations Derived from EPT</td>
</tr>
<tr>
<td>INVVID</td>
<td>Invalidate Translations Based on VPID</td>
</tr>
<tr>
<td>VMCALL</td>
<td>Call to VM Monitor</td>
</tr>
</tbody>
</table>
The following items summarize the life cycle of a VMM and its guest software as well as the interactions between them:

- Software enters VMX operation by executing a VMXON instruction.
- Using VM entries, a VMM can then turn guests into VMs (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.
- VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the VM using a VM entry.
- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

That's enough for now!

In this part, I explained about general keywords that you should be aware and we create a simple lab for our future tests. In the next part, I will explain how to enable VMX on your machine using the facilities we create above, then we survey among the rest of the virtualization so make sure to check the blog for the next part.

References


Published in CPU, Hypervisor and Tutorials

Create a virtual machine  How to create Virtual Machine  Hypervisor fundamentals  Hypervisor Tutorials

Intel Virtualization  Intel VMX  Intel VTX Tutorial  using CPU Virtualization  VMM Implementation