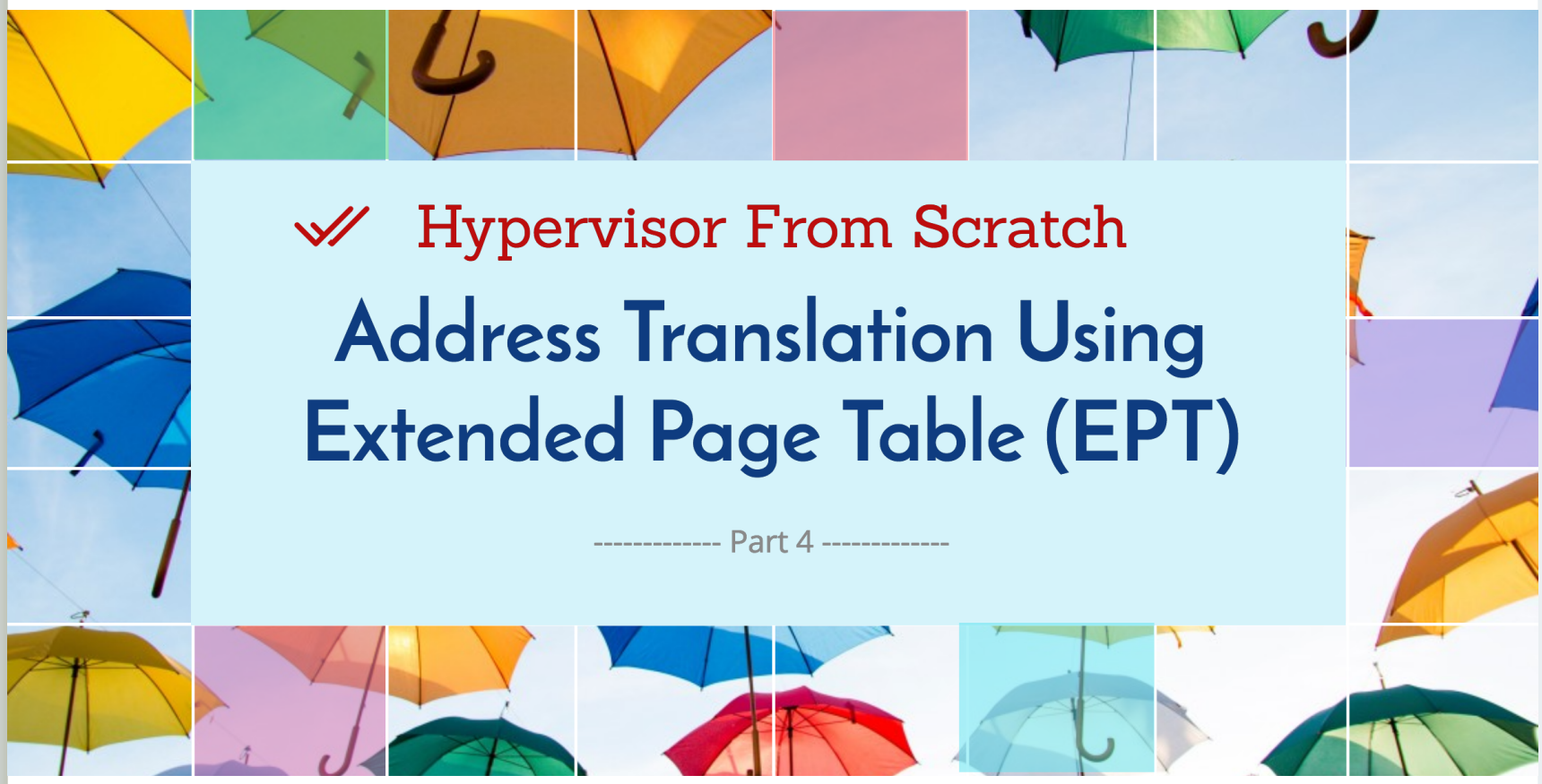
Hypervisor From Scratch – Part 4: Address Translation Using Extended Page Table (EPT)

0

Published October 5, 2018 by Sinaei



Hello guys!

Welcome to the fourth part of the “Hypervisor From Scratch”. This part is primarily about translating guest address through **Extended Page Table (EPT)** and its implementation. We also see how shadow tables work and other cool stuff.

First of all, make sure to read the earlier parts before reading this topic as these parts are really dependent on each other also you should have a basic understanding of paging mechanism and how page tables work. A good article is here for paging tables.

Most of this topic derived from **Chapter 28** – (**VMX SUPPORT FOR ADDRESS**

TRANSLATION) available at Intel 64 and IA-32 architectures software developer's manual combined volumes 3.

The full source code of this tutorial is available on GitHub :

[https://github.com/SinaKarvandi/Hypervisor-From-Scratch]

Before starting, I should give my thanks to Petr Beneš, as this part would never be completed without his help.

Introduction

Second Level Address Translation (**SLAT**) or nested paging, is an extended layer in the paging mechanism that is used to map hardware-based virtualization virtual addresses into the physical memory.

AMD implemented **SLAT** through the **Rapid Virtualization Indexing (RVI)** technology known as **Nested Page Tables (NPT)** since the introduction of its third-generation **Opteron** processors and microarchitecture code name **Barcelona**. **Intel** also implemented **SLAT** in **Intel® VT-x technologies** since the introduction of microarchitecture code name **Nehalem** and its known as **Extended Page Table (EPT)** and is used in **Core i9, Core i7, Core i5**, and **Core i3** processors.

ARM processors also have some kind of implementation known as known as **Stage-2 page-tables**.

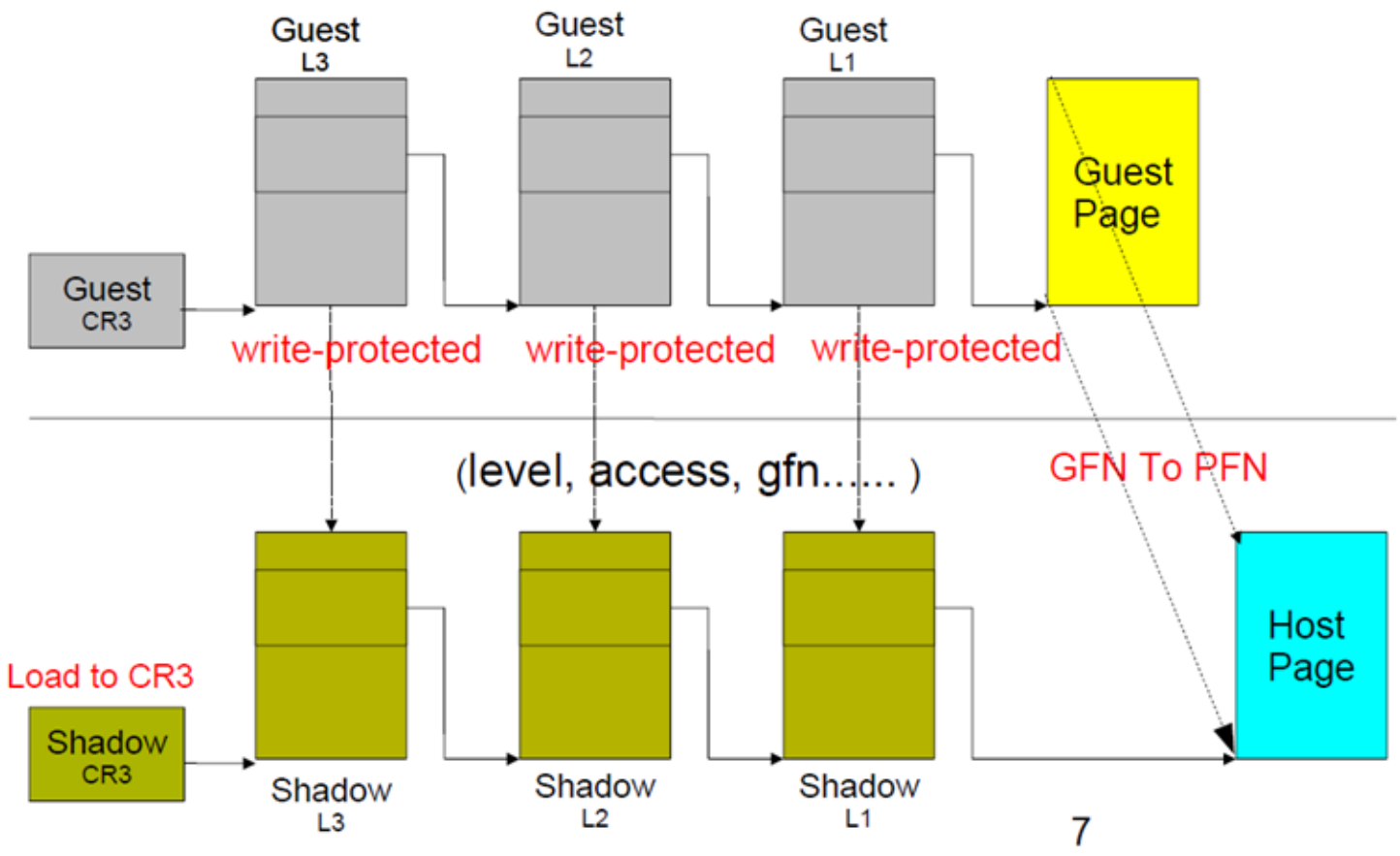
There are two methods, the first one is Shadow Page Tables and the second one is Extended Page Tables.

Software-assisted paging (Shadow Page Tables)

Shadow page tables are used by the hypervisor to keep track of the state of physical memory in which the guest thinks that it has access to physical memory but in the real world, the hardware prevents it to access hardware memory otherwise it will control the host and it is not what it intended to be.

In this case, VMM maintains shadow page tables that map guest-virtual pages directly to machine pages and any guest modifications to V->P tables synced to VMM

V->M shadow page tables.



By the way, using Shadow Page Table is not recommended today as always lead to VMM traps (which result in a vast amount of VM-Exits) and losses the performance due to the TLB flush on every switch and another caveat is that there is a memory overhead due to shadow copying of guest page tables.

Hardware-assisted paging (Extended Page Table)

To reduce the complexity of Shadow Page Tables and avoiding the excessive vm-exits and reducing the number of TLB flushes, EPT, a hardware-assisted paging strategy implemented to increase the performance.

According to a **VMware** evaluation paper: “**EPT** provides performance gains of up to 48% for MMU-intensive benchmarks and up to 600% for MMU-intensive microbenchmarks”.

EPT implemented one more page table hierarchy, to map Guest-Virtual Address to Guest-Physical address which is valid in the main memory.

In EPT,

- One page table is maintained by guest OS, which is used to generate the guest-physical address.
- The other page table is maintained by VMM, which is used to map guest physical address to host physical address.

so for each memory access operation, EPT MMU directly gets the guest physical address from the guest page table and then gets the host physical address by the VMM mapping table automatically.

Extended Page Table vs Shadow Page Table

EPT:

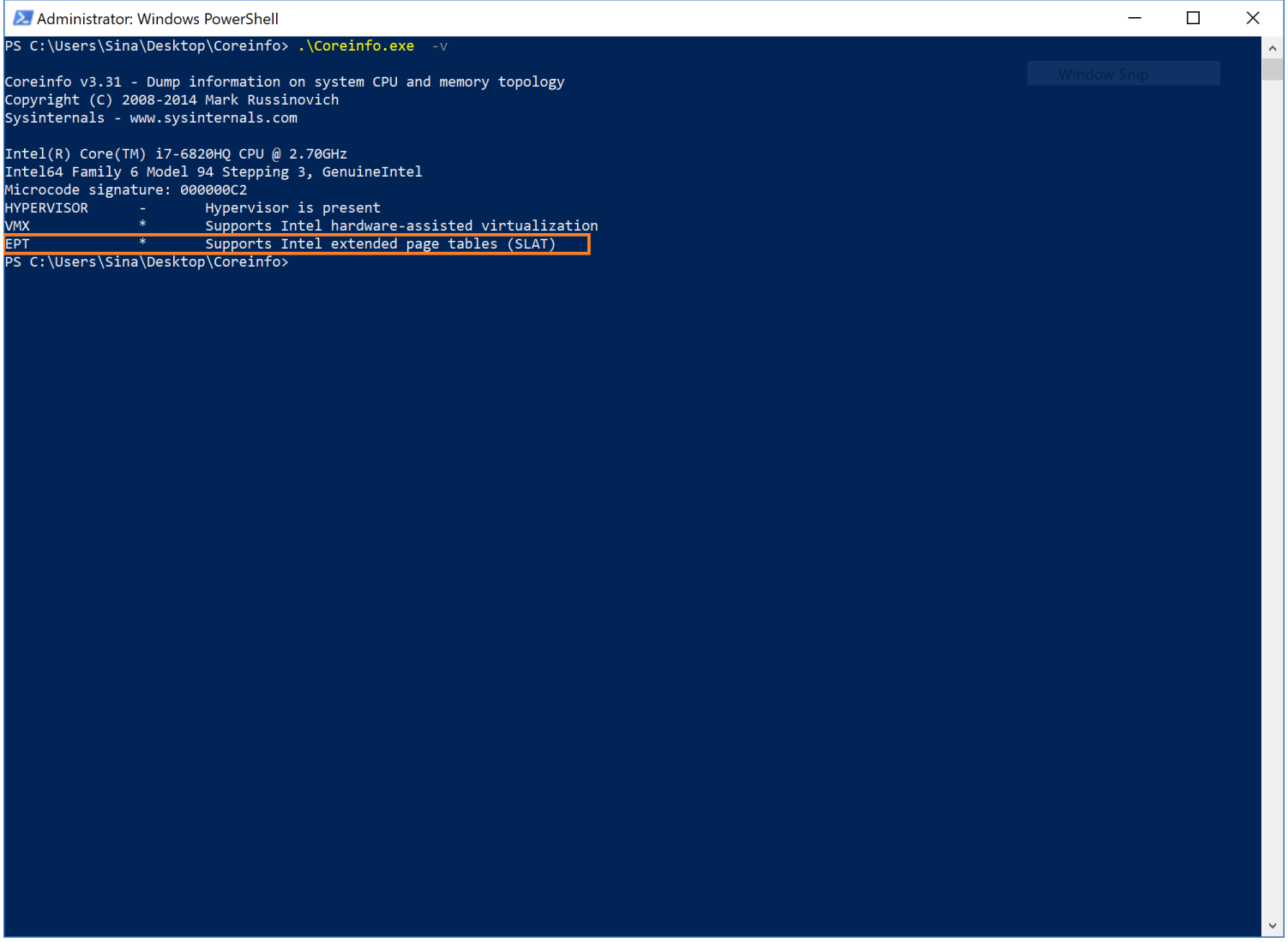
- Walk any requested address
  - Appropriate to programs that have a large amount of page table miss when executing
  - Less chance to exit VM (less context switch)
- Two-layer EPT
  - Means each access needs to walk two tables
- Easier to develop
  - Many particular registers
  - Hardware helps guest OS to notify the VMM

SPT:

- Only walk when SPT entry miss
  - Appropriate to programs that would access only some addresses frequently
  - Every access might be intercepted by VMM (many traps)
- One reference
  - Fast and convenient when page hit
- Hard to develop
  - Two-layer structure
  - Complicated reverse map
  - Permission emulation

Detecting Support for EPT, NPT

If you want to see whether your system supports EPT on Intel processor or NPT on AMD processor without using assembly (CPUID), you can download **coreinfo.exe** from Sysinternals, then run it. The last line will show you if your processor supports EPT or NPT.



EPT Translation

EPT defines a layer of address translation that augments the translation of linear addresses.

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain addresses that would normally be treated as physical addresses (and used to access memory) are instead treated as guest-physical addresses. Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

EPT is used when the “enable EPT” VM-execution control is 1. It translates the guest-physical addresses used in VMX non-root operation and those used by VM entry for event injection.

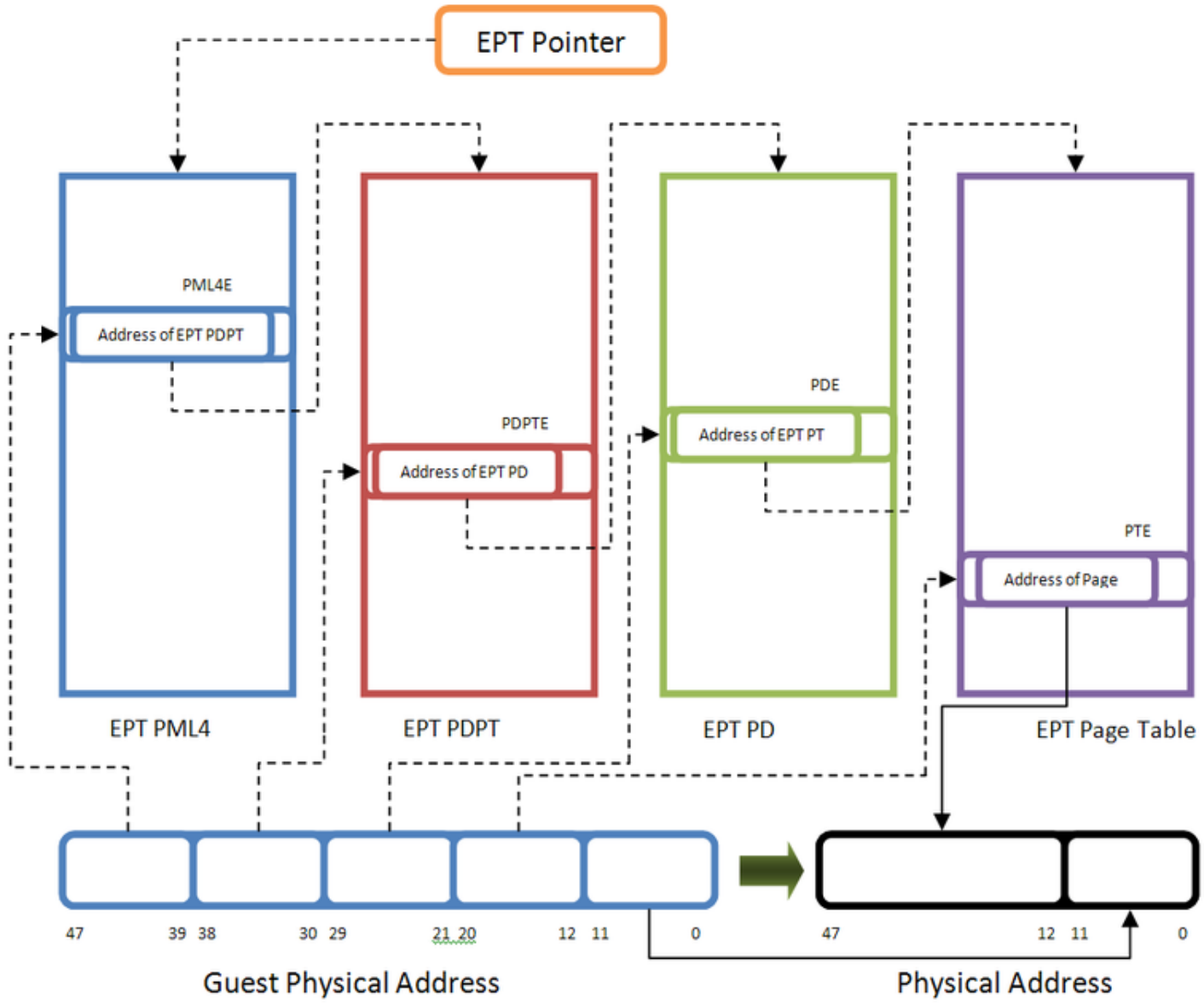
EPT translation is exactly like regular paging translation but with some minor differences. In paging, the processor translates Virtual Address to Physical Address

while in EPT translation you want to translate a Guest Virtual Address to Host Physical Address.

If you're familiar with paging, the 3rd control register (CR3) is the base address of PML4 Table (in an x64 processor or more generally it points to root paging directory), in EPT guest is not aware of EPT Translation so it has CR3 too but this CR3 is used to convert Guest Virtual Address to Guest Physical Address, whenever you find your target Guest Physical Address, it's **EPT mechanism that treats your Guest Physical Address like a virtual address and the EPTP is the CR3**.

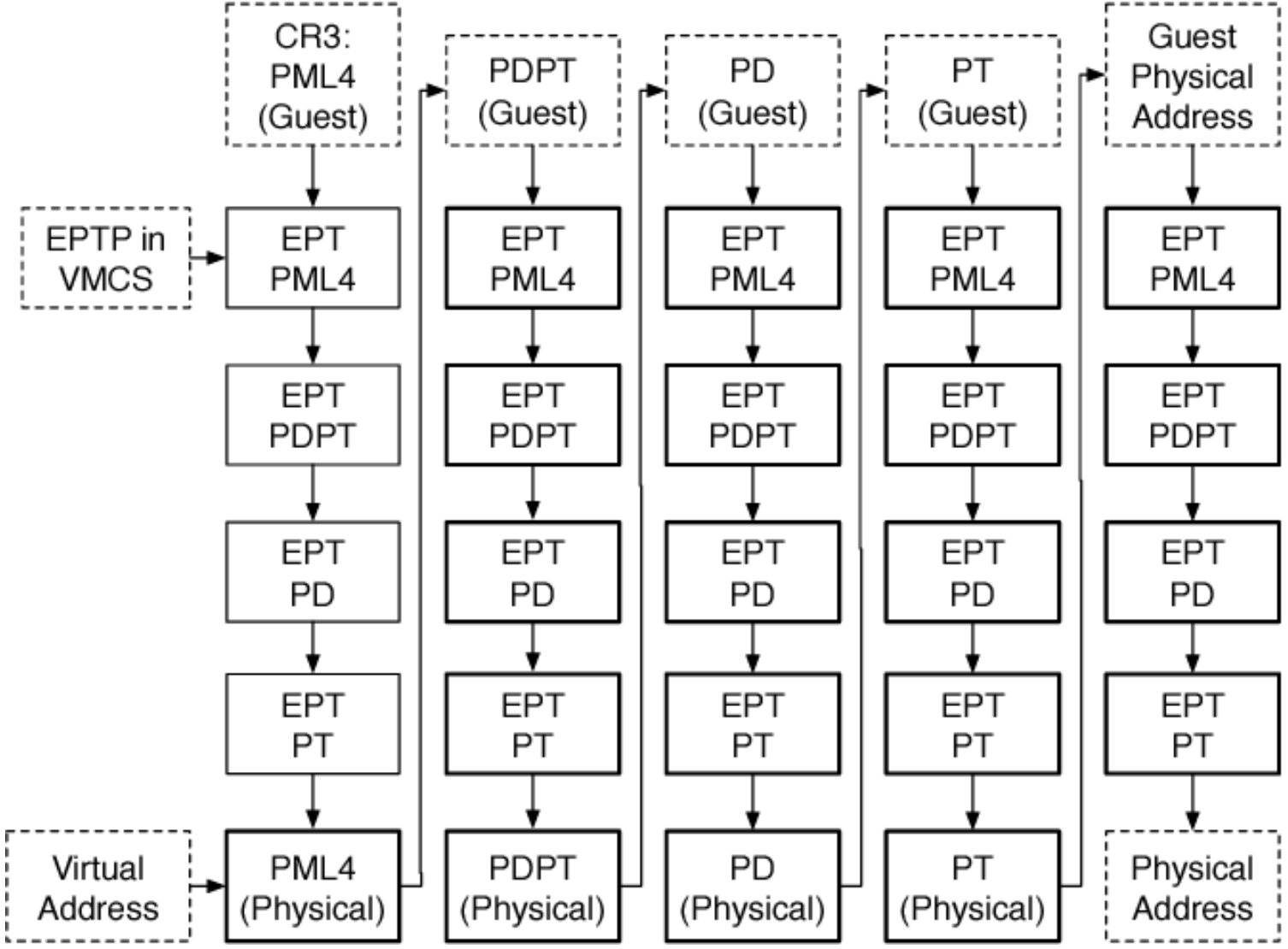
Just think about the above sentence one more time!

So your target physical address should be divided into 4 part, the first 9 bits points to EPT PML4E (note that PML4 base address is in EPTP), the second 9 bits point the EPT PDPT Entry (the base address of PDPT comes from EPT PML4E), the third 9 bits point to EPT PD Entry (the base address of PD comes from EPT PDPTE) and the last 9 bit of the guest physical address point to an entry in EPT PT table (the base address of PT comes form EPT PDE) and now the EPT PT Entry points to the host physical address of the corresponding page.



You might ask, as a simple Virtual to Physical Address translation involves accessing 4 physical address, so what happens ?!

The answer is the processor internally translates all tables physical address one by one, that’s why paging and accessing memory in a guest software is slower than regular address translation. The following picture illustrates the operations for a Guest Virtual Address to Host Physical Address.



If you want to think about x86 EPT virtualization, assume, for example, that CR4.PAE = CR4.PSE = 0. The translation of a **32-bit** linear address then operates as follows:

- Bits 31:22 of the linear address select an entry in the guest page directory located at the guest-physical address in CR3. The guest-physical address of the guest page-directory entry (PDE) is translated through EPT to determine the guest PDE's physical address.
- Bits 21:12 of the linear address select an entry in the guest page table located at the guest-physical address in the guest PDE. The guest-physical address of the guest page-table entry (PTE) is translated through EPT to determine the guest PTE's physical address.
- Bits 11:0 of the linear address is the offset in the page frame located at the guest-physical address in the guest PTE. The guest physical address determined by this offset is translated through EPT to determine the physical address to which the original linear address translates.

Note that PAE stands for **P**hysical **A**ddress **E**xtension which is a memory management feature for the x86 architecture that extends the address space and PSE stands for **P**age **S**ize **E**xtension that refers to a feature of x86 processors that allows for pages larger than the traditional 4 KiB size.

In addition to translating a guest-physical address to a host physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called **EPT violations** and cause **VM-exits**.

Keep in mind that address **never** translates through EPT, when there is no access. That your guest-physical address is never used until there is access (Read or Write) to that location in memory.

Implementing Extended Page Table (EPT)

Now that we know some basics, let's implement what we've learned before. Based on Intel manual we should write (VMWRITE) EPTP or Extended-Page-Table Pointer to the VMCS. The EPTP structure described below.

Table 24-8. Format of Extended-Page-Table Pointer

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 28.2.6): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved.[1]
5:3	This value is 1 less than the EPT page-walk length (see Section 28.2.2)
6	Setting this control to 1 enables accessed and dirty flags for EPT (see Section 28.2.4)[2]
11:7	Reserved
N–1:12	Bits N–1:12 of the physical address of the 4-KByte aligned EPT PML4 table[3]
63:N	Reserved

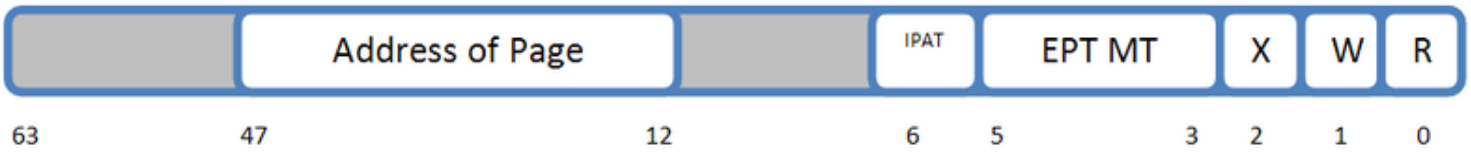
The above tables can be described using the following structure :

```
1  // See Table 24-8. Format of Extended-Page-Table Pointer
2  typedef union _EPTP {
3   ULONG64 All;
4   struct {
5   UINT64 MemoryType : 3; // bit 2:0 (0 = Uncacheable (UC) - 6 = Write - bac
6   UINT64 PageWalkLength : 3; // bit 5:3 (This value is 1 less than the EPT
7   UINT64 DirtyAndAceessEnabled : 1; // bit 6  (Setting this control to 1 en
```

```
  UINT64 Reserved1 : 5; // bit 11:7
  UINT64 PML4Address : 36;
  UINT64 Reserved2 : 16;
  }Fields;
}EPTP, *PEPTP;
```

Each entry in all EPT tables is 64 bit long. EPT PML4E and EPT PDPTE and EPT PD are the same but EPT PTE has some minor differences.

An EPT entry is something like this :



Ok, Now we should implement tables and the first table is PML4. The following table shows the format of an EPT PML4 Entry (PML4E).

Table 28-1. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed from the 512-GByte region controlled by this entry
2	If the "mode-based execute control for EPT" VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 512-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 512-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N–1):12	Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry[1]
51:N	Reserved (must be 0)
63:52	Ignored

PML4E can be a structure like this :

```
// See Table 28-1.
typedef union _EPT_PML4E {
 ULONG64 All;
 struct {
 UINT64 Read : 1; // bit 0
 UINT64 Write : 1; // bit 1
 UINT64 Execute : 1; // bit 2
 UINT64 Reserved1 : 5; // bit 7:3 (Must be Zero)
```

```
 9   UINT64 Accessed : 1; // bit 8
10   UINT64 Ignored1 : 1; // bit 9
11   UINT64 ExecuteForUserMode : 1; // bit 10
12   UINT64 Ignored2 : 1; // bit 11
13   UINT64 PhysicalAddress : 36; // bit (N-1):12 or Page-Frame-Number
14   UINT64 Reserved2 : 4; // bit 51:N
15   UINT64 Ignored3 : 12; // bit 63:52
16   }Fields;
17 }EPT_PML4E, *PEPT_PML4E;
```

As long as we want to have a 4-level paging, the second table is EPT Page-Directory-Pointer-Table (PDTP), the following picture illustrates the format of PDPTE :

Table 28-3. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that References an EPT Page Directory

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 1-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed from the 1-GByte region controlled by this entry
2	If the "mode-based execute control for EPT" VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte region controlled by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N–1):12	Physical address of 4-KByte aligned EPT page directory referenced by this entry[1]
51:N	Reserved (must be 0)
63:52	Ignored

PDPTE's structure is like this :

```
 1 // See Table 28-3
 2 typedef union _EPT_PDPTE {
 3  ULONG64 All;
 4  struct {
 5  UINT64 Read : 1; // bit 0
 6  UINT64 Write : 1; // bit 1
 7  UINT64 Execute : 1; // bit 2
 8  UINT64 Reserved1 : 5; // bit 7:3 (Must be Zero)
 9  UINT64 Accessed : 1; // bit 8
10  UINT64 Ignored1 : 1; // bit 9
11  UINT64 ExecuteForUserMode : 1; // bit 10
12  UINT64 Ignored2 : 1; // bit 11
13  UINT64 PhysicalAddress : 36; // bit (N-1):12 or Page-Frame-Number
14  UINT64 Reserved2 : 4; // bit 51:N
15  UINT64 Ignored3 : 12; // bit 63:52
16  }Fields;
```

```
17 }EPT_PDPTE, *PEPT_PDPTE;
```

For the third table of paging we should implement an EPT Page-Directory Entry (PDE) as described below:

Table 28-5. Format of an EPT Page-Directory Entry (PDE) that References an EPT Page Table

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 2-MByte region controlled by this entry
1	Write access; indicates whether writes are allowed from the 2-MByte region controlled by this entry
2	If the "mode-based execute control for EPT" VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte region controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte region controlled by this entry
6:3	Reserved (must be 0)
7	Must be 0 (otherwise, this entry maps a 2-MByte page)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte region controlled by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
9	Ignored
10	Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte region controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N–1):12	Physical address of 4-KByte aligned EPT page table referenced by this entry[1]
51:N	Reserved (must be 0)
63:52	Ignored

PDE's structure:

```
1  // See Table 28-5
2  typedef union _EPT_PDE {
3   ULONG64 All;
4   struct {
5   UINT64 Read : 1; // bit 0
6   UINT64 Write : 1; // bit 1
7   UINT64 Execute : 1; // bit 2
8   UINT64 Reserved1 : 5; // bit 7:3 (Must be Zero)
9   UINT64 Accessed : 1; // bit 8
10  UINT64 Ignored1 : 1; // bit 9
11  UINT64 ExecuteForUserMode : 1; // bit 10
12  UINT64 Ignored2 : 1; // bit 11
13  UINT64 PhysicalAddress : 36; // bit (N-1):12 or Page-Frame-Number
14  UINT64 Reserved2 : 4; // bit 51:N
15  UINT64 Ignored3 : 12; // bit 63:52
16   }Fields;
17 }EPT_PDE, *PEPT_PDE;
```

The last page is EPT which is described below.

Table 28-6. Format of an EPT Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry
1	Write access; indicates whether writes are allowed from the 4-KByte page referenced by this entry
2	If the "mode-based execute control for EPT" VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 4-KByte page controlled by this entry
5:3	EPT memory type for this 4-KByte page (see Section 28.2.6)
6	Ignore PAT memory type for this 4-KByte page (see Section 28.2.6)
7	Ignored
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
9	If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
10	Execute access for user-mode linear addresses. If the "mode-based execute control for EPT" VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 4-KByte page controlled by this entry. If that control is 0, this bit is ignored.
11	Ignored
(N–1):12	Physical address of the 4-KByte page referenced by this entry[1]
51:N	Reserved (must be 0)
62:52	Ignored
63	Suppress #VE. If the "EPT-violation #VE" VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 25.5.6.1). If "EPT-violation #VE" VM-execution control is 0, this bit is ignored.

PTE will be :

Note that you have, EPTMemoryType, IgnorePAT, DirtyFlag and SuppressVE in addition to the above pages.

```
// See Table 28-6
typedef union _EPT_PTE {
 ULONG64 All;
 struct {
 UINT64 Read : 1; // bit 0
 UINT64 Write : 1; // bit 1
 UINT64 Execute : 1; // bit 2
 UINT64 EPTMemoryType : 3; // bit 5:3 (EPT Memory type)
 UINT64 IgnorePAT : 1; // bit 6
 UINT64 Ignored1 : 1; // bit 7
 UINT64 AccessedFlag : 1; // bit 8
 UINT64 DirtyFlag : 1; // bit 9
 UINT64 ExecuteForUserMode : 1; // bit 10
 UINT64 Ignored2 : 1; // bit 11
 UINT64 PhysicalAddress : 36; // bit (N-1):12 or Page-Frame-Number
 UINT64 Reserved : 4; // bit 51:N
 UINT64 Ignored3 : 11; // bit 62:52
 UINT64 SuppressVE : 1; // bit 63
 }Fields;
}EPT_PTE, *PEPT_PTE;
```

There are other types of implementing page walks (2 or 3 level paging) and if you set the 7th bit of PDPTE (Maps 1 GB) or the 7th bit of PDE (Maps 2 MB) so instead of implementing 4 level paging (like what we want to do for the rest of the topic) you set those bits but keep in mind that the corresponding tables are different. These tables described in (Table 28-4. Format of an EPT Page-Directory Entry (PDE) that Maps a 2-MByte Page) and (Table 28-2. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page). Alex Ionescu's SimpleVisor is an example of implementing in this way.

An important note is almost all the above structures have a 36-bit Physical Address which means our hypervisor supports only 4-level paging. It is because every page table (and every EPT Page Table) consist of 512 entries which means you need 9 bits to select an entry and as long as we have 4 level tables, we can't use more than 36 (4 * 9) bits. Another method with wider address range is not implemented in all major OS like Windows or Linux. I'll describe EPT PML5E briefly later in this topic but we don't implement it in our hypervisor as it's not popular yet!

By the way, N is the physical-address width supported by the processor. CPUID with 80000008H in EAX gives you the supported width in EAX bits 7:0.

Let's see the rest of the code, the following code is the **Initialize_EPTP** function which is responsible for allocating and mapping EPTP.

Note that the **PAGED_CODE()** macro ensures that the calling thread is running at an IRQL that is low enough to permit paging.

```
1 UINT64 Initialize_EPTP()
2 {
3  PAGED_CODE();
4         ...
```

First of all, allocating EPTP and put zeros on it.

```
1  // Allocate EPTP
2  PEPTP EPTPointer = ExAllocatePoolWithTag(NonPagedPool, PAGE_SIZE, POOLTAG)
3
4  if (!EPTPointer) {
5  return NULL;
6  }
7  RtlZeroMemory(EPTPointer, PAGE_SIZE);
```

Now, we need a blank page for our EPT PML4 Table.

```
// Allocate EPT PML4
PEPT_PML4E EPT_PML4 = ExAllocatePoolWithTag(NonPagedPool, PAGE_SIZE, POOL
if (!EPT_PML4) {
ExFreePoolWithTag(EPTPointer, POOLTAG);
return NULL;
}
RtlZeroMemory(EPT_PML4, PAGE_SIZE);
```

And another empty page for PDPT.

```
//  Allocate EPT Page-Directory-Pointer-Table
PEPT_PDPTE EPT_PDPT = ExAllocatePoolWithTag(NonPagedPool, PAGE_SIZE, POOL
if (!EPT_PDPT) {
ExFreePoolWithTag(EPT_PML4, POOLTAG);
ExFreePoolWithTag(EPTPointer, POOLTAG);
return NULL;
}
RtlZeroMemory(EPT_PDPT, PAGE_SIZE);
```

Of course its true about Page Directory Table.

```
// Allocate EPT Page-Directory
PEPT_PDE EPT_PD = ExAllocatePoolWithTag(NonPagedPool, PAGE_SIZE, POOLTAG)

if (!EPT_PD) {
ExFreePoolWithTag(EPT_PDPT, POOLTAG);
ExFreePoolWithTag(EPT_PML4, POOLTAG);
ExFreePoolWithTag(EPTPointer, POOLTAG);
return NULL;
}
RtlZeroMemory(EPT_PD, PAGE_SIZE);
```

The last table is a blank page for EPT Page Table.

```
// Allocate EPT Page-Table
PEPT_PTE EPT_PT = ExAllocatePoolWithTag(NonPagedPool, PAGE_SIZE, POOLTAG)

if (!EPT_PT) {
ExFreePoolWithTag(EPT_PD, POOLTAG);
ExFreePoolWithTag(EPT_PDPT, POOLTAG);
ExFreePoolWithTag(EPT_PML4, POOLTAG);
ExFreePoolWithTag(EPTPointer, POOLTAG);
return NULL;
}
RtlZeroMemory(EPT_PT, PAGE_SIZE);
```

Now that we have all of our pages available, let's allocate two page (2*4096) continuously because we need one of the pages for our RIP to start and one page for our Stack (RSP). After that, we need two EPT Page Table Entries (PTEs) with

permission to **execute**, **read**, **write**. The physical address should be divided by 4096 (PAGE_SIZE) because if we dived a hex number by 4096 (0x1000) 12 digits from the right (which are zeros) will disappear and these 12 digits are for choosing between 4096 bytes.

By the way, we let stack be executable too and that's because, in a regular VM, we should put RWX to all pages because its the responsibility of internal page tables to set or clear NX bit. We need to change them from EPT Tables for special purposes (e.g intercepting instruction fetch for a special page). Changing from EPT tables will lead to EPT-Violation, in this way we can intercept these events.

The actual need is two page but we need to build page tables inside our guest software thus we allocate up to 10 page.

I'll explain about intercepting pages from EPT, later in these series.

```
// Setup PT by allocating two pages Continuously
// We allocate two pages because we need 1 page for our RIP to start and

const int PagesToAllocate = 10;
UINT64 Guest_Memory = ExAllocatePoolWithTag(NonPagedPool, PagesToAllocate
RtlZeroMemory(Guest_Memory, PagesToAllocate * PAGE_SIZE);

for (size_t i = 0; i < PagesToAllocate; i++)
{
EPT_PT[i].Fields.AccessedFlag = 0;
EPT_PT[i].Fields.DirtyFlag = 0;
EPT_PT[i].Fields.EPTMemoryType = 6;
EPT_PT[i].Fields.Execute = 1;
EPT_PT[i].Fields.ExecuteForUserMode = 0;
EPT_PT[i].Fields.IgnorePAT = 0;
EPT_PT[i].Fields.PhysicalAddress = (VirtualAddress_to_PhysicalAddress( Gu
EPT_PT[i].Fields.Read = 1;
EPT_PT[i].Fields.SuppressVE = 0;
EPT_PT[i].Fields.Write = 1;

}
```

Note: **EPTMemoryType** can be either 0 (for uncached memory) or 6 (write-back) memory and as we want our memory to be cacheable so put 6 on it.

The next table is PDE. PDE should point to PTE base address so we just put the address of the first entry from the EPT PTE as the physical address for Page Directory Entry.

```
// Setting up PDE
```

```
EPT_PD->Fields.Accessed = 0;
EPT_PD->Fields.Execute = 1;
EPT_PD->Fields.ExecuteForUserMode = 0;
EPT_PD->Fields.Ignored1 = 0;
EPT_PD->Fields.Ignored2 = 0;
EPT_PD->Fields.Ignored3 = 0;
EPT_PD->Fields.PhysicalAddress = (VirtualAddress_to_PhysicalAddress(EPT_P
EPT_PD->Fields.Read = 1;
EPT_PD->Fields.Reserved1 = 0;
EPT_PD->Fields.Reserved2 = 0;
EPT_PD->Fields.Write = 1;
```

Next step is mapping PDPT. PDPT Entry should point to the first entry of Page-Directory.

```
// Setting up PDPTE
EPT_PDPT->Fields.Accessed = 0;
EPT_PDPT->Fields.Execute = 1;
EPT_PDPT->Fields.ExecuteForUserMode = 0;
EPT_PDPT->Fields.Ignored1 = 0;
EPT_PDPT->Fields.Ignored2 = 0;
EPT_PDPT->Fields.Ignored3 = 0;
EPT_PDPT->Fields.PhysicalAddress = (VirtualAddress_to_PhysicalAddress(EPT
EPT_PDPT->Fields.Read = 1;
EPT_PDPT->Fields.Reserved1 = 0;
EPT_PDPT->Fields.Reserved2 = 0;
EPT_PDPT->Fields.Write = 1;
```

The last step is configuring PML4E which points to the first entry of the PTPT.

```
// Setting up PML4E
EPT_PML4->Fields.Accessed = 0;
EPT_PML4->Fields.Execute = 1;
EPT_PML4->Fields.ExecuteForUserMode = 0;
EPT_PML4->Fields.Ignored1 = 0;
EPT_PML4->Fields.Ignored2 = 0;
EPT_PML4->Fields.Ignored3 = 0;
EPT_PML4->Fields.PhysicalAddress = (VirtualAddress_to_PhysicalAddress(EPT
EPT_PML4->Fields.Read = 1;
EPT_PML4->Fields.Reserved1 = 0;
EPT_PML4->Fields.Reserved2 = 0;
EPT_PML4->Fields.Write = 1;
```

We’ve almost done! Just set up the EPTP for our VMCS by putting 0x6 as the memory type (which is write-back) and we walk 4 times so the page walk length is 4-1=3 and PML4 address is the physical address of the first entry in the PML4 table.

I’ll explain about DirtyAndAcessEnabled field later in this topic.

```
// Setting up EPTP
EPTPointer->Fields.DirtyAndAceessEnabled = 1;
EPTPointer->Fields.MemoryType = 6; // 6 = Write-back (WB)
```

```
EPTPointer->Fields.PageWalkLength = 3;  // 4 (tables walked) - 1 = 3
EPTPointer->Fields.PML4Address = (VirtualAddress_to_PhysicalAddress(EPT_PM
EPTPointer->Fields.Reserved1 = 0;
EPTPointer->Fields.Reserved2 = 0;
```

and the last step.

```
DbgPrint("[*] Extended Page Table Pointer allocated at %llx",EPTPointer);
return EPTPointer;
```

All the above page tables should be aligned to 4KByte boundaries but as long as we allocate >= PAGE_SIZE (One PFN record) so it's automatically 4kb-aligned.

Our implementation consist of 4 tables, therefore, the full layout is like this:

63–52	51–M[1]	M-1–32	31–12	11–10	9–8	7	6	5–3	2	1	0	
Reserved			Address of EPT PML4 table	Rsvd.			A/D	EPT PWL-1	EPT PSMT			EPTP[2]
Ignored		Rsvd.	Address of EPT page-directory-pointer table	Ign.	XU[3]	Ign.	A	Reserved	X[4]	W	R	PML4E: present[5]
SVE[6]	Ignored								0	0	0	PML4E: not present
SVE	Ignored	Rsvd.	Physical address of 1GB page	Reserved	Ign.	XU	D	A	1	IPAT	EPT MT	PDPTE: 1GB page
Ignored		Rsvd.	Address of EPT page directory		Ign.	XU	Ign.	A	0	Rsvd.		PDPTE: page directory
SVE	Ignored								0	0	0	PDTPE: not present
SVE	Ignored	Rsvd.	Physical address of 2MB page	Reserved	Ign.	XU	D	A	1	IPAT	EPT MT	PDE: 2MB page
Ignored		Rsvd.	Address of EPT page table		Ign.	XU	Ign.	A	0	Rsvd.		PDE: page table
SVE	Ignored								0	0	0	PDE: not present
SVE	Ignored	Rsvd.	Physical address of 4KB page		Ign.	XU	D	A	Ign	IPAT	EPT MT	PTE: 4KB page
SVE	Ignored								0	0	0	PTE: not present

Figure 28-1. Formats of EPTP and EPT Paging-Structure Entries

Accessed and Dirty Flags in EPTP

In EPTP, you'll decide whether enable accessed and dirty flags for EPT or not using the 6th bit of the extended-page-table pointer (EPTP). Setting this flag causes processor accesses to guest paging structure entries to be treated as writes.

For any EPT paging-structure entry that is used during guest-physical-address translation, bit 8 is the accessed flag. For an EPT paging-structure entry that maps a page (as opposed to referencing another EPT paging structure), bit 9 is the dirty flag.

Whenever the processor uses an EPT paging-structure entry as part of the guest-physical-address translation, it sets the accessed flag in that entry (if it is not already set).

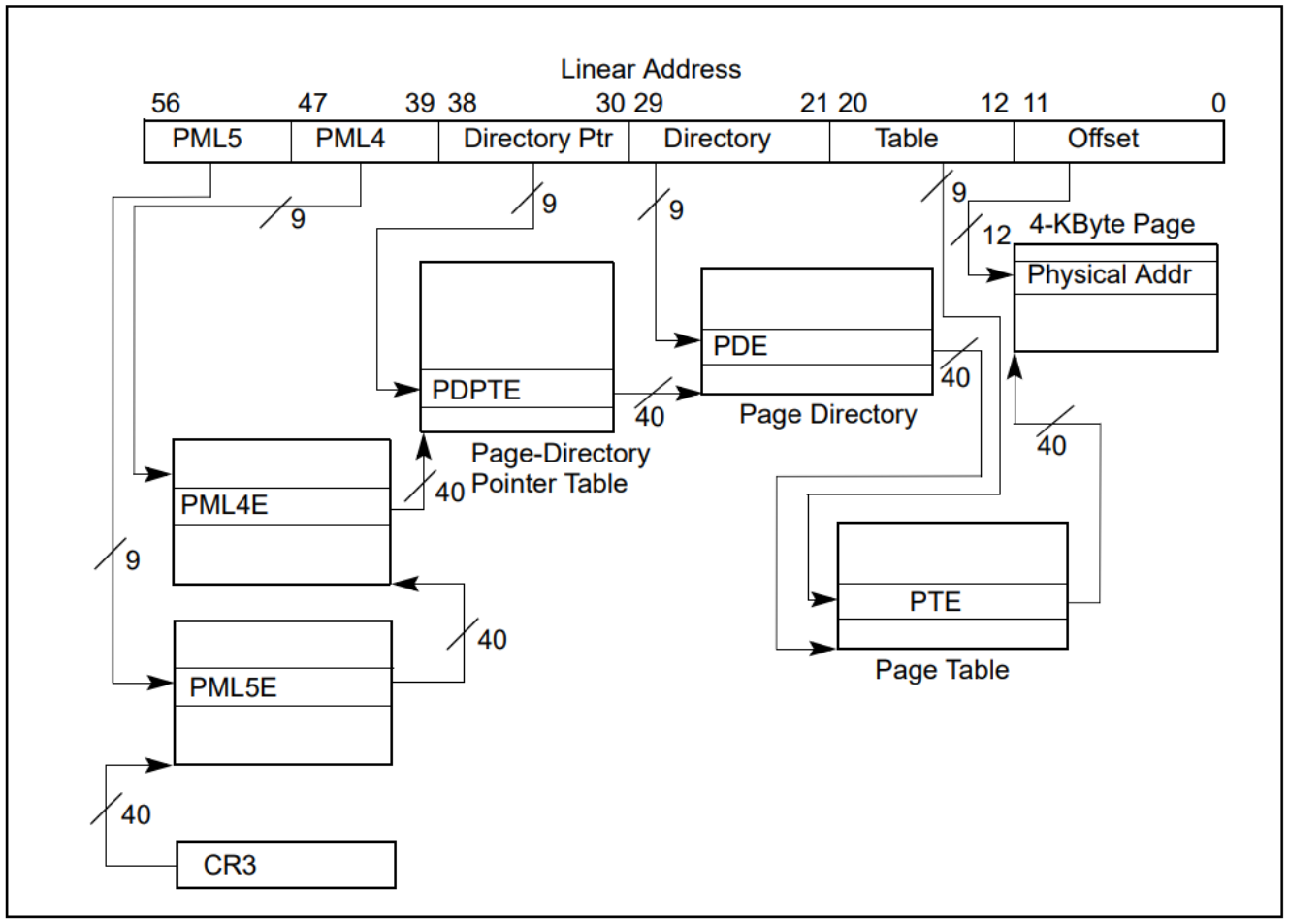
Whenever there is a write to a guest-physical address, the processor sets the dirty flag (if it is not already set) in the EPT paging-structure entry that identifies the final physical address for the guest-physical address (either an EPT PTE or an EPT paging-structure entry in which bit 7 is 1).

These flags are “sticky,” meaning that, once set, the processor does not clear them; only software can clear them.

5-Level EPT Translation

Intel suggests a new table in translation hierarchy, called PML5 which extends the EPT into a 5-layer table and guest operating systems can use up to 57 bit for the virtual-addresses while the classic 4-level EPT is limited to translating 48-bit guest-physical
addresses. None of the modern OSs use this feature yet.

PML5 is also applying to both EPT and regular paging mechanism.



Translation begins by identifying a 4-KByte naturally aligned EPT PML5 table. It is located at the physical address specified in bits 51:12 of EPTP. An EPT PML5 table comprises 512 64-bit entries (EPT PML5Es). An EPT PML5E is selected using the physical address defined as follows.

- Bits 63:52 are all 0.
- Bits 51:12 are from EPTP.
- Bits 11:3 are bits 56:48 of the guest-physical address.
- Bits 2:0 are all 0.
- Because an EPT PML5E is identified using bits 56:48 of the guest-physical address, it controls access to a 256-TByte region of the linear address space.

The only difference is you should put PML5 physical address instead of the PML4 address in EPTP.

For more information about 5-layer paging take a look at this Intel documentation.

Invalidating Cache (INVEPT)

Well, Intel's explanation about Cache invalidating is really vague and I couldn't understand it completely but I asked Petr and he explains me in this way:

- VMX-specific TLB-management instructions:

    - **INVEPT** – Invalidate cached Extended Page Table (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.

    - **INVVPID** – Invalidate cached mappings of address translation based on the Virtual Processor ID (VPID).

Imagine we access guest-physical-address 0x1000,it'll get translated to host-physical-address 0x5000. Next time, if we access 0x1000, the CPU won't send the request to the memory bus but uses cached memory instead. it's faster. Now let's say we change EPT_PDPT->PhysicalAddress to point to different EPT PD or change the attributes of one of the EPT tables, now we have to tell the processor that your cache is invalid and that's what exactly INVEPT performs.

Now we have two terms here, **Single-Context** and **All-Context**.

Single-Context means, that you invalidate all EPT-derived translations based on a single EPTP (in short: for single VM).

All-Context means that you invalidate all EPT-derived translations. (for every-VM).

So in case if you wouldn't perform INVEPT after changing EPT's structures, you would be risking that the CPU would reuse old translations.

Basically, any change to EPT structure needs INVEPT but switching EPT (or VMCS) doesn't need INVEPT because that translation will be "tagged" with the changed EPTP in the cache.

The following assembly function is responsible for INVEPT.

```
1  INVEPT_Instruction PROC PUBLIC
2          invept  rcx, oword ptr [rdx]
3          jz @jz
4          jc @jc
5          xor     rax, rax
6          ret
7
8  @jz:    mov     rax, VMX_ERROR_CODE_FAILED_WITH_STATUS
9          ret
```

```
10
11 @jc:    mov     rax, VMX_ERROR_CODE_FAILED
12         ret
13 INVEPT_Instruction ENDP
```

Note that **VMX_ERROR_CODE_FAILED_WITH_STATUS** and **VMX_ERROR_CODE_FAILED** define like this.

```
1     VMX_ERROR_CODE_SUCCESS               = 0
2     VMX_ERROR_CODE_FAILED_WITH_STATUS    = 1
3     VMX_ERROR_CODE_FAILED                = 2
```

Now, we implement INVEPT.

```
unsigned char INVEPT(UINT32 type, INVEPT_DESC* descriptor)
{
 if (!descriptor)
 {
 static INVEPT_DESC zero_descriptor = { 0 };
 descriptor = &zero_descriptor;
 }

 return INVEPT_Instruction(type, descriptor);
}
```

To invalidate all the contexts use the following function.

```
unsigned char INVEPT_ALL_CONTEXTS()
{
 return INVEPT(all_contexts ,NULL);
}
```

And the last step is for Single-Context INVEPT which needs an EPTP.

```
unsigned char INVEPT_SINGLE_CONTEXT(EPTP ept_pointer)
{
 INVEPT_DESC descriptor = { ept_pointer, 0 };
 return INVEPT(single_context, &descriptor);
}
```

Using the above functions in a modification state, tell the processor to invalidate its cache.

Conclusion

In this part, we see how to initialize the Extended Page Table and map guest physical address to host physical address then we build the EPTP based on the allocated

addresses.

The future part would be about building the VMCS and implementing other VMX instructions. Don’t forget to check the blog for the future posts.

Have a good time!

References

[1] Vol 3C – 28.2 THE EXTENDED PAGE TABLE MECHANISM (EPT) (https://software.intel.com/en-us/articles/intel-sdm)

[2] Performance Evaluation of Intel EPT Hardware Assist (https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf)

[3] Second Level Address Translation (https://en.wikipedia.org/wiki/Second_Level_Address_Translation)

[4] Memory Virtualization (http://www.cs.nthu.edu.tw/~ychung/slides/Virtualization/VM-Lecture-2-2-SystemVirtualizationMemory.pptx)

[5] Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d (https://software.intel.com/en-us/articles/best-practices-

for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d)

[6] 5-Level Paging and 5-Level EPT (https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf)

[7] Xen Summit November 2007 – Jun Nakajima (http://www-archive.xenproject.org/files/xensummit_fall07/12_JunNakajima.pdf)

[8] gipervizor against rutkitov: as it works (http://developers-club.com/posts/133906/)

[9] Intel SGX Explained (https://www.semanticscholar.org/paper/Intel-SGX-Explained-Costan-Devadas/2d7f3f4ca3fbb15ae04533456e5031e0d0dc845a)

[10] Intel VT-x (https://github.com/tnballo/notebook/wiki/Intel-VTx)

[11] Introduction to IA-32e hardware paging (https://www.triplefault.io/2017/07/introduction-to-ia-32e-hardware-paging.html)

PAGES

Blog Map

Tools & Scripts

Tutorials

Sinaei

Judas tree , What kind of mystery is this, that every spring, Comes with our hearts' mourning, Judas tree, You be elate, You sing my unsang song...

Published in **CPU**, **Hypervisor** and **Tutorials**

All-Context | EPT | EPTP | Extended Page Table | Extended Page Table Pointer | hypervisor paging | INVEPT | Nested Page Tables | NPT | Rapid Virtualization Indexing | RVI | Second Level Address Translation | Single-Context | SLAT | Stage-2 page-tables

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name*

Jane Doe

Email*

name@email.com

Website

http://google.com

Post Comment

Search …

Search

RECENT POSTS

Hypervisor From Scratch – Part 4: Address Translation Using Extended Page Table (EPT)

Hypervisor From Scratch – Part 3: Setting up Our First Virtual Machine

Using Intel's Streaming SIMD Extensions 3 (MONITOR\MWAIT) As A Kernel Debugging Trick

Hypervisor From Scratch – Part 2: Entering VMX Operation

A Tour of Mount in Linux

RECENT COMMENTS

IRQL_EQUALITY on *Hypervisor From Scratch – Part 3: Setting up Our First Virtual Machine*

Kasbarg (Shayan) on *Hypervisor From Scratch – Part 1: Basic Concepts & Configure Testing Environment*

Sinaei on *Hypervisor From Scratch – Part 2: Entering VMX Operation*

Carl OS on *Hypervisor From Scratch – Part 1: Basic Concepts & Configure Testing Environment*

Necrolis on *Hypervisor From Scratch – Part 2: Entering VMX Operation*

ARCHIVES

October 2018

September 2018

August 2018

July 2018

June 2018

May 2018

April 2018

March 2018

January 2018

December 2017

November 2017

October 2017

September 2017

August 2017

April 2017

March 2017

CATEGORIES

.Net Framework

Android

Cisco

CPU

Debugging

Emulator

Hypervisor

Instrumentation

Kernel Mode

Linux

Malware

Network

Pentest

Programming

Ransomware

Security

Social

Software

SysAdmin

Tutorials

User Mode

Windows

TAGS

active directory Assembly x64 Visual Studio begining cache cisco Create a virtual machine debian debugging kernel mode debug virtual machine debug windows getting started with pykd helloworld How to create Virtual Machine Hypervisor fundamentals Hypervisor Tutorials Intel Virtualization Intel VMX Intel VTX Tutorial ios ipsec kernel-mode linux network opensource Page management in Windows PFN PFN Database proxy PyKD example PyKD sample PyKD scripts PyKD tutorial run PyKD command Setting up Virtual Machine Monitor start systemd tunnel using CPU Virtualization VMCS VMM Implementation VMM Tutorials VMWare and Windbg windows server x64 assembly in driver _MMPFN