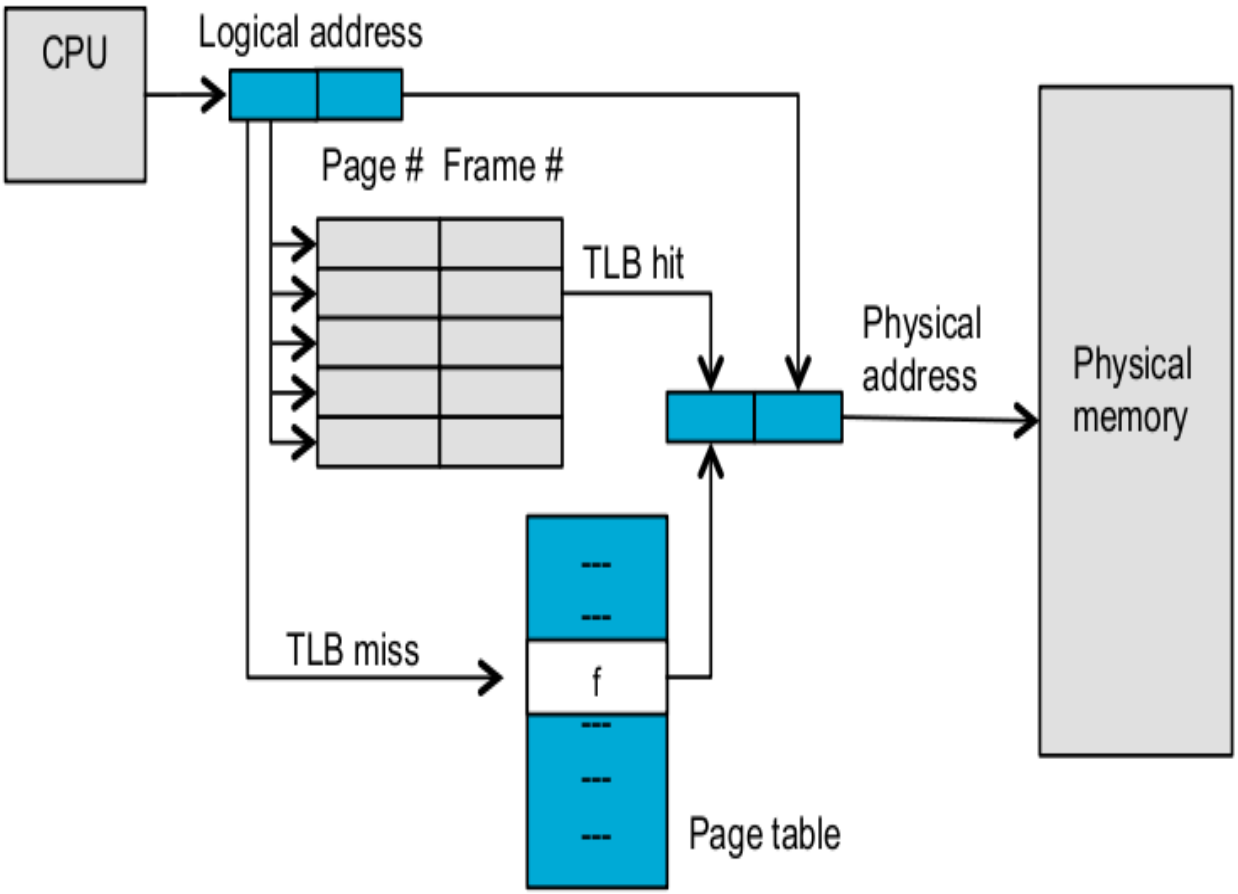
Memory Management

General Paging Unit



1:-Logical Address is divided into two parts such that

Logical Address = Page_# + Page_offset

2:-Page_# is looked up into the TLB if matched then it is called TLB hit otherwise TLB miss.

3:-If TLB hit is occurred,the relevant Frame_# and Page_offset makes Physical Address.

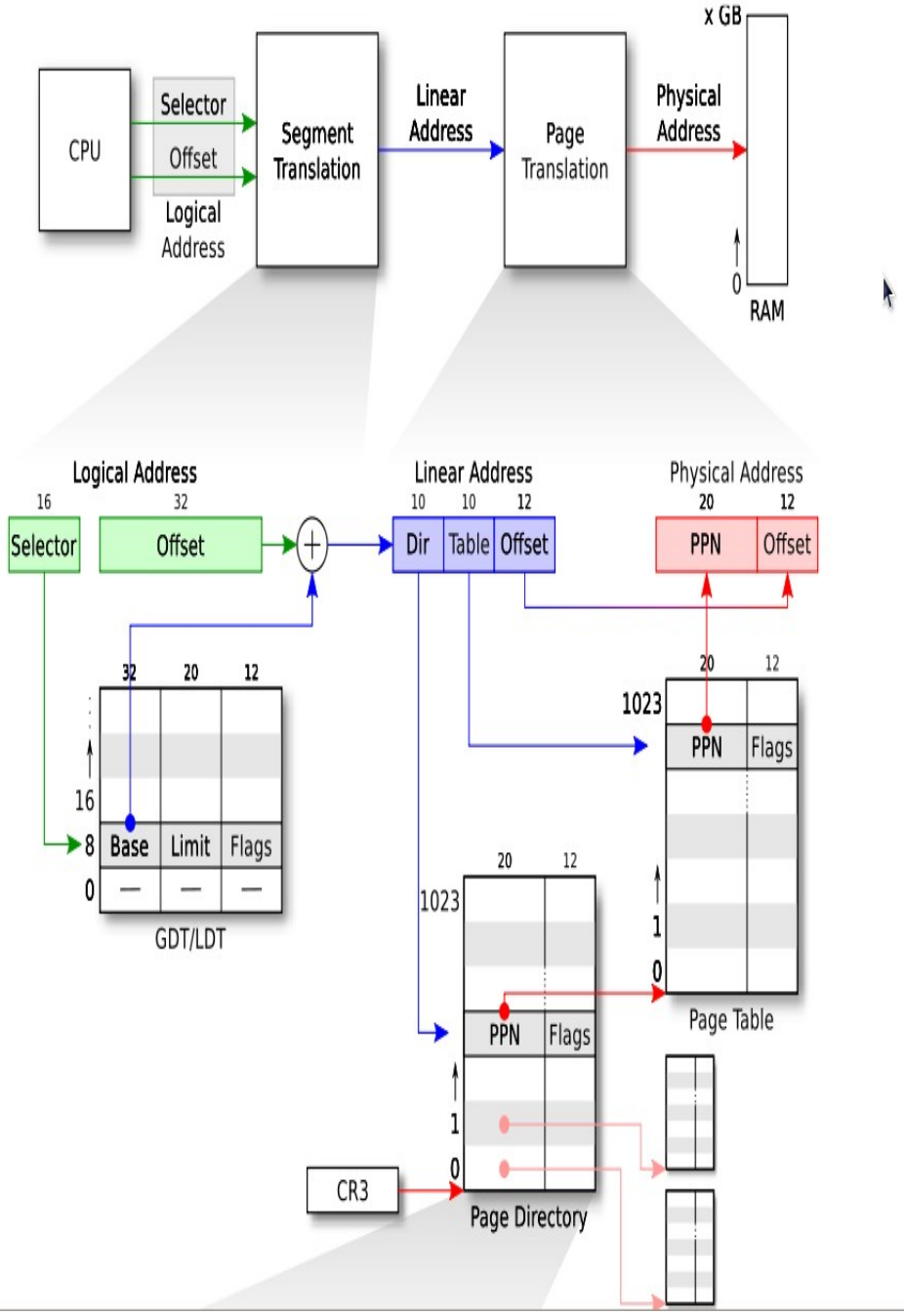
Physical Address = Frame_# + Page_offset

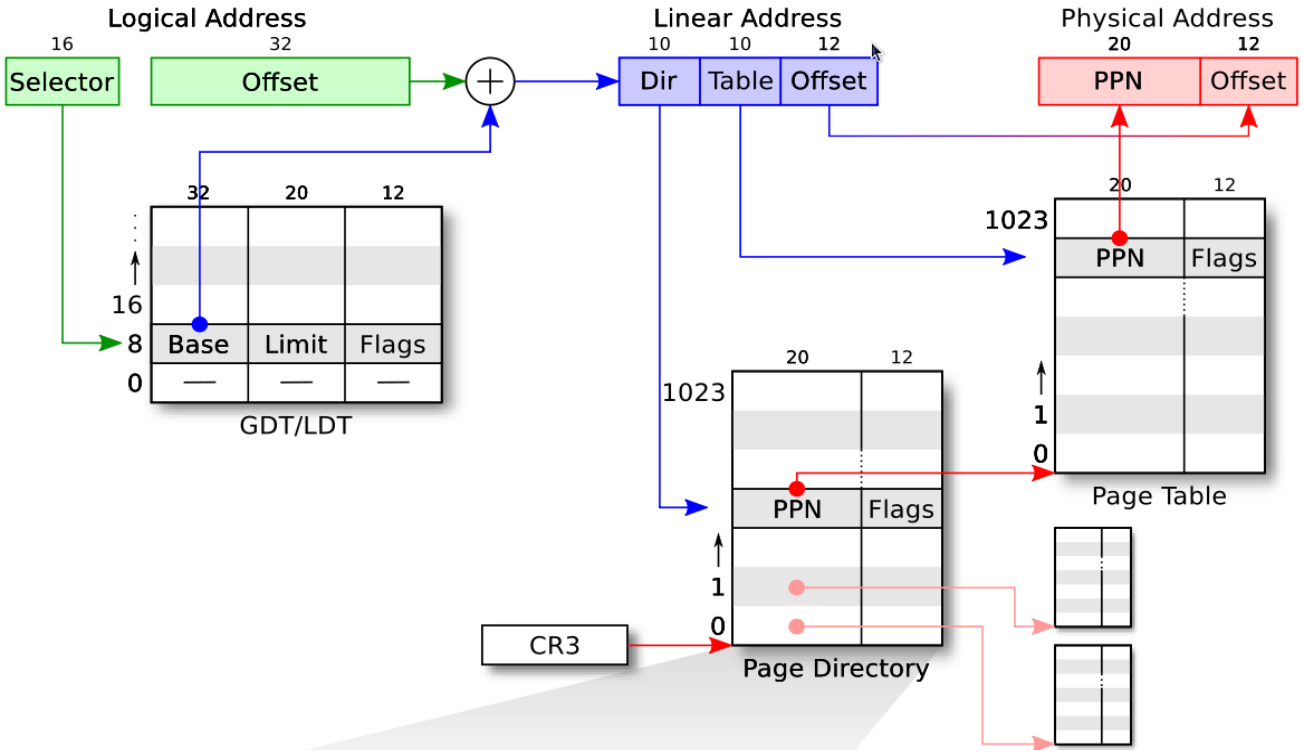
4:-On TLB miss the Page_table is referenced to find Frame_# of the relevant Page_# entry.

5:-Finally,by using Physical Address the relevant data is accessed from Cache and Main Memory orderly.

x86 Virtual to Physical Address Translation

x86 address translation





1.CPU generates Logical Address,which is first converted into Linear Address by Segmentation Unit.

Logical Address = Segment_# + Segment_offset

2.Segment_# select the segment_descriptor table(Global or Local Descriptor table,it's address is stored in gdtr and ldtr registers respectively) entry. The formate of the entry is given in the figure.

3.Segment_offset is first checked whether it is less than the limit field as provided in the segment entry,if yes then base plus offset makes Linear Address. Otherwise segmentation fault error is raised.

Linear Address = Base (in GDT/LDT entry) + Segment_offset

4.The Linear Address then divided into three fields

Linear Address = Dir_offset + Table_offset + Page_offset

5.The Page Directory base address is stored in CR3 register.

6.The Physical Address is then calculated as fellows:

Physical Address = Page_Table[Page_Dir[CR3 + Dir_offset] + Table_offset] + Page_offset

The following figure shows the complete segmentation and paging address translation of x86 architecture.

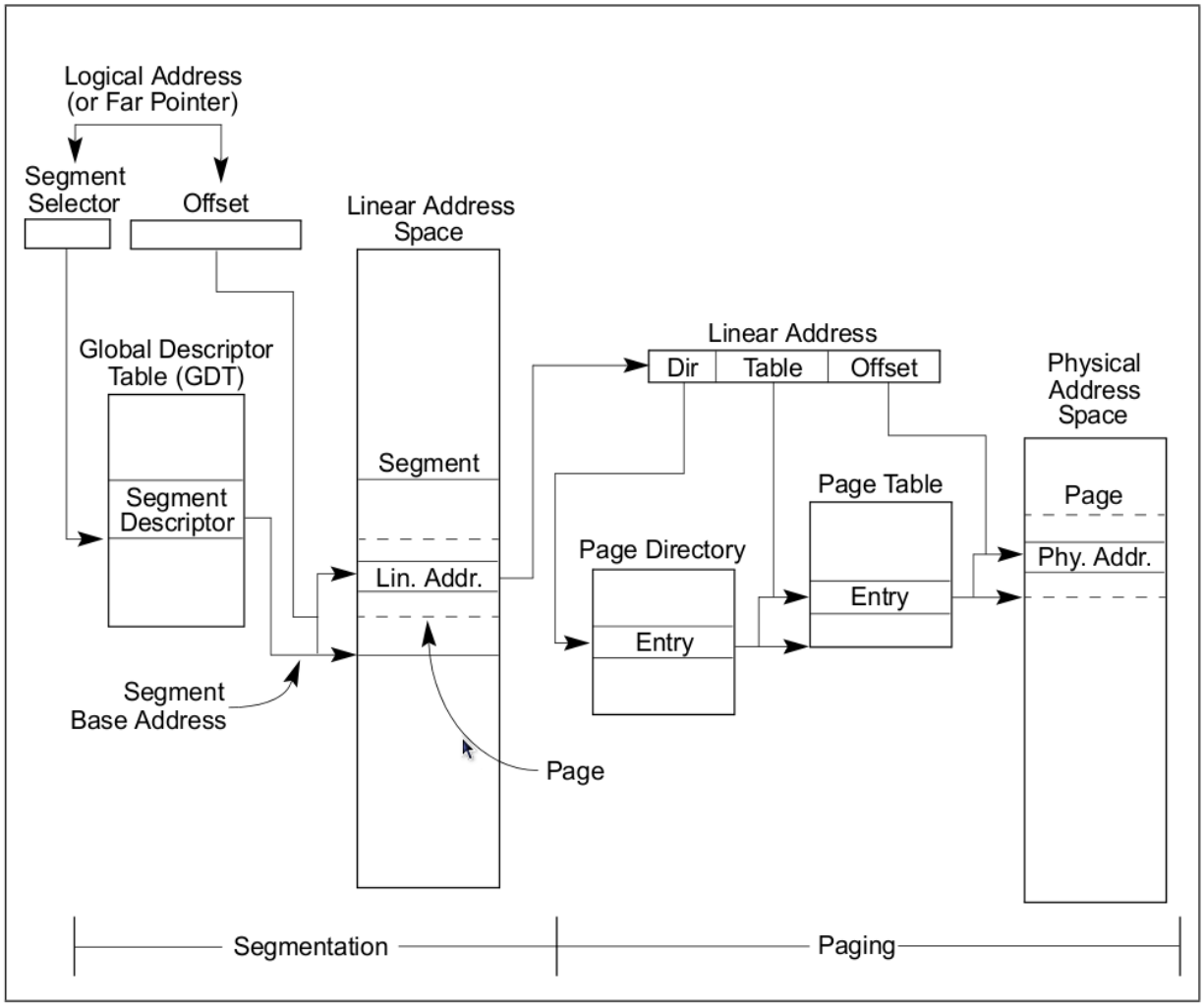


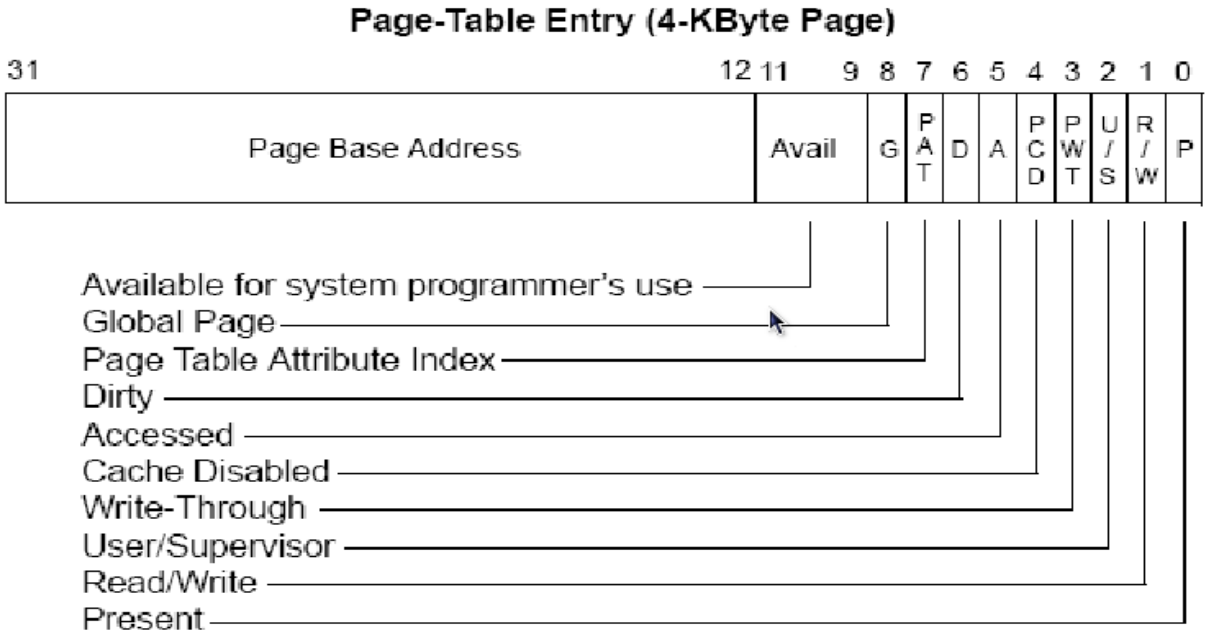
Figure 3-1. Segmentation and Paging

All Segments and Page-Table descriptors are also stored into the memory.

x86 Translation Look Aside Buffer

1. TLB managed by Processor and OS.
2. CPU fills TLB on demand from Page table(the OS unaware of TLB misses)
3. CPU evicts entries when a new entry must be added and no free slots exist.
4. OS must ensure TLB/Page table consistency by flushing entries are needed
5. when page tables are updated or switched.
6. TLB entries can be removed by the OS one at a time using the INVLPG instruction
7. or the entire TLB can be flushed at once by writing a new entry into CR3 register.

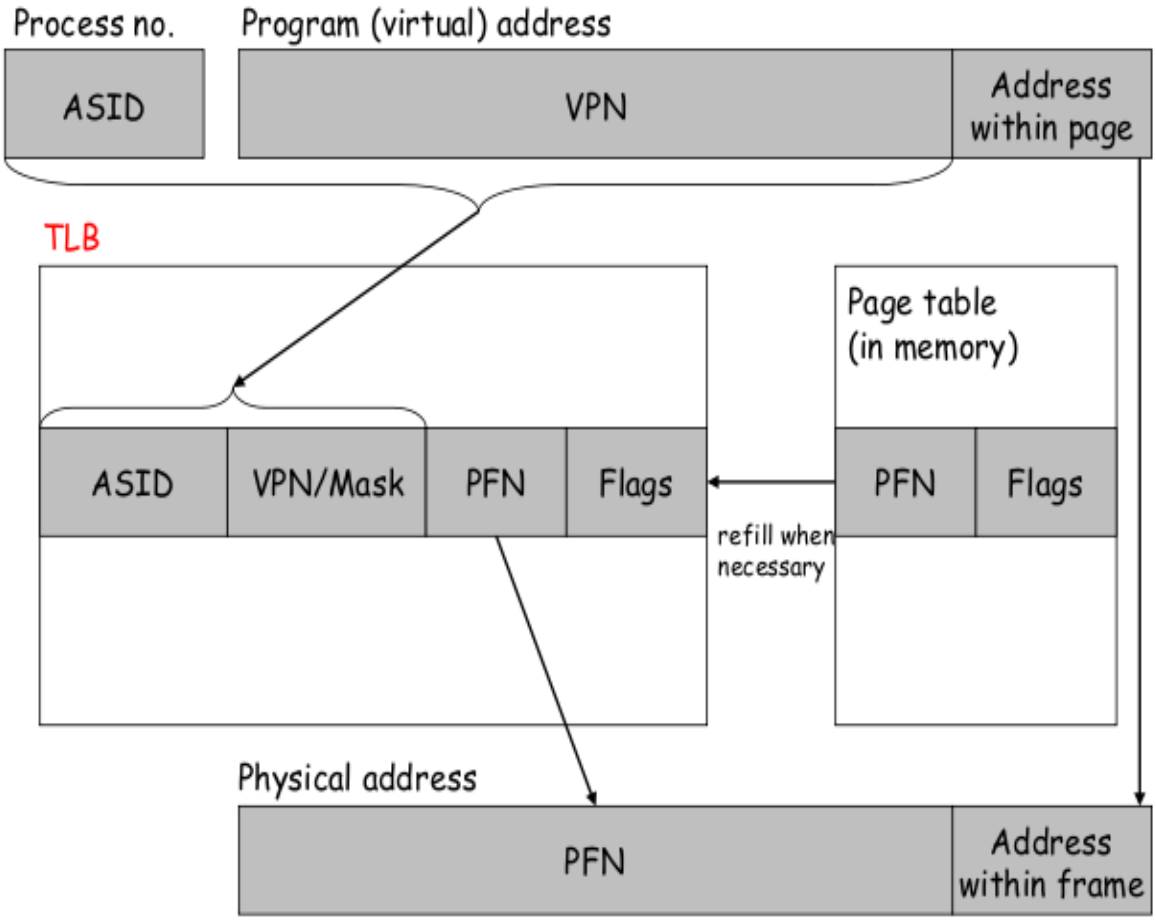
Here's is the Page table entry on which both hardware and software are agreed.



The Page table entry field “Page Base Address” gives the actual “Physical Frame No”.

Physical Address = Page Base Address + Page_offset

MIPS Paging Unit



1. The Virtual Address if divided into three fields:-

    Virtual Address = ASID(Address Space Identifier) + VPN (Virtual Page No.) + Page_offset

2. ASID + VPN are compared with TLB fields,if matched the relevant PFN is returned.

    Physical Address = PFN (Physical Frame No.) + Page_offset

MIPS TLB Entry

- *VPN*: The high order bits of the virtual address (*the virtual address of the page less low bits*). It becomes VPN2 with the double entry, to emphasize

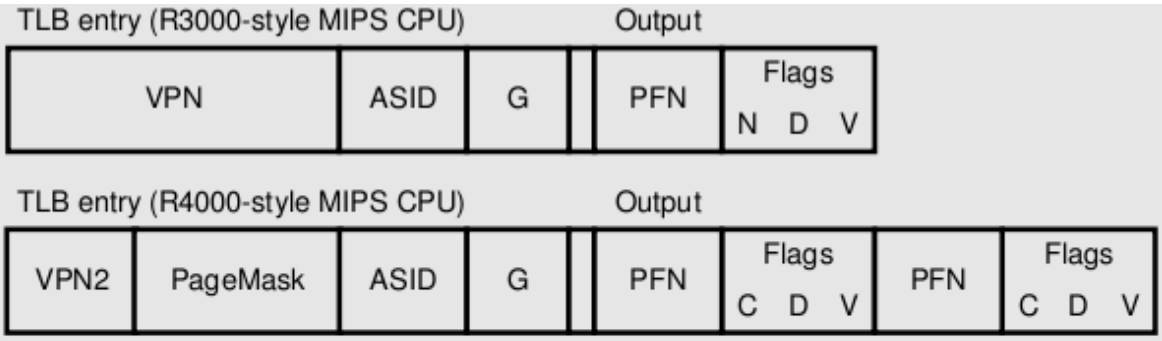


Figure 6.3: TLB entry fields

that if each physical page is 4KB, the virtual address selecting a pair of entries loses its least-significant bit (which now selects the left or right output field).

- *PageMask*: This is only found on later CPUs. It controls how much of the virtual address is compared with the VPN and how much is passed through to the physical address; a match on fewer bits maps a larger region. MIPS CPUs can be set up to map up to 16MB with a single entry. With all page sizes, the most significant masked bit is used to select the even or odd entry.

- *ASID*: Marks the translation as belonging to a particular address space, so it won't be matched unless the CPU's current ASID value matches too The G bit, if set, disables the ASID match , making the translation entry apply to all address spaces (so this part of the address map is shared between all spaces). The ASID is 6 bits long on early CPUs, 8 bits on later ones. [1]

The TLB's output side gives you the physical frame number and a small but sufficient bunch of flags:

- *Physical frame number (PFN)*: This is the physical address with the low 12 bits cut off.

- *Cadre control (N/C)*: The 32-bit CPUs have just the N (noncacheable) bit — 0 for cacheable, 1 for noncacheable.

- *Write control bit (D)*: Set 1 to allow stores to this page to happen.

- *Valid bit (V)*: If this is 0, the entry is unusable. This seems pretty pointless: Why have a record loaded into the TLB if you don't want the translation to work? It's because the software routine that refills the TLB is optimized for speed and doesn't want to check for special cases. When some further processing is needed before a program can use a page referred to by the memory-held table, the memory-held entry can be left marked invalid. After TLB refill, this will cause a different kind of trap, invoking special processing without having to put a test in every software refill event.

Translating an address is now simple, and we can amplify the description above:

- *CPU generates a program address*: This is accomplished either for an instruction fetch, a load, or for a store that doesn't lie in the special unmapped regions of the MIPS address space.

  The low 12 bits are separated off, and the resulting VPN together with the current value of the ASID field in `EntryHi` is used as the key to the TLB, as modified in effect by the `PageMask` and G fields in TLB entries.

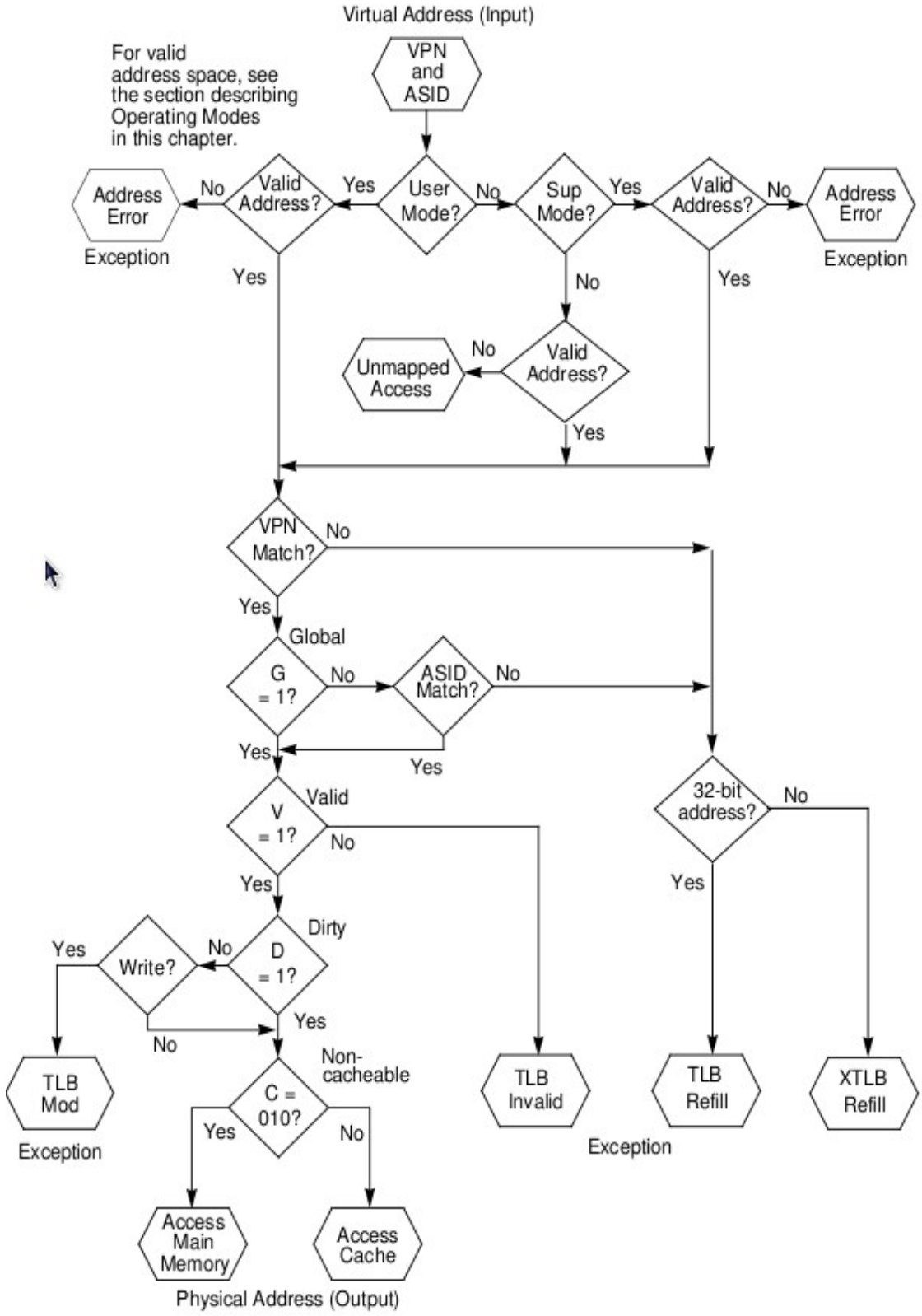
- *TLB matches key*: The matching entry is selected. The PFN is glued to the low-order bits of the program address to form a complete physical address.

- *Valid*? The V and D bits are consulted. If it isn't valid or a store is being attempted with D onset, the CPU takes a trap. As with all translation traps, the `BadVaddr` register will be filled with the offending program address; as with any TLB exception, the TLB `EntryHi` register will be preloaded with the VPN of the offending address.

  Don't use the convenience registers `Context` (and `XContext` on 64-bit CPUs) other than in TLB miss processing. At other times they might track things like `BadVaddr` or they might not; either would be a legitimate implementation.

- *Cached*? If the C bit is set the CPU looks in the cache for a copy of the physical location's data; if it isn't there it will be fetched from memory and a copy left in the cache. Where the C bit is clear the CPU neither looks in nor refills the cache.

How the Virtual Address is mapped into the TLB entry has been explained in below diagram.



CPU Control Register for Memory Management

Register mnemonic	*CP0 register number*	*Description*
EntryHi	10	Together these registers hold everything needed for a TLB entry. All reads and writes to the TLB must be staged through them. **EntryHi** holds the VPN and ASID; in MIPS32/64 CPUs, each entry maps two consecutive VPNs to different physical pages, so the PFN and access permission flags are specified independently for the two pages by registers called **EntryLo0** and **EntryLo1**. The field **EntryHi(ASID)** does double duty, since it remembers the currently active ASID. **EntryHi** grows to 64 bits in 64-bit CPUs but in such a way as to preserve the illusion of a 32-bit layout for software that doesn't need long addresses. **PageMask** can be used to create entries that map pages bigger than 4 KB; see section 6.2.1. Some CPUs support smaller page sizes, but that's barely mentioned in this book.
EntryLo0–1	2–3	
PageMask	5	
Index	0	This determines which TLB entry will be read/written by appropriate instructions.
Random	1	This pseudorandom value (actually a free-running counter) is used by a **tlbwr** to write a new TLB entry into a randomly selected location. Saves time when processing TLB refill traps for software that likes the idea of random replacement (there is probably no viable alternative).
Context	4	These are convenience registers, provided to speed up the processing of TLB refill traps. The high-order bits are read/write; the low-order bits are taken from the VPN of the address that couldn't be translated. The register fields are laid out so that, if you use the favored arrangement of memory-held copies of memory translation records, then following a TLB refill trap **Context** will contain a pointer to the page table record needed to map the offending address. See section 6.2.4. **XContext** does the same job for traps from processes using more than 32 bits of effective address space; a straightforward extension of the **Context** layout to larger spaces would be unworkable because of the size of the resulting data structures. Some 64-bit CPU software is happy with 32-bit virtual address spaces; but for when that's not enough, 64-bit CPUs are equipped with "mode bits" **SR(UX)**, **SR(KX)**, which can be set to invoke an alternative TLB refill handler; in turn that handler can use **XContext** to support a huge but manageable page table format.
XContext	20	

TLB Key Fields—EntryHi and PageMask

Figure 6.2 shows these registers, which, with **`EntryLo`** (described below), are the programmer's only view of a TLB entry. The diagram shows both the MIPS32 and MIPS64 versions of **`EntryHi`**, with these fields:

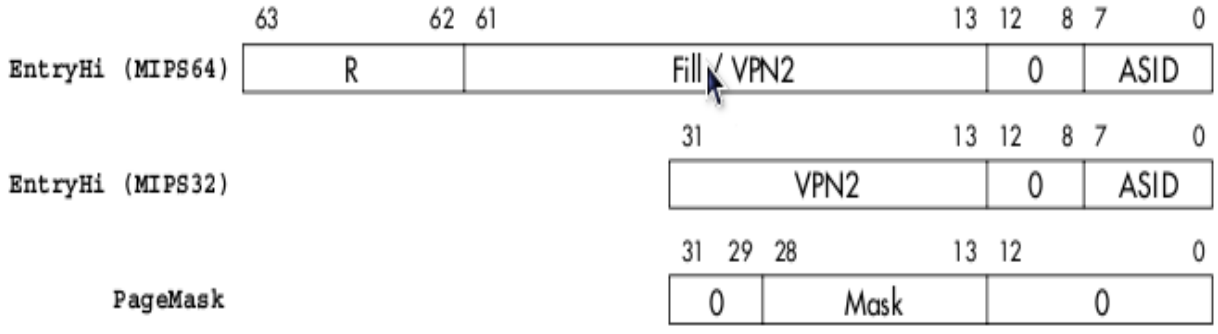
- *VPN2 (virtual page number)*: These are the high-order bits of a program address (with the low, in-page bits 0–13 omitted). Bits 0–12 would be the in-page address, but bit 13 of the program address isn't looked up either: Each entry is going to map a pair of 4-KB pages, and bit 13 will automatically select between the two possible output values.

  Following a refill exception, this field is set up automatically to match the program address that could not be translated. When you want to write a different TLB entry, or attempt a TLB probe, you have to set it up by hand.

  The MIPS64 version of **`EntryHi`** allows for bigger virtual address regions than any implemented so far—to as large as 2^{62} bits, though current CPUs typically only implement 2^{40} bits. The implemented bits of **`VPN2`** go up far enough to cope, and, in fact, you find out how much virtual space you've got through this register. If you write all-ones to **`EntryHi (Fill/VPN2)`** and then read it back, the valid bits of **`EntryHi (VPN2)`** will be those that read back as 1.

  In use, higher bits of **`VPN2`** than are used by your CPU *must* be written as all ones or all zeros, matching the most significant bit of the **`R`** field. Equivalently, the higher bits are all 1 when accessing kernel-only address spaces and all 0 otherwise.

  If you are only using the 32-bit instruction set, this will happen automatically, because when you work this way all register values contain the 64-bit sign extension of a 32-bit number. Therefore, 32-bit software running on 64-bit hardware can pretend it has a 32-bit **`EntryHi`** layout.



Note also there are a few unused zero bits below the VPN2 field; in fact they're not always unused—some CPUs can be configured to support a 1-KB page size, in which case VPN2 and the corresponding mask field grow downward by 2 bits.

- *ASID (address space identifier)*: This is normally left holding the operating system's idea of the current address space. This is not changed by exceptions, so after a refill exception, this will still have the right value in it for the currently running process.

  An OS using multiple address spaces will maintain this field to the current address space. But because it is tucked into **`EntryHi`**, you have to be careful when using **`tlbr`** to inspect TLB entries; that operation overwrites the whole of **`EntryHi`**, so you will have to restore the correct current ASID value afterward.

- *R*: (64-bit version only) This is an address region selector. But you can consistently regard this field as just more bits of **`EntryHi(VPN2)`**; it's just the highest-order bits of the 64-bit MIPS virtual address. However, if you remember the 64-bit extended-memory map (see Figure 2.2 in section 2.8), you can see that these high-order bits select memory regions with different access privileges:

`R` *value*	*Region name*	*Brief description*
0	xuseg	User-mode accessible space in low virtual memory
1	xsseg	Supervisor-mode accessible space (supervisor mode is optional)
2	xkphys	Kernel-only large windows on physical memory (cached and uncached)
3	xkseg	Kernel-mode space (includes MIPS32 compatibility segments)

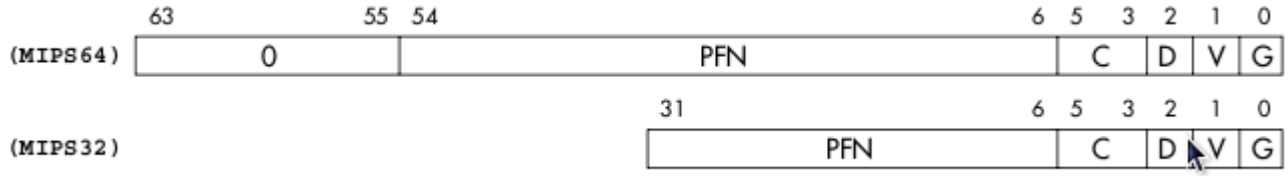
  The **`R`** bits are unlike the high bits of VPN2 because they can indeed take on different values—an implementation-defined number of high-order bits of **`EntryHi(VPN2)`** may only be all ones or all zeros.

The **`PageMask`** register allows you to set up TLB fields that map larger pages. The **`PageMask(Mask)`** field represents part of the TLB entry, and 1 bits have the effect of causing the corresponding bit of the virtual address to be ignored when matching the TLB entry (and causing that bit to be carried unchanged to the resulting physical address), effectively matching a larger page size.

TLB Output Fields—EntryLo0-1

Figure 6.3 shows both the 64- and 32-bit versions of **`EntryLo`**, whose fields are as follows:

- *PFN*: These are the high-order bits of the physical address to which values matching the entry field corresponding to **`EntryHi(VPN2)`** s will be translated. The active width of this field depends on the physical memory space supported by your CPU. MIPS32 CPUs are quite often attached to external interfaces limited to 2^{32} bytes range, but the MIPS32 version of **`EntryLo`** can potentially support as much as a $2^{3}8$ bytes physical range (the 26-bit **`PFN`** provides $2^{2}6$ pages, each 4 KB or $2^{1}2$ bytes in size).



- *C*: A 3-bit field originally defined for cache-coherent multiprocessor systems to set the "cache algorithm" (or "cache coherency attribute"—
- *D (dirty)*: This functions as a write-enable bit. Set 1 to allow writes, 0 to cause any store using this translation to be trapped. See section 14.4.7 for an explanation of the term *dirty*.
- *V (valid)*: If set 0, any use of an address matching this entry will cause an exception. Used either to mark a page that is not available for access (in a true virtual memory system) or to mark one **`EntryLo`** part of a paired translation as not available.
- *G (global)*: When the G bit in a TLB entry is set, that TLB entry will match solely on the VPN field, regardless of whether the TLB entry's ASID field matches the value in **`EntryHi`**. That provides an efficient mechanism to implement parts of the address space that are shared between all processes. Note that there is really only one "G" bit in a TLB entry, although there are two **`EntryLo`** registers: bad things will happen if you have different values in **`EntryLo0(G)`** and **`EntryLo1(G)`**.
- *Fields called 0 and unused PFN bits*: These fields always return zero, but unlike many reserved fields, they do not need to be written as zero (nothing happens regardless of the data written). This is important; it means that the memory-resident data used to generate **`EntryLo`** when refilling the TLB can contain some software-interpreted data in these fields, which the TLB hardware will ignore without the need to spend precious CPU cycles masking it out.

Selecting a TLB Entry—Index, Random, and Wired Registers

The **Index** register is used to pick a particular TLB entry—they run from zero up to the number of entries less one. You set **Index** when you deliberately want to read or write a particular entry, and **Index** is also set automatically when you do a software search for a TLB entry using **tlbp**.

Its low bits hold the TLB index.[2] Not many bits are needed, since no MIPS CPU has yet had a TLB with more than 128 entries. The high bit (bit 31) is magic, being set by **tlbp** when the probe fails to finds a matching entry. Bit 31 is a good choice—because it makes the value appear negative, it's easy to test for.

Random holds an index into the TLB that counts (downward, if that's important to you) with each instruction the CPU executes. It acts as an index into the TLB for the write-entry instruction **tlbwr**, facilitating a random replacement strategy when you need to write a new TLB entry.

You never have to read or write the **Random** register in normal use. The hardware sets the **Random** field to its maximum value—the highest-numbered entry in the TLB—on reset, and it decrements every clock period until it reaches a floor value, when it wraps back to its maximum and starts again.

TLB entries from 0 upward whose index is less than the floor value are therefore immune from random replacement, and an OS can use those slots for permanent translation entries—they are referred to as *wired* in MIPS OS documentation. The **Wired** register allows you to specify that floor and thus to determine the number of wired entries. When you write **Wired**, the **Random** register is reset to point to the top of the TLB.

Page-Table Access Helpers—Context and XContext

When the CPU takes an exception because a translation isn't in the TLB, the virtual address whose translation wasn't available is already in **BadVAddr**. The page-resolution address is also reflected in **EntryHi(VPN2)**, which is thereby preset to exactly the value needed to create a new entry to translate the missed address.

But to further speed the processing of this exception, the **Context** or **XContext** register repackages the page-resolution address in a format that can act as a ready-made pointer to a memory-based page table.

MIPS32 CPUs have just the **Context** register, which helps out the TLB refill process for 32-bit virtual address spaces; MIPS64 CPUs add the **XContext** register, to be used when using a larger address space (up to 40 bits).

The registers (both MIPS32 and MIPS64 versions) are shown in Figure 6.4.

Note that **XContext** is the only register in which the MIPS64 definition does not exactly define field boundaries: the **XContext(BadVPN2)** field grows on CPUs supporting virtual address regions bigger than 2^{40} bits and pushes the **R** and **PTEBase** fields left (the latter is squashed to fit).

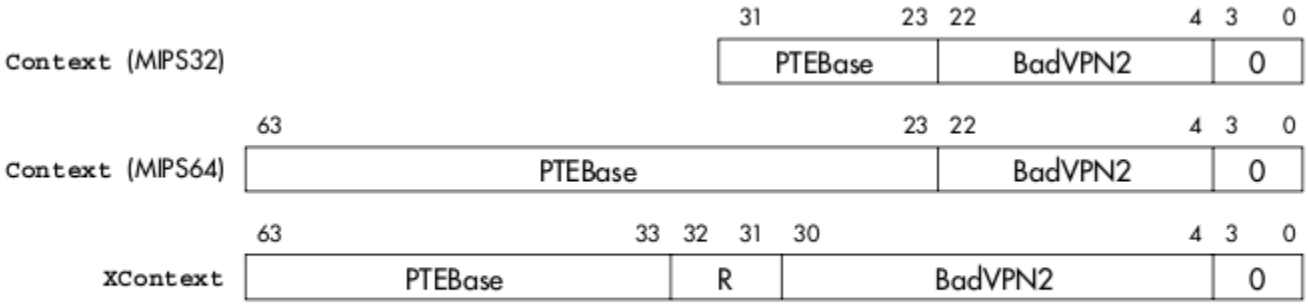


FIGURE 6.4 Fields in the MIPS32 and MIPS64 **Context** registers.

- **Context (PTEBase)**: This field just stores what you put in it. To exploit the register as its designers intended, write in the high-order bits of an (appropriately aligned) starting address of a memory-resident page table. The starting address must be picked to have zeros in bits 22 and downward—that's a 4-MB boundary. While it would be grossly inefficient to provide that alignment in physical or unmapped memory, the intention is that this table should be put in the kseg2 mapped region. See below for how this works.
- **Context (BadVPN2)/XContext (BadVPN2)**: Following a TLB-related exception, this holds the page address, which is just the high-order bits of **BadVAddr**.

  Why the "2" in the name? Recall that in the MIPS32/MIPS64 TLB, each entry maps an adjacent pair of virtual-address pages onto two independent physical pages.

  The **BadVPN2** value starts at bit 4, so as to precalculate a pointer into a table of 16-byte entries whose base address is in **PTEBase**. If the OS maintains this table so that the entry implicitly accessed by a particular virtual page address contains exactly the right **EntryLo0–1** data to create a TLB entry translating that page, then you minimize the work a TLB miss exception handler has to carry out; you can see that in section 6.5. If you're only translating 32-bit addresses and don't need too many bits of software-only state in the page table, you could get by with an eight-byte page-table entry. This turns out to be one of the reasons why Linux doesn't use the **Context** registers in the prescribed manner.

  The **XContext (BadVPN2)** field may be larger than is shown in Figure 6.4 if your CPU can handle more than 2^{40} bits of user virtual address space. When that happens, the **R** and **PTEBase** fields are pushed along to make space.

- **`XContext(PTEBase)`**: The page table base for 64-bit address regions must be aligned so that all the bits below those specified by **`XContext(PTEBase)`** are zero: That's 8 GB aligned. That sounds intolerable, but there is a suitable large, mapped, kernel-only-accessible region ("xkseg") in the basic MIPS64 memory map.
- **`XContext(R)`**: TLB misses can come from any mapped region of the CPU's memory map, not just from user space. All regions lie within one overarching 64-bit space, but are much smaller than is required to pack it full (you might like to refer to Figure 2.2 in section 2.8). Usable 64-bit virtual addresses are divided into four "xk..." segments within which you can use a 62-bit in-segment address.

  So as to save space in **`XContext`**, the miss address as shown here is kept as a separate 40-bit in-region page address (**`BadVPN2`**) and a 2-bit mapped-region selector **`XContext(R)`**, defined as follows:

`R` *value*	*Region name*	*Brief description*
0	xuseg	User-mode accessible space in low virtual memory
1	xsseg	Supervisor-mode accessible space (supervisor mode is optional)
2		Would correspond to unmapped segments, not used
3	xkseg	Kernel-mode mapped space (including old kseg2)

Programming the TLB

TLB entries are set up by writing the required fields into **`EntryHi`** and **`EntryLo`**, then using a **`tlbwr`** or **`tlbwi`** instruction to copy that entry into the TLB proper.

When you are handling a TLB refill exception, you will find that **`EntryHi`** has been set up for you already.

Be very careful not to create two entries that will match the same program address/ASID pair. If the TLB contains duplicate entries, an attempt to translate such an address, or probe for it, has the potential to damage the CPU chip. Some CPUs protect themselves in these circumstances by a TLB shutdown, which shows up as the **`SR(TS)`** bit being set. The TLB will now match nothing until a hardware reset.

System software often won't need to read TLB entries at all. But if you need to read them, you can find the TLB entry matching some particular program address using **`tlbp`** to set up the **`Index`** register. Don't forget to save **`EntryHi`** and restore it afterward—the **`EntryHi(ASID)`** field is likely to be important.

Use a **`tlbr`** to read the TLB entry into **`EntryHi`** and **`EntryLo0-1`**.

You'll see references in the CPU documentation to separate "ITLB" and "DTLB" (or sometimes collectively "uTLB"—the "u" is for "micro") structures. The micro-TLBs perform translation for instruction and data addresses, respectively; these are tiny hardware-managed caches of translations, whose operation is completely transparent to software—they are automatically invalidated whenever you write an entry to the main TLB.

How Refill Happens

When a program makes an access in any of the translated address regions (normally kuseg for application programs under a protected OS and kseg2 for kernel-privilege mappings), and no translation record is present, the CPU takes a TLB refill exception.

The TLB can only map a fraction of the physical memory range of a modern server or workstation. Large OSs maintain some kind of memory-held page table that holds a large number of page translations and uses the TLB as a cache of recently used translations. For efficiency, it's common to arrange that the page table will be an array of ready-to-use TLB entries; for even more efficiency, you can set locate and structure the table so that you can use the **`Context`** or **`XContext`** register as a pointer into it.

Since MIPS systems usually run OS code in the untranslated kseg0 memory region, the common situation will be a miss by a user-privilege program. Several hardware features are provided, with the aim of speeding up the exception handler in this common case. First, these refill exceptions are vectored through a low-memory address used for no other exception.[3] Second, a series of cunning tricks allows the memory-held page table to be located in kernel virtual memory (the kseg2 region or its 64-bit alternative) so that physical memory space is not needed for the parts of the page table that map "holes" in the process's address map.

And to top it off, the **`Context`** or **`XContext`** register can be used to give immediate access to the right entry from a memory-held page table.

We'll work through this process in section 6.5. But before we get too far into it, we should note that use of all these features is *not compulsory*. In a smaller system the TLB can be used to produce a fixed or rarely changing translation from program (virtual) to physical addresses; in these cases it won't even need to be a cache.

Hardware-Friendly Page Tables and Refill Mechanism

There's a particular translation mechanism that the MIPS architects undoubtedly had in mind for user addresses in a UNIX-like OS. It relies upon building a page table in memory for each address space. The page table consists of a linear array of entries, indexed by the VPN, whose format is matched to the bitfields of the **`EntryLo`** register. The paired TLBs need 2 × 64-bit entries, 16 bytes per entry.

That minimizes the load on the critical refill exception handler but opens up other problems. Since each 8 KB of user address space takes 16 bytes of table space, the entire 2 GB of user space needs a 4-MB table, which is an embarrassingly large chunk of data.[4] Of course, most user address spaces are only filled at the bottom (with code and data) and at the top (with a downward growing stack) with a huge gap in between. The solution MIPS adopted is inspired by DEC's VAX architecture, and is to locate the page table itself in virtual memory in a kernel-mapped (kseg2 or xkseg) region. This neatly solves two problems at once:

- It saves physical memory: Since the unused gap in the middle of the page table will never be referenced, no physical memory need actually be allocated for those entries.
- It provides an easy mechanism for remapping a new user page table when changing context, without having to find enough virtual addresses in the OS to map all the page tables at once. Instead, you have a different kernel memory map for each different address space, and when you change the **`ASID`** value, the kseg2 pointer to the page table is now automatically remapped onto the correct page table. It's nearly magic.

The MIPS architecture supports this kind of linear page table in the form of the **`Context`** register (or **`XContext`** for extended addressing in 64-bit CPUs).

If you make your page table start at a 4-MB boundary (since it is in virtual memory, any gap created won't use up physical memory space) and set up the **`Context`** PTEBase field with the high-order bits of the page table starting the address, then, following a user refill exception, the **`Context`** register will contain the address of the entry you need for the refill with no further calculation needed.

So far so good: But this scheme seems to lead to a fatal vicious circle, where a TLB refill exception handler may itself get a TLB refill exception, because the kseg2 mapping for the page table isn't in the TLB. But we can fix that, too.

If a nested TLB refill exception happens, it happens with the CPU already in exception mode. In MIPS CPUs, a TLB refill from exception mode is directed to the general exception entry point, where it will be detected and can be handled specially.

TLB Miss Handling

A TLB miss exception always uses a dedicated entry point unless the CPU is already handling an exception—that is, unless **SR(EXL)** is set.

Here is the code for a TLB miss handler for a MIPS32 CPU (or a MIPS64 CPU handling translations for a 32-bit address space):

```
    .set    noreorder
    .set    noat
TLBmiss32:
    mfc0    k1,C0_CONTEXT   # (1)
    lw      k0,0(k1)        # (2)
    lw      k1,8(k1)        # (3)
    mtc0    k0,C0_ENTRYLO0  # (4)
    mtc0    k1,C0_ENTRYLO1  # (5)
    ehb                     # (6)
    tlbwr                   # (7)
    eret                    # (8)
    .set    at
    .set    reorder
```

Following is a line-by-line analysis of the code:

(1) The **k0–1** general-purpose registers are conventionally reserved for the use of low-level exception handlers; we can just go ahead and use them.

(2–5) There are a pair of physical-side (**EntryLo**) descriptions in each TLB entry (you might like to glance back at the TLB entry diagram, Figure 6.1). The layout of the MIPS32/64 **Context** register shown in Figure 6.4 reserves 16 bytes for each paired entry (eight bytes of space for each physical page), even though MIPS32's **EntryLo0–1** are 32-bit registers. This is for compatibility with the 64-bit page table and to provide some spare fields in the page table to keep software-only information.

Interleaving the **lw**/**mtc0** sequences here will save time: Few MIPS CPUs can keep going without pause if you use loaded data in the very next instruction.

These loads are vulnerable to a nested TLB miss if the kseg2 address's translation is not in the page table. We'll talk about that later.

(6) It's no good writing the entry with **tlbwr** until it will get the right data from **EntryLo1**. The MIPS32 architecture does not guarantee this will be ready for the immediately following instruction, but it does guarantee that the sequence will be safe if the instructions are separated by an **ehb** (execution hazard barrier) instruction—see section 3.4 for more information about hazard barriers.

(7) This is random replacement of a translation pair as discussed.

(8) MIPS32 (and all MIPS CPUs later than MIPS I) have the **eret** instruction, which returns from the exception to the address in **EPC** and unsets **SR(EXL)**.

So what happens when you get another TLB miss? The miss from exception level invokes not the special high-speed handler but the general-purpose exception entry point. We're already in exception mode, so we don't alter the exception return register **EPC**.

The **Cause** register and the address-exception registers (**BadVAddr**, **EntryHi**, and even **Context** and **XContext**) will relate to the TLB miss on the page table address in kseg2. But **EPC** still points back at the instruction that caused the original TLB miss.

The exception handler will fix up the kseg2 page table miss (so long as this was a legal address) and the general exception handler will return into the user program. Of course, we haven't done anything about the translation for the user address that originally caused the user-space TLB miss, so it will immediately miss again. But the second time around, the page table translation will be available and the user miss handler will complete successfully. Neat.

XTLB Miss Handler

MIPS64 CPUs have two special entry points. One—shared with MIPS32 CPUs—is used to handle translations for processes using only 32 bits of address space; an additional entry point is provided for programs marked as using the bigger address spaces available with 64-bit pointers.

The status register has three fields, **SR(UX)**, **SR(SX)**, and **SR(KX)**, that select which exception handler to use based on the CPU privilege level at the time of the failed translation.[5]

With the appropriate status bit set (**SR(UX)** for user mode), a TLB miss exception uses a different vector, where we should have a routine that will reload translations for a huge address space. The handler code (of an XTLB miss handler for a CPU with 64-bit address space) looks much like the 32-bit version,

except for the 64-bit-wide registers and the use of the **XContext** register in place of **Context**:

```
    .set    noreorder
    .set    noat
TLBmissR4K:
    dmfc0   k1,C0_XCONTEXT
    ld      k0,0(k1)
    ld      k1,8(k1)
    dmtc0   k0,C0_ENTRYLO0
    dmtc0   k1,C0_ENTRYLO1
    ehb
    tlbwr
    eret
    .set    at
    .set    reorder
```

Note, though, that the resulting page table structure in kernel virtual memory is far bigger and will undoubtedly be in the giant xkseg region.

I should remind you again that this system is not compulsory, and in fact is not used by the MIPS version of Linux (which is overwhelmingly the most popular translated-address OS for MIPS applications). It's a rather deeply ingrained design choice in the Linux kernel that the kernel's own code and data are not remapped by a context switch, but exactly that is required for the kseg2/xkseg page table trick described here. See section 14.4.8 for how it's done.

Differences Of MIPS and x86

(1) (Handle TLB Miss)

Hardware-Managed TLB:- (Intel IA-32(x86)):

Typical of early Memory-Management Units.
A hardware state machine is used to refill the TLB.
In the event of a TLB miss, the state machine would walk the page table, locate the mapping, insert it into the TLB, and restart the computation.

Advantage: Performance
Disadvantage: Inflexibility of Page table organization design
The page table organization is effectively fixed in the hardware design.
The operating system has no flexibility in choosing a design.

Software-Managed TLB:- (MIPS)

Typical of recent Memory-Management Units.
No hardware TLB-refill state machine to handle TLB misses.
On a TLB miss, the hardware interrupts the operating system and vectors to a software routine that walks the page table and refills the TLB.

Advantage: Flexibility of Page table organization design
The page table can be defined entirely by the operating system, since hardware never directly manages the table.
Disadvantage: Performance cost.
The TLB miss handler that walks the page table is an operating system primitive which usually requires 10 to 100 instructions. If the handler code is not in the instruction cache at the time of the TLB miss exception, the time to handle the miss can be much longer than in the hardware walked scheme.
In addition, the use of precise exception handling mechanisms adds to the cost by flushing the pipeline, removing a possibly large number of instructions from the reorder buffer. This can add hundreds of cycles to the overhead of walking the page table by software.

(2) (Process's Address Space)

Per-process virtual address space:
The effective or logical virtual address generated by a process is extended by an address-space identifier (ASID) in included in TLB and page table entries (PTE s) to distinguish between processes or contexts
Each process may have a separate page table to handle address translation.
e.g. MIPS, Alpha, PA-RISC, UltraSPARC.
Global system-wide virtual address space:
The effective or logical virtual address generated by a process is extended by a segment number forming a global, flat or extended global virtual address. (paged segmentation)
Usually a number of segment registers specify the segments assigned to a process.
A global page translation table may be used for all processes running on the system.
e.g. IA-32 (x86), Power-PC.

Comparisons of six architectures features in Memory Management Unit

	MIPS	Alpha	PowerPC	PA-RISC	SPARC	x86
Address Space Protection	ASIDs	ASIDs	Segmentation	Multiple ASIDs & Segmentation	ASIDs	Segmentation
Shared Memory	GLOBAL bit in TLB entry	GLOBAL bit in TLB entry	Segmentation	Multiple ASIDs & Segmentation	Indirect specification of ASIDs	Segmentation & Sharing PTE pages
Large Address Spaces	64-bit addressing	64-bit addressing	52-bit segmented addressing	96-bit segmented addressing	64-bit addressing	None
Fine-Grained Protection	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry, also per-segment
Sparse Address Spaces	Software-managed TLB	Software-managed TLB	Inverted software TLB cache	Software-managed TLB	Software-managed TLB	None
Superpages	Variable page size set in TLB entry: 4KB-16MB, by 4	Groupings of 8, 64, or 512 pages (set in TLB entry)	Block Address Translation: 128KB-256MB, by 2	Variable page size set in TLB entry: 4KB-64MB, by 4	Variable page size set in TLB entry: 8KB, 64KB, 512KB, 4MB	Variable page size set in TLB entry: 4KB or 4MB

Translation Look Aside Buffer

Hardware Managed TLB.

A hardware-managed storage location stores virtual to physical memory translations that are inserted into that storage location directly by a microprocessor. In other words, the virtual to physical memory translation is inserted into the storage location by the microprocessor without a "special computer program instruction" that instructs the microprocessor to insert the translation into the particular storage location. An example of a hardware-managed TLB is found in Intel's Pentium(**x86**).
A hardware-managed TLB may automatically fill the TLB with a requested physical page number when a TLB miss occurs. This hardware assist may occur concurrently with other microprocessor functions. Further, a hardware-managed TLB may ensure that the most recently used physical pages are stored in the TLB.
The primary disadvantage of hardware-managed TLBs is that the storage of the most recently used page numbers may not be optimal for certain computer programs. Thus, certain computer programs may require significantly more execution time than would be required if the TLB was more optimally managed.

Software Managed TLB.

A software-managed TLB may be controlled by a computer program running on a microprocessor such as a computer operating system. A computer program may store certain virtual to physical memory translations in the TLB regardless of physical page use patterns. For example, virtual to physical memory translations for operating system kernels, frame buffers, or input-output areas may be stored in the TLB regardless of their use patterns. A virtual to physical memory translation that remains stored in the TLB regardless of the physical page use pattern is known as "locked down" or "pinned" in the TLB. An example of a software-managed TLB is found in the **MIPS** processor.
A disadvantage of software-managed TLBs is that they may not be automatically filled by microprocessor hardware concurrent with other microprocessor activity. Another disadvantage of software-managed TLBs is that they are less flexible than hardware-managed TLBs. For example, computer programs may not take advantage of additional TLB resources that are included in certain high-performance microprocessors. Similarly, microprocessor manufacturers may be required to build software-managed TLBs that are backwardly compatible with previous non-optimal software-managed TLBs.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

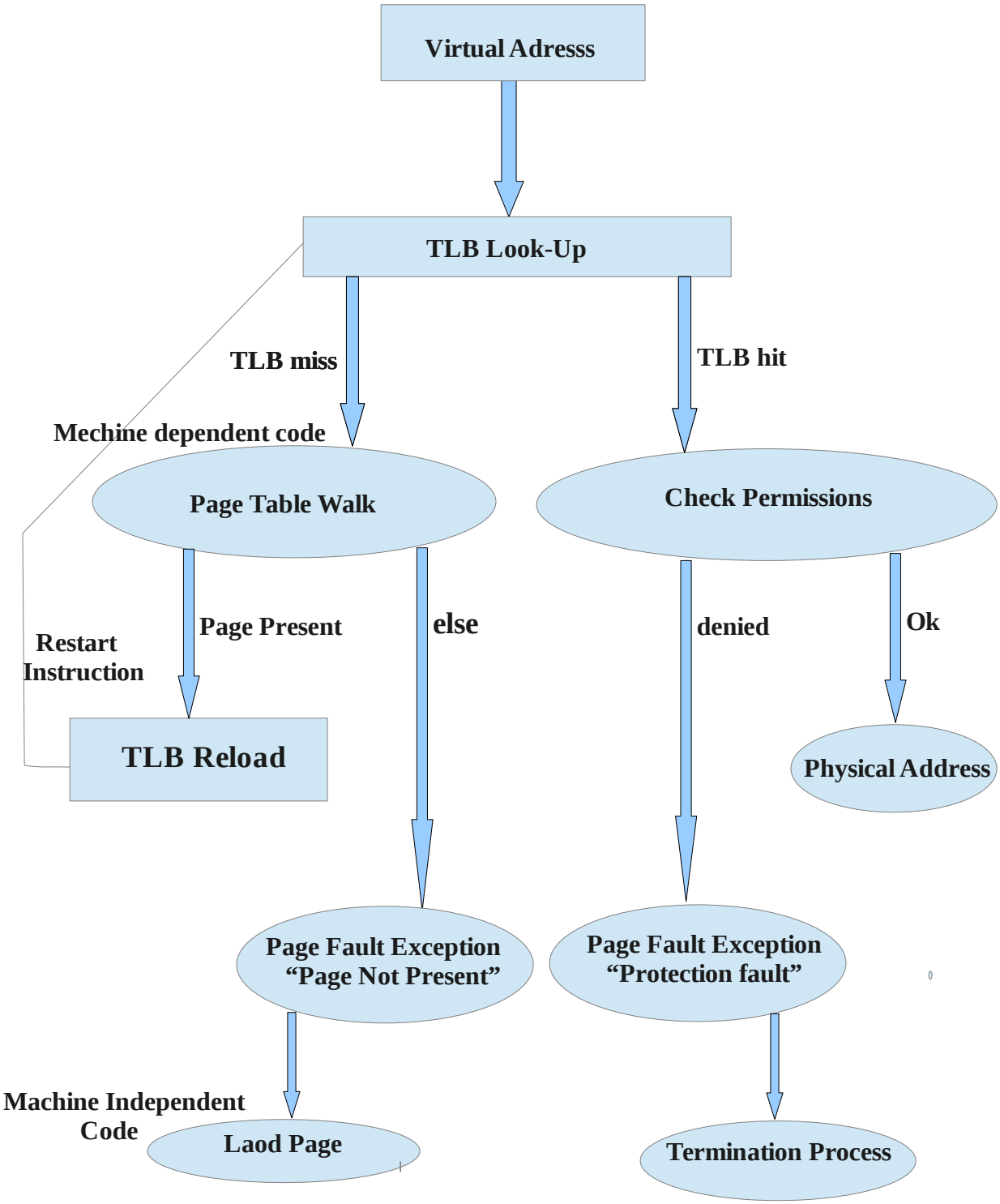
Flushing TLB:-

x86 : Flush entire TLB on Process Switch.
MIPS :Flushing entries(some) only necessary when process id reused.

Reloading TLB:-

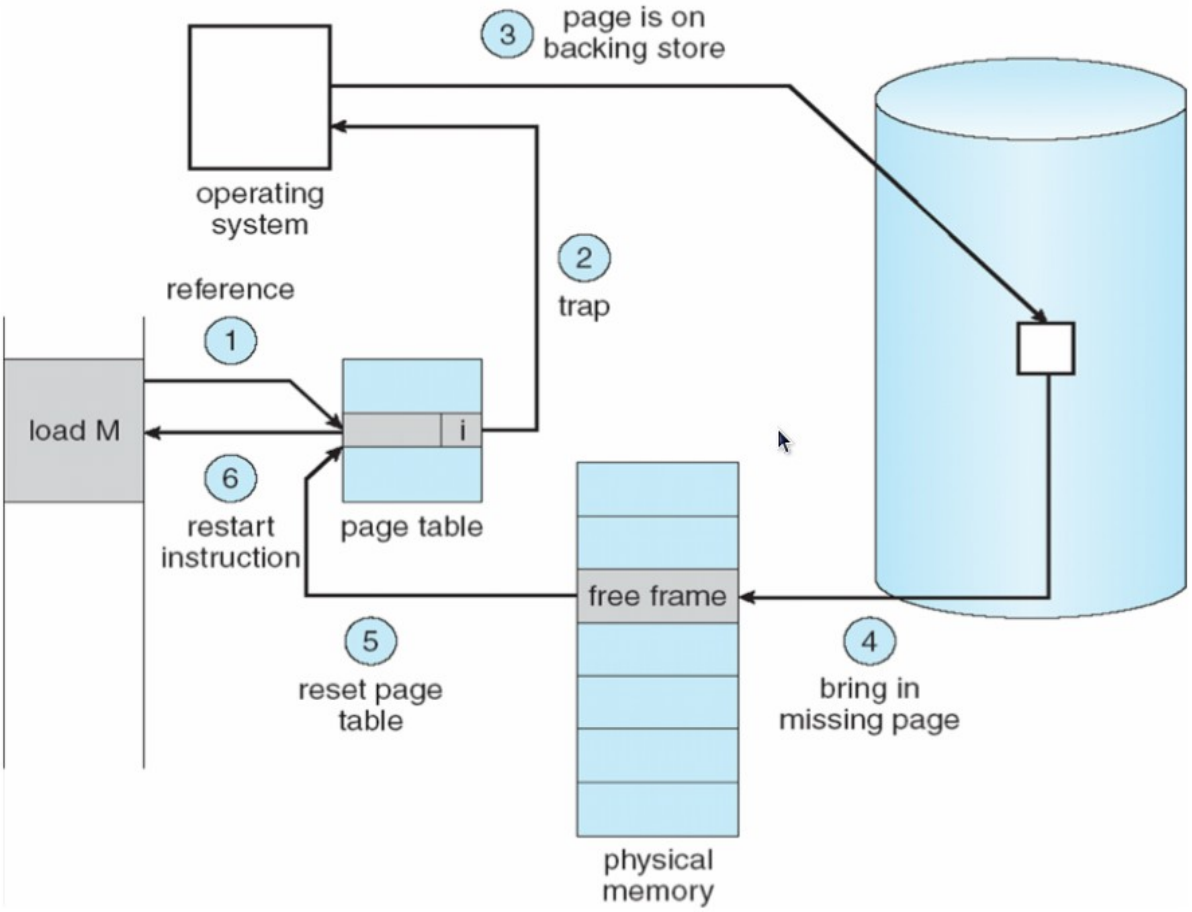
x86:-Hardware and Software must agree on page table layout
MIPS:-OS designer can pick any page layout-page table is only read and written by OS.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Address Translation and TLB



1. **Page Table Walk** and **TLB Reload** can be done in Software or Hardware.
2. **Load Page** and **Process Termination** will be done by OS.
3. **TLB Look-Up** and **Check Permissions** will be done by the Hardware.

General Page Fault Handling



1. CPU Generates Logical Address for load M. A reference to the page table indicates that page is not into the memory and gives its disk address.

2. A trap generated. System goes into the privileged mode (kernel mode).

3. The page is located onto the disk.

4. Room is made available into the memory and the page is moved.

5. Page table is updated to indicate that the page is now available into the memory.

6. The instruction is restarted (now the system is in user mode).

Page fault Handler in Linux kernel

Function: do_page_fault() (**arch/i386/mm/fault.c**) /****** for x86 architecture ****/

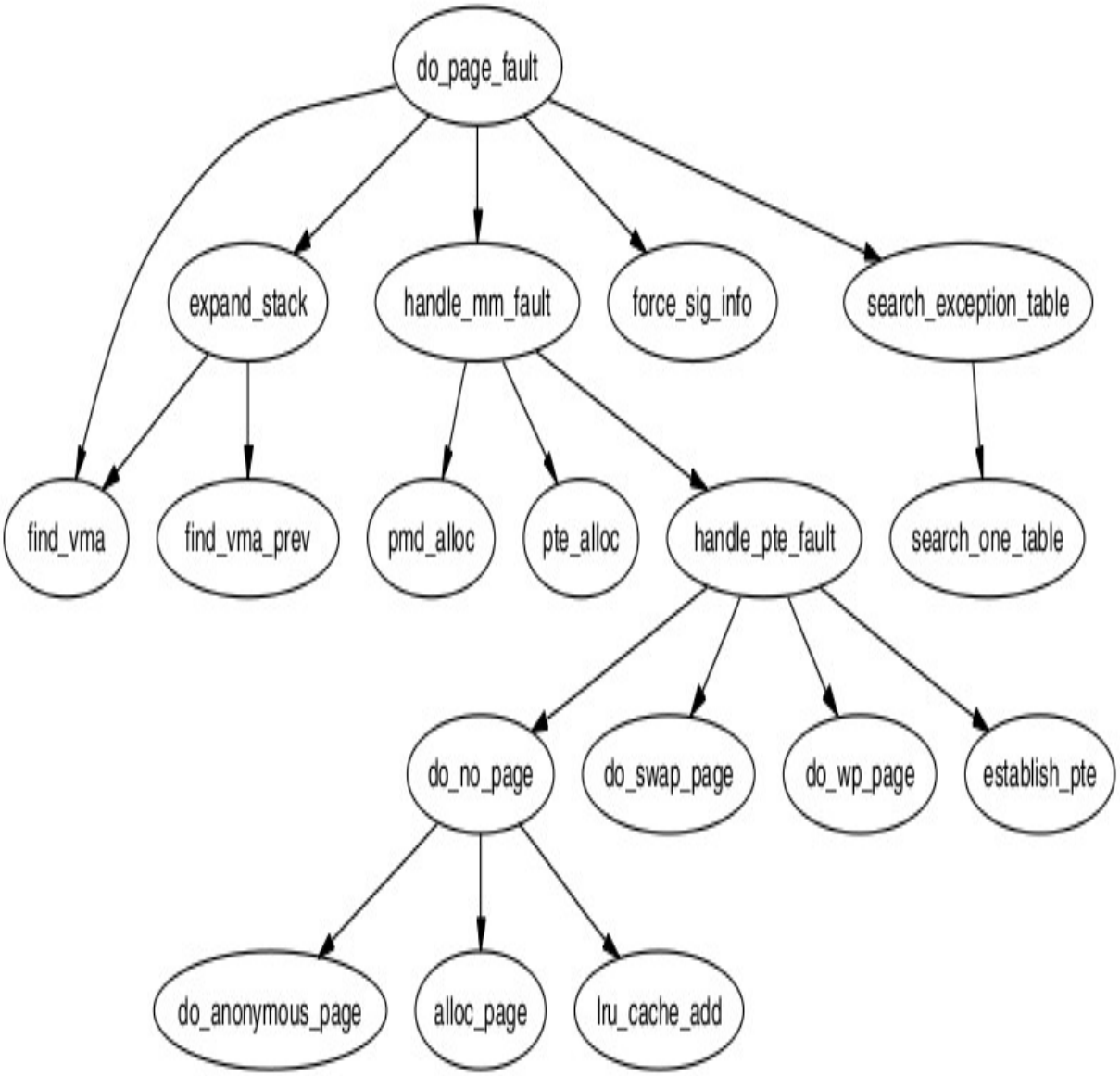


Figure 4.11. Call Graph: do_page_fault()

1. Xen Memory Management

In this section we address the changes Xen makes to memory management to make virtualization possible. To do this we sum up memory management in the Linux kernel and address the changes. In particular we will cover page tables, shadow page tables, page faults, ballooning and grant tables.

1.1 Memory in Xen

Xen allocates a small portion of the physical memory for its own use and it also reserves a fixed portion in the upper virtual address space of each guest VM on the system e.g. 64MB on the IA-32 architecture. This is to prevent that the TLB is flushed every time a hyper switch is performed. All memory allocations are performed at a page level granularity and the VMM tracks the ownership and use of each page to enforce isolation.

In Linux the memory is normally allocated in contiguous blocks of machine memory, but because the VMM allocates at page level it can not be guaranteed that it will be a contiguous block. This can be a problem because most operating systems does not have good support for fragmented memory. To overcome this problem Xen introduces a pseudo-physical memory, often referred to as physical memory. We will adapt this terminology. Physical memory is a per guest VM abstraction of machine addresses. When using physical memory the address space will look like it is one contiguous range of memory from the guest VMs point of view. However it may actually be allocated in any given order in machine memory. To make this possible the VMM contains a globally readable machine-to-physical table which contains the mappings from machine page frames to physical ones. Furthermore each guest VM has a physical-to-machine table with the reverse mapping.

Throughout the rest of this report we will use the terminology in Table 2.1. The page table (PT) mapping, which is placed in the guest OS, performs a translation from a virtual-to-machine address. The usage of PTs in both Linux and Xen will be explained in the next sections. The next two mappings, machine-to-physical (M2P) and physical-to-machine (P2M), are the ones mentioned before. The last translation, Shadow Page Tables (SPT), is also a mapping from virtual to machine addresses. The latter is however is an optimization .

1.2 Linux Page Tables

In the following section the Linux PTs will be explained and the usage of them.

The PT structure in Linux consists of an architecture independent three level structure, even on systems where the architecture does not support it e.g. the IA-32 architecture. On most architectures the Memory Management Unit (MMU) handles the Pts.

Each process has a mm struct that points into the Page Global Directory (PGD) of the process, which is a physical page in memory. On the IA-32 architecture the PGD is a single 4KB page and each entry is a 32 bit word, which means that it can contain 1024 entries. Each active entry in the PGD points to an entry in the Page Middle Directory (PMD), which again points to a Page Table Entry (PTE) entry. Finally the PTE points to the page where the actual user data is saved. If a page should be swapped out to backing storage the PTE will contain a swap entry for that page. It should be noted that when the PT of a process is loaded on the IA-32 architecture the Translation Lookaside Buffer (TLB) is flushed. The TLB is a cache where address translations from virtual to physical is saved to speed up memory lookups.

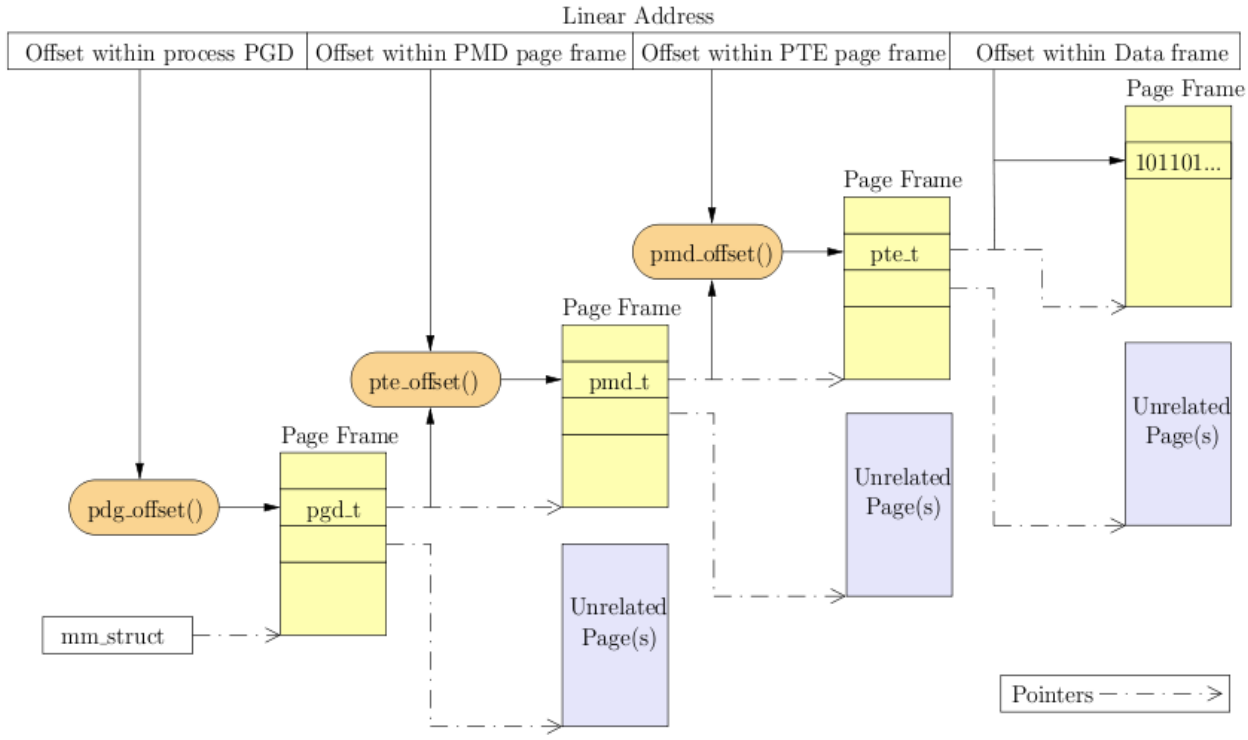


Figure 2.2: Lookup in the Linux page tables

Figure 2.2 illustrates how a lookup for a given page is performed. The linear address is split into four parts, where each part is used as an offset. The first three parts are used on the PTs and the last one on the actual page. First we find the right index into the PGD table by using the first part of the address as an offset. We follow the pointer located at that offset into the PMD page frame. Then we use the second offset from the address to find the right place in the PMD table and follows that pointer into the PTE. We use the third part of the address as an offset into the PTE and finds the pointer to the actual page containing the data. The last part of the address is used as an offset into the actual page.

As mentioned this three leveled PT structure is not supported on the IA-32 architecture. Actually the IA-32 only supports a two level structure, which is handled by directly “folding back” the PMD onto the PGD and it is optimized out at compile time [19, p. 33]. It should be noted that this is only true when not using the Page Address Extension (PAE), which adds four bits more to give 36 bit addressing. This will break the 4GB memory limit and provide a total of 64GB memory.

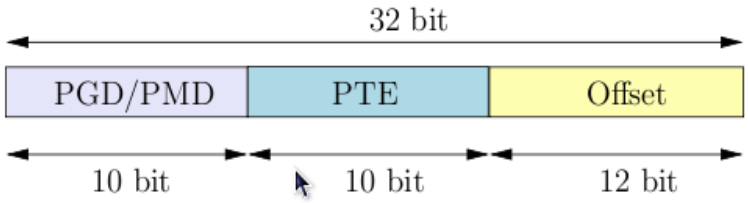


Figure 2.3: Virtual address on the IA-32 architecture

On the IA-32, without PAE, a virtual address is 32 bit, a single word. As it can be seen in Figure 2.3 on the preceding page the virtual address is split into three parts. The first 10 bits of the address is used as an index into the PGD, which means that the table has 1024 entries (210) and since each entry is 4 bytes wide it requires exactly one page (4KB) in memory. The next 10 bits are used as index into the PTE in the same manner. These lookups in the page tables gives the location of the actual page in memory and the last 12 bits are used as an offset into that page.

1.3 The Page Table Entries

Each entry in the page table structure consists of the structs pgd t, pmd t and pte t, which again are architecture dependent.

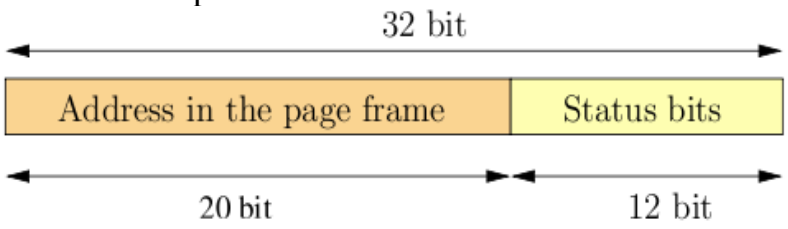


Figure 2.4: A single 32 bit page table entry on the IA-32 architecture

1.4 Xen Page Tables

The most significant changes in the memory management system in Xen is the management of PTs and the possibility to create shadow page tables.

When using Page tables the guest OS has direct read access to the Page tables, while updates of the page tables must be validated by the VMM [3]. A single page can have five different types and they are mutually-exclusive. The two types PD (page directory) and PT (page table) are used to indicate that a page is part of a page table structure. The local descriptor table (LDT) and global descriptor table (GDT) is used by the guest OS if it does not support paging but segmentation. The last type is used to indicated that a page is writable (RW). That the types are mutually-exclusive is very useful when validating a write to a page frame. This protects the page tables in the guest OS from accidentally or purposefully violating other VMs address space.

Furthermore each page in the memory has a reference counter, which keeps track of the number of references to it. A page can not be reallocated as long as it has a type and the reference counter has not reached zero. When a guest VM creates a new process it is expected to allocate and initialize its own PT from inside its address space and register it with the VMM. When it has registered with the VMM, there are two possible ways to make updates to the PTs.

Instant validation: When a process in the guest OS wants to update one of its PTs it makes a hypercall (mmu update), which transfers control to the hypervisor. The hypervisor then checks that the update does not violate the isolation of the guest VM. If it violates the memory constrains, the guest OS is denied write access to the page table. If it does not violate any constrains it is allowed to complete the write operation and update the PT.

Just in time validation: This method gives the guest OS the illusion that their page tables are directly writable. The VMM traps all writes to memory pages of the type PT. If a write occurs, then the VMM will allow writes to that page, but at the same time disconnects it from the currently active page table. In this way the guest OS can safely make updates to the page, because the updated entries cannot be used by the MMU until the VMM validates the page and re-connects it to the page table [45]. The VMM re-connects the page when: the TLB is flushed, a page in the unconnected page-tables page is accessed or the guest OS modifies another PTE entry in a separate PTE.
The PTs are only handled this way when the guest VM request it through a hypercall vm assist. It should be noted that writable PTs do not yield full virtualization of the MMU. The memory management code in the guest OS still needs to be aware of Xen.

1.5 Shadow Page Tables

As mentioned the normal PTs are address translations from virtual-to-machine, where each VM has direct read access to its own PT. The only restriction is updates, which has to be validated by the VMM.

When in SPT mode, the PT in the VM is not used directly by the hardware. In fact the guest OS does not have access to the SPT, which is used by the hardware [46] and the OS PT is not performing a mapping from virtual-to-machine, but a virtual-to-physical mapping.

The SPT gives the VMM the opportunity to track all writes performed to the pages in the PT. This have been used in e.g. live migration [8] to detect dirty pages while performing live migration. SPTs have a performance penalty in the form of keeping two PT structures up-to-date and the extra memory usage needed to save the SPT. The SPT is dynamically generated from the two mappings (PT, P2M), which can be seen in Figure 2.5 on the next page and can be discarded at any time.

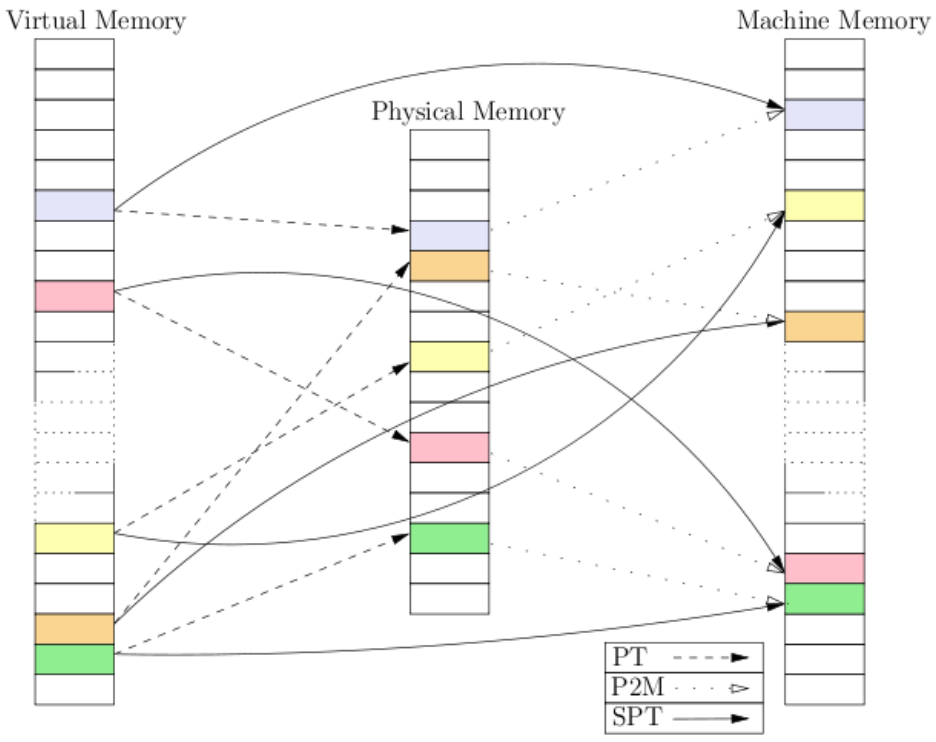


Figure 2.5: Mappings of Page Tables, Physical-to-Machine and Shadow Page Tables.

In fact each time a context-, world- or hyper switch occurs the SPT is discharged. This can however be optimized by having a SPT cache, which will take up extra memory to make SPT persistent between switches.

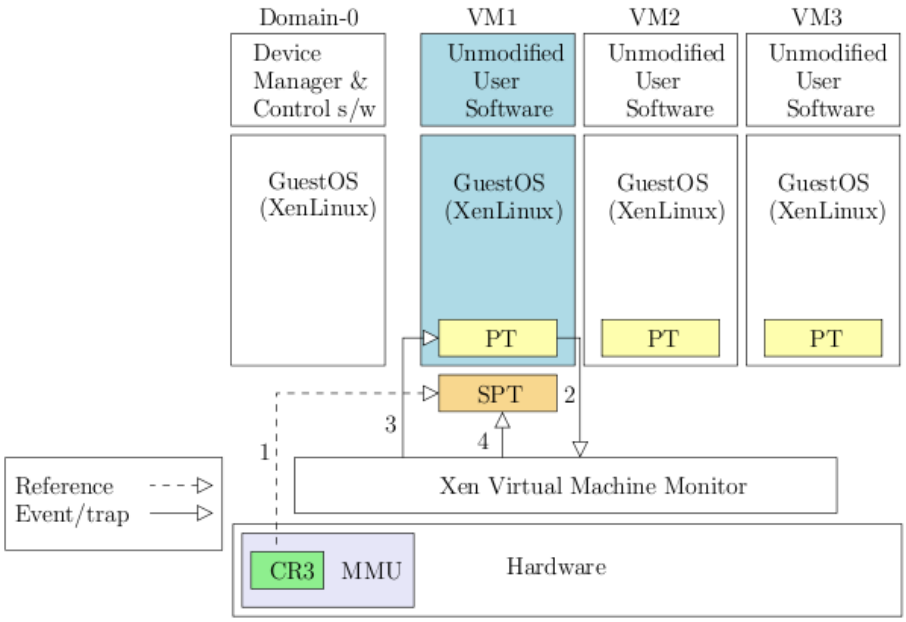


Figure 2.6: Update of the Guest OS Page Tables with Shadow Page Tables on the IA-32 Architecture.

Figure 2.6 illustrate three VMs, where VM1 is currently active and uses a SPT. The following enumeration explains how a change to the guest VMs PT is handle in shadow mode.

1. When Xen is not in SPT mode the active VMs page table is accessed via the CR3 register in the processor. This means that all translations are handle by the hardware MMU. But as it can be seen in the figure, the CR3 register points to the SPT, which the VM does not have direct access to.

2. When the guest OS tries to perform a write to a PT it gets trapped by the VMM, which as always validates the write to ensure isolation.

3.The write is granted or denied and the guest OS is notified with an event.

4. If the write was granted then the write is also propagated to the SPT.

We have now explained the functionality behind page tables and shadow page tables in Xen. We will now explain what happens when the memory management unit raises an exception, also known as a page fault.

1.6 Page Faults

This section is about page faults in the Linux kernel and how Xen handles these.

The pages of a process are not necessarily in memory in Linux. Often they are swapped out to disk and if accessed a page fault is generated. A page fault can also occur when some page is marked as read-only and a write operation is performed on that page. Linux uses a Demand Fetch policy for dealing with pages that is not in memory, which means that a page is not fetched into memory before the hardware raises a page fault. What the OS does is that it traps the page fault and allocates a page frame to bring the needed page back into memory. Page faults are divided into two groups depending on how expensive they are: major and minor faults. A major fault occurs when an expensive operation is required to handle the fault e.g. a disk read is needed to fetch a page from swap. A minor, or soft, page fault is when it is fairly simple to correct the fault e.g. a write to a page, which has been marked as read-only.

When a page fault occurs on the IA-32 architecture the fault is trapped by the kernel and the exception handler is called. This reads the faulting address from the processor register CR2. The first thing the exception handler checks which context the page fault came from, kernel or user space. If the page fault happened in kernel space and the faulting address is in the kernels address space, the fault is handled. If the kernel tried to access memory outside its address space or if the page fault occurred in interrupt context the kernel generates an oops.

A segmentation fault (SIGV) will be generated and the process is terminated if the page fault occurred in user space and one of the following scenarios are true: The user space process has tried to access 1) a NULL pointer, 2) kernel space, 3) invalid virtual address or 4) tried to write to a read-only section in virtual memory. If none of these are true, then the exception table is accessed to find the address of the “fixup” code and the EIP register (program pointer) is updated to point to that code, which is then executed. When this has been done the control is handed back to the process that invoked the page fault.

In Xen page faults are handled as in unmodified Linux with the exception that a guest VM can not read the CR2 register, because it requires privileged instructions. So the VMM has to copy the faulting address from the CR2 register into the stack frame of the guest OS, which has been extend with one word to contain the address. The faulted guest OS can then read the address and handle the fault, which may lead to updates of the page table. It should be noted that the handling of page faults is architecture dependent and the register mentioned in this section is only true for the IA-32 architecture.

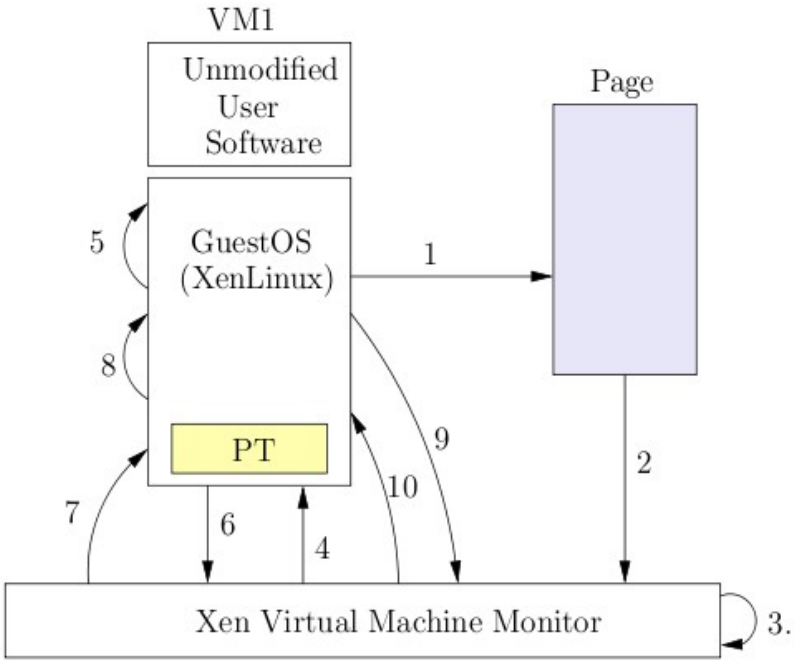


Figure 2.7: Handling of Page Faults in Xen

The following enumeration explains how Xen handles a page fault and which parts of the virtualized environment handles the different tasks. The page fault process is illustrated in Figure 2.7.

1. A process in the guest OS tries to read or write to a page, which triggers a page fault.

2. The VMM traps the page fault, which requires a hyper switch to the VMM.

3. The VMM copies the faulting address from the register in the processor into the extended stack frame.

4. The VMM notifies the guest OS by sending an event.

5. The guest OS reads the faulting address from the extended stack frame and finds the address of the “fixup” code in the exception stack.

6. We have to make a hyper switch back to the VMM to update the program counter such that the “fixup” code is executed. This is because the program counter can only be updated by executing privileged instructions.

7. A hyper switch back to the guest OS is required. This is because it is in the guest OS address space that the “fixup” code is located.
8. The “fixup” code is executed.
9. The execution of the “fixup” code requires that the PT in the guest OS is updated. This update has to be validated by the VMM and yet another hyper switch is required.
10. The process that provoked the page fault can continue its execution.

With the explanation of page faults, we have covered the usage of page tables. Dynamically changing the memory footprint of a VM is difficult. In the next section we explain a technique called ballooning, which is used to change the memory footprint of a VM.

2. Memory Virtualization

Beyond CPU virtualization, the next critical component is memory virtualization. This involves sharing the physical system memory and dynamically allocating it to virtual machines. Virtual machine memory virtualization is very similar to the virtual memory support provided by modern operating systems. Applications see a contiguous address space that is not necessarily tied to the underlying physical memory in the system. The operating system keeps mappings of virtual page numbers to physical page numbers stored in page tables. All modern x86 CPUs include a memory management unit (MMU) and a translation lookaside buffer (TLB) to optimize virtual memory performance.

To run multiple virtual machines on a single system, another level of memory virtualization is required. In other words, one has to virtualize the MMU to support the guest OS. The guest OS continues to control the mapping of virtual addresses to the guest memory physical addresses, but the guest OS cannot have direct access to the actual machine memory. The VMM is responsible for mapping guest physical memory to the actual machine memory, and it uses shadow page tables to accelerate the mappings. As depicted by the red line in Figure 8, the VMM uses TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to physical memory mapping, the VMM updates the shadow page tables to enable a direct lookup. MMU virtualization creates some overhead for all virtualization approaches, but this is the area where second generation hardware assisted virtualization will offer efficiency gains.

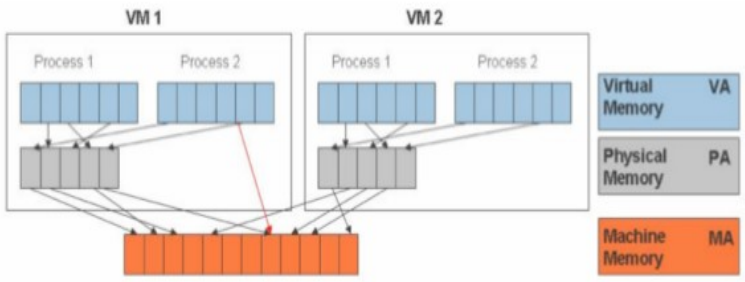


Figure 8 – Memory Virtualization