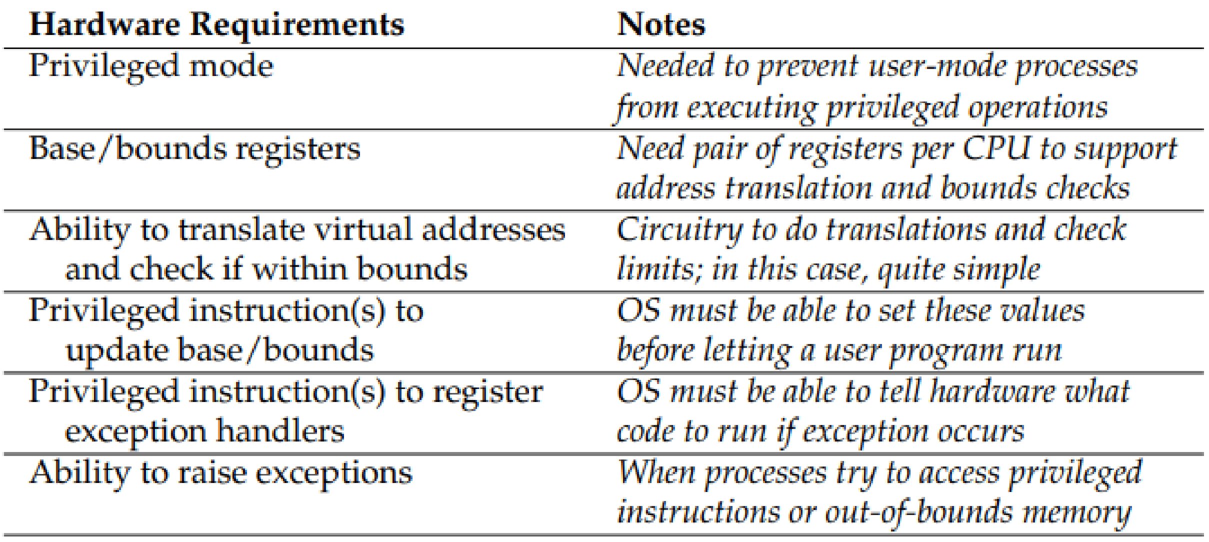
Memory Virtualisation

OSTEP Chapter 15 - Mechanism: Address Translation

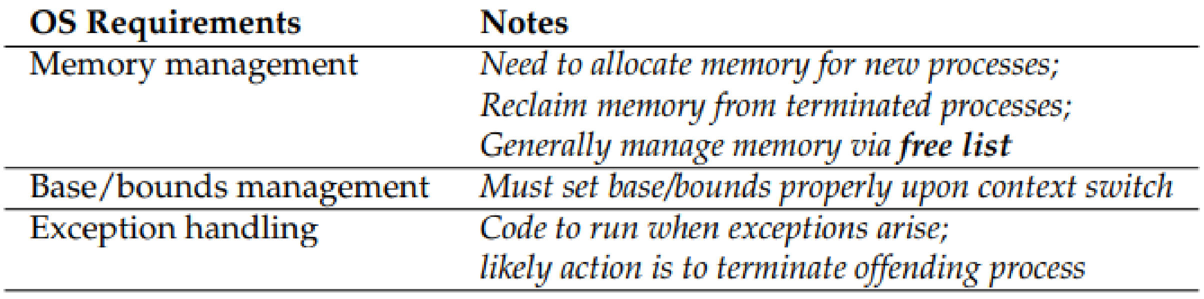
- Base and bound registers help to translate the virtual address to physical address
- Base and bounds registers are hardware structures kept on the chip (one pair per CPU). Sometimes people call the part of the processor that helps with address translation the memory management unit (MMU)
- The bounds register can either hold the size of the address space or the physical end address of the address space



Hardware Requirements	**Notes**
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

- First, the OS must take action when a process is created, finding space for its address space in memory
- OS using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32 KB
- Second, upon termination of a process, the OS must put its memory back on the free list, and cleans up any associated data structures as need be
- Third, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the process structure or process control block (PCB)

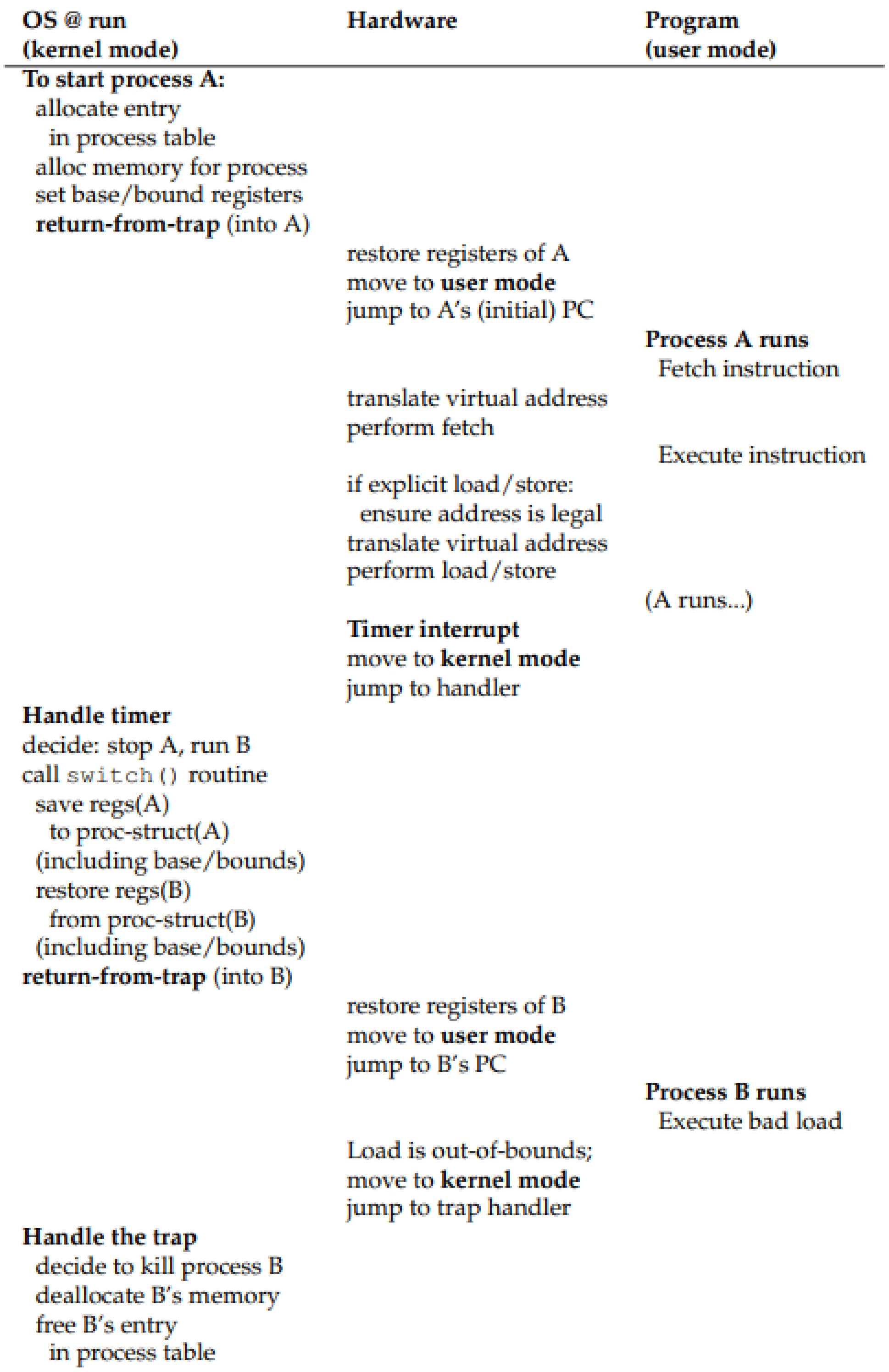


OS Requirements	**Notes**
Memory management	*Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise; likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

- We should note that when a process is stopped (i.e., not running), it is possible for the OS to move an address space from one location in memory to another rather easily. To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location. When the process is resumed, its (new) base register is restored, and it begins running again, oblivious that its instructions and data are now in a completely new spot in memory.

- Fourth, it should provide handlers for various faults or invalid memory accesses that a running process might make and likely terminate a process in such a case



OS @ run **(kernel mode)**	**Hardware**	**Program** **(user mode)**
To start process A: allocate entry in process table alloc memory for process set base/bound registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs Fetch instruction
	translate virtual address perform fetch	
		Execute instruction
	if explicit load/store: ensure address is legal translate virtual address perform load/store	
		(A runs...)
	Timer interrupt move to **kernel mode** jump to handler	
Handle timer decide: stop A, run B call `switch()` routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) **return-from-trap** (into B)		
	restore registers of B move to **user mode** jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to **kernel mode** jump to trap handler	
Handle the trap decide to kill process B deallocate B's memory free B's entry in process table		

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

- Because the process stack and heap are not too big, all of the space between the two is simply wasted. This type of waste is usually called internal fragmentation, as the space inside the allocated unit is not all used (i.e., is fragmented) and thus wasted

OSTEP Chapter 16 - Segmentation

- There is a big chunk of free space in the middle between stack and heap and thus base and bounds approach does not let us use the physical space so efficiently allowing us to schedule large number of processes or processes having large memory requirements
- To tackle this problem, the idea of **Segmentation** was proposed. The address space of a process is divided into logical segments namely code, stack and heap, and base-bound support for these segments is provided from hardware (i.e. extra registers in MMU)
- The hardware takes the virtual address being referred to and find a corresponding mapping and thus translates it into a physical address
- To find which segment an adress belong to, we can use the top 2 bits of the 14 bits VA, for e.g. to find the segment we are referring to, and use the corresponding base and bounds register for that segment
- In the implicit approach, the hardware determines the segment by noticing how the address was formed. If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment; if the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap
- For the stack, we require some extra hardware support to check whether the concerned segment grows in the positive or negative direction
- We first find the VA, its corresponding segment and its offset, subtract offset from the size of segment and then add this to the physical offset pointed by the base register, the absolute value of this base address should be less than the bounds register
- By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory
- Also the execute bit can be set to let the hardware know that this segment contains instructions which can be executed
- Support for fine-grained segmentation requires large size of segmentation tables in the MMU
- **What should the OS do on a context switch?** You should have a good guess by now: the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again
- a program may call malloc() to allocate an object. In some cases, the existing heap will be able to service the request, and thus malloc() will find free space for the object and return a pointer to it to the caller. In others, however, the heap segment itself may need to grow. In this case, the memory-allocation library will perform a system call to grow the heap (e.g., the traditional UNIX sbrk() system call). The OS will then (usually) provide more space, updating the segment size register to the new (bigger) size, and informing the library of success; the library can then allocate space for the new object and return successfully to the calling program. Do note that the OS could reject the request, if no more physical memory is available, or if it decides that the calling process already has too much
- The problem of allocating memory to the newly created processes. The memory gets filled of small holes called as **external fragmentation**.
- One solution to this is that OS can compact physical memory by rearranging existing segments
- Compaction may consume a large amount of time
- Compaction also (ironically) makes requests to grow existing segments hard to serve, and may thus cause further rearrangement to accommodate such requests

Summary

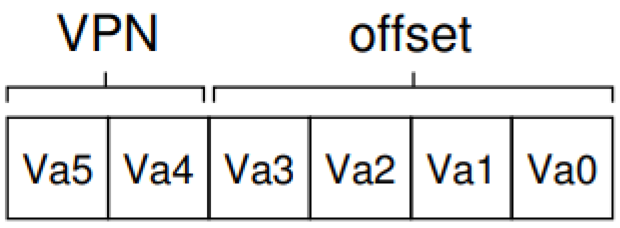
- Segmentation can better support sparse address spaces by avoiding huge waste of memory between logical segments
- Allows code sharing as well
- Leads to external fragmentation
- Still sparsely allocated heaps are a problem because whole heap segment has to be allocated some physical memory according to the rules of segmentation

OSTEP Chapter 18 - Paging: Introduction

- Segmentation leads to external fragmentation
- It may be beneficial to chop the physical memory into equal size pieces
- We thus divide the address space of a process into fixed size units called **pages**
- Physical memory is similarly divided into equal sized page frames
- Do not have to make assumptions as to how heap or stack may grow
- To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a **per-process** data structure known as a **page table**
- The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides

Address translation using page tables

- To translate the virtual address that the process generated, we have to first split it into two components: the virtual page number (VPN), and the offset within the page
- because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). Thus, our virtual address can be conceptualized as follows:



- Of this space, the first 2 bits signify which page the address belongs to and the remaining four bits indicate the offset in that page
- E.g. VA 21 = 010101, is 5th offset on page 1
- Use page table to find which page fram page 1 resides in and extact information stored at 5th offset in that page, in our case 7

Understanding page tables

- To store the page tables we don't have special dedicated hardware support inside MMU as page tables are very large in size. They are thus stored somewhere in memory
- For now, consider a page table to be a linear array, where VPN is the key and PTE stores the corresponding PFN
- Understanding PTE(s):
  - **valid bit** : whether the particular PTE is valid
    If a process tries to access entries having invalid entry, it will likely return a fault to OS thus

terminating the process We dot have the need to allocate physical frames corresponding to the entries marked as invalid
- **protection bits**: indicates whether the page can be read from, written to or executed from. In case of some invalid access, it will likely return in a page fault
- **present bit**: indicates whether the page is in physical memory or disk (i.e. **swapped out**). Swapping allows the OS to free up physical memory by moving rarely-used pages to disk
- **dirty bit**: indicates whether the page has been modified since it has been brought into memory
- **reference bit**: indicates whether a page has been accessed since it has been brought into memory, used to keep track of how frequently a page is visited

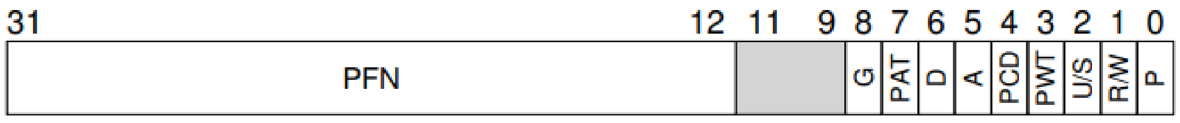


Figure 18.5: **An x86 Page Table Entry (PTE)**

- It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the page frame number (PFN) itself

Paging: Too Slow

- To fetch the desired data, the system must first translate the virtual address (21) into the correct physical address (117).
- Thus, before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory
- Assuming a single register stores the physical address of the page table base VPN = (VirtualAddress & VPN_MASK) >> SHIFT PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
- Finding the actual physial address

```
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

- Thus one memory reference leads to two memory accesses slowing down the process
- We thus have two problems to solve:
  - Large size tables
  - Extra memory accesses
- Physical address translation

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
```

```
    // Form the address of the page-table entry (PTE)
    PTEAddr = PTBR + (VPN * sizeof(PTE))

    // Fetch the PTE
    PTE = AccessMemory(PTEAddr)

    // Check if process can access the page
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        // Access is OK: form physical address and fetch it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```

Summary

- Does not lead to external fragmentation
- Flexible, enabling the sparse use of virtual address space
- Can lead to slow machine if not implemented properly
- Inefficient storage, memory will be filled with page table entries and less space will be available for storing application data itself

OSTEP Chapter 19 - Paging: Faster Translations (TLBs)

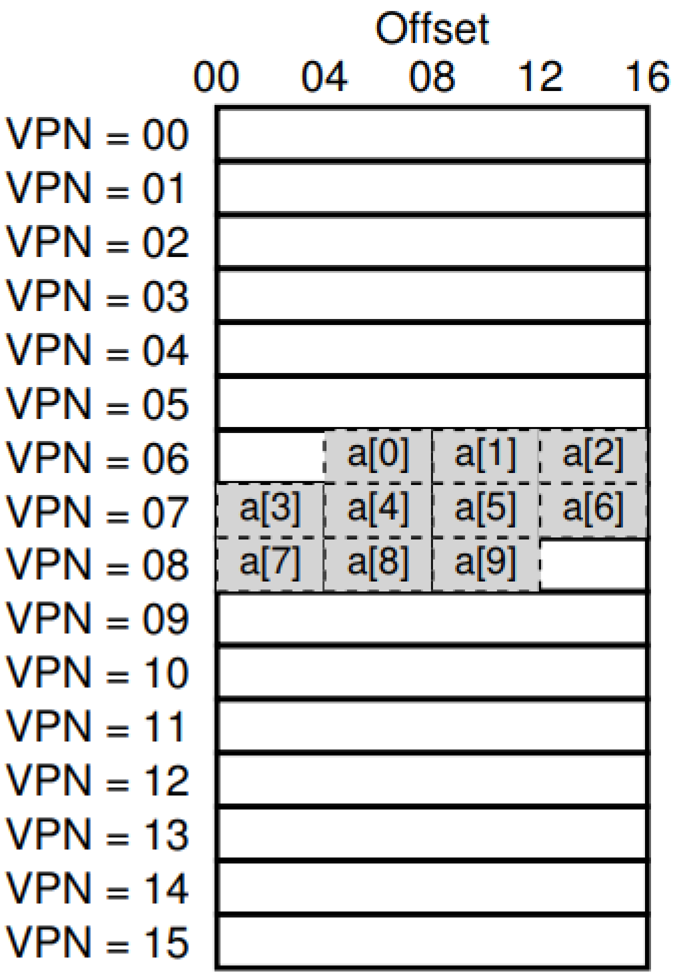
- To speed address translations, we are going to use **TLB (Translation-Lookaside Buffers)**.
- TLB is part of hardware's MMU
- The hardware first checks if the translation for the VA is present in TLB and then access the page table

TLB basic algorithm

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
```

```
            RaiseException(PROTECTION_FAULT)
        else
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
```

Accessing an array



- Accessing this array has 70% efficiency, there will be three misses, one per page
- As the size of page grows (generally 4Kb), even lesser TLB misses
- The TLB performance is this high because of **spatial locality.**
- If after 1 loop, the array is accessed once again the efficiency will be 100% given that the TLB is big enough to store enough entries

Who handles TLB misses?

- return-from-trap in case of TLB miss has to be different from the general TLB miss in the sense that it has to start from the same instruction that has led to a TLB miss. The general return-from-trap starts from the instruction following the instruction that has led to a TLB miss
- Remember not to cause looping while handling a TLB miss. This may occur if the TLB miss is for calling the TLB miss handler itself. Thus we can either keep these handlers in physical memory where

they are unmapped or we can allocate some special TLB entries for these handlers which save their translation permanently. These translations always end up in a **hit**
- The OS then updates the cache via a special instruction to make the next access to the VA a hit
- Software-based-caches are thus better as they can implement any data structure to implement the page table without making any change in the hardware

NOTE:
A valid TLB shows that the TLB entry is valid and can be used for translation whereas invalid bit means that the page table should be refferred to, to get the correct translation of this page. A valid page table entry on the other hand means that this page can be accessed and if it is not set, it means that the user has tried to access an invalid address.

- A TLB entry is of the form
  VPN | PFN | other bits

> - Note that both the VPN and PFN are present in each entry, as a translation could end up in any of these locations (in hardware terms, the TLB is known as a fully-associative cache)

- other bits:
  - **valid bit:** says whether the entry has a valid translation
  - **protection bits:**: says about the user/OS access to the page frame
  - address space identifiers and dirty bits as well may exist

TLB Issue: Context Switch

- The entries for 1 process may be different from that of the other
- We can flush the whole TLB on context-switch but that would lead to a lot of overhead on context-switch
- The other solution is to assign an **ASID(Address-Space Identifier)**
- The hardware finds the ASID of the current process set some register to hold the ASID value of the current process
- There may be two entries pointing to the same PFN, this may be because different processes may share some memory, generally code and thus may be pointing to the same PFN, this is a desired characteristic
- To replace an existing entry, we can either use **LRU** or **random policy** (performs better mostly).

TLB entries

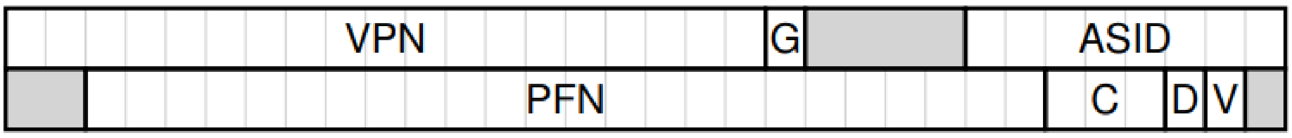


Figure 19.4: **A MIPS TLB Entry**

- The MIPS R4000 supports a 32-bit address space with 4KB pages.
- Expect 20-bit VPN and 12-bit offset in our typical virtual address. However, as you can see in the TLB, there are only 19 bits for the VPN; as it turns out, user addresses will only come from half the address space (the rest reserved for the kernel) and hence only 19 bits of VPN are needed.

- The VPN translates to up to a 24-bit physical frame number (PFN), and hence can support systems with up to 64GB of (physical) main memory
- global bit (G), which is used for pages that are globally-shared among processes. Thus, if the global bit is set, the ASID is ignored.
- 8-bit ASID, which the OS can use to distinguish between address spaces

> - what should the OS do if there are more than 256 2^8 processes running at a time?

- a dirty bit which is marked when the page has been written to
- a valid bit which tells the hardware if there is a valid translation present in the entry
- a few are reserved for the OS. A wired register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS; the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., in the TLB miss handler)
- We refer to this phenomenon as exceeding the **TLB coverage**, if the number of pages a process accesses exceeds the number of pages table entries that the TLB can hold

OSTEP Chapter 20 - Paging: Smaller Tables

- We have to store the page tables in less space as per-process storing the page tables will occupy a lot of memory
- Using larger page tables can reduce the number of entries in the page table per-process but this would at the same time allocate a bigger page for storing less data and thus may lead to internal fragmentation
- Hence, generally the size of pages used in systems is that of 4KB or 8KB

Hybrid Approach: Pages and Segments

-

OSTEP Chapter 21 - Beyond Physical Memory: Mechanisms

- We relax the assumption that the address space of every process always fits inside the physical memory
- The idea is that we assume we have infinitely large memory which is slower and we can swap pages in and out of physical memory to this swap area
- In operating systems, we generally refer to such space as **swap space**, because we swap pages out of memory to it and swap pages into memory from it
- The OS can read from and write to the swap space, in page-sized units
- Example

- 4-page physical memory and an 8-page swap space.

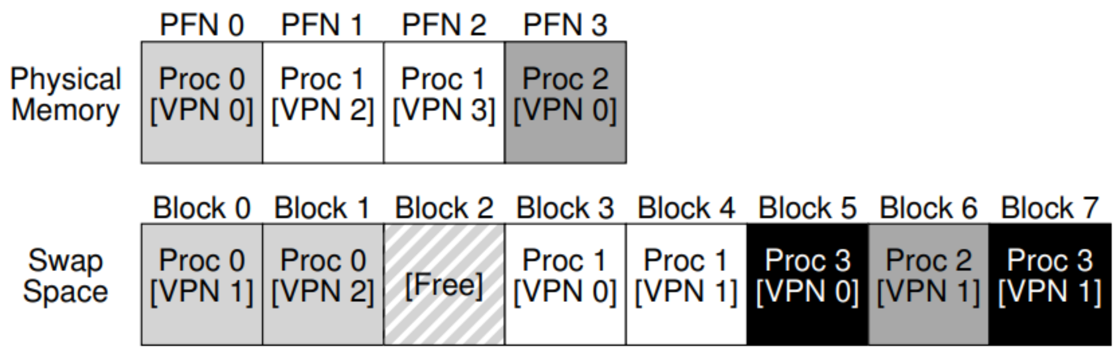


Figure 21.1: **Physical Memory and Swap Space**

  - Three processes (Proc 0, Proc 1, and Proc 2) are actively sharing physical memory; each of the three, however, only have some of their valid pages in memory
  - The rest located in swap space on disk
  - A fourth process (Proc 3) has all of its pages swapped out to disk, and thus clearly isn't currently running.
  - One block of swap remains free
  - Even from this tiny example, hopefully you can see how using swap space allows the system to pretend that memory is larger than it actually is
- We should note that swap space is not the only on-disk location for swapping traffic. For example, assume you are running a program binary (e.g., ls, or your own compiled main program). The code pages from this binary are initially found on disk, and when the program runs, they are loaded into memory
- Assuming a hardware-managed TLB, a VA access proceeds in the following fashion:
  1. Goes to the TLB, if finds the entry corresponding to the given VA valid and protection bits set, it returns
  2. Else goes to the page table, and accesses the PTE, if found valid and present in the memory, updates the TLB, else if not valid returns segmentation fault
  3. if not present but valid, calls the page-fault handler which finally loads the page back into the memory. Uses **present bit** to check this

Page Fault

- **In both hardware-managed and software-managed TLBs, if the page is not present in physical memory, the control goes back to the page-fault handler i.e. the OS**
- The OS uses the PTE to store the address of the page it has been swapped to in the disk and uses this address to access the disk and load its content to a new Page in memory, update the PFN to the new PFN and later set the valid and present bit to 1
- Note that while the I/O is in flight, the process will be in the blocked state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware

> - They call this as page fault, find the exact definition of a page fault and in what scenarios it may occur, why in assignment it was occuring for other cases as well?

- Hardware control flow for accessing the page

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

- OS page fault handler

```
PFN = FindFreePhysicalPage()
if (PFN == -1) // no free page found
    PFN = EvictPage() // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
PTE.present = True // update page table with present
PTE.PFN = PFN // bit and translation (PFN)
RetryInstruction() // retry instruction
```

- **To keep a small amount of memory free, most operating systems thus have some kind of high watermark (HW) and low watermark (LW) to help decide when to start evicting pages from memory. How this works is as follows: when the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available. The background thread, sometimes called the swap daemon or page daemon1 , then goes to sleep, happy that it has freed some memory for running processes and the OS to use**