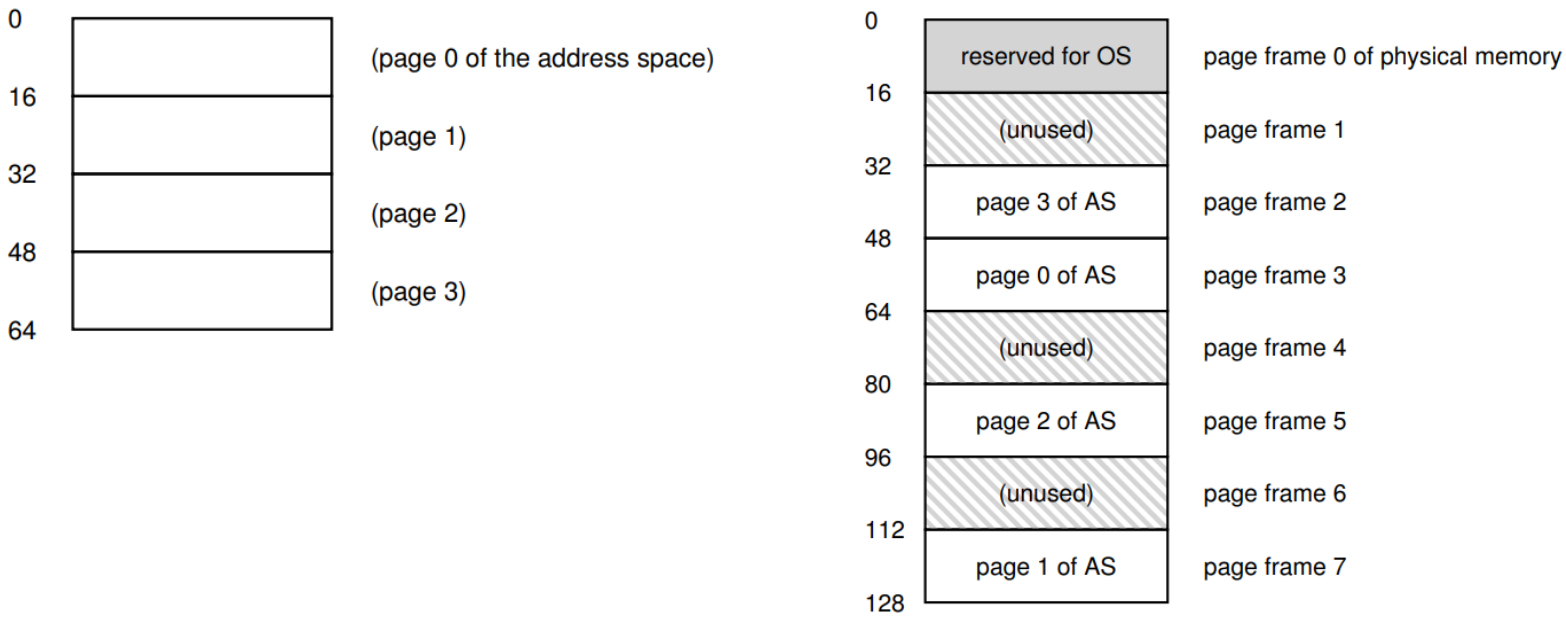
Paging

Conceptually Segmentation solves many problems to virtualize memory – however the varying size of the segments leads to complicated memory management and fragmentation.

The idea of *paging* dates back to early systems where instead of separate segments of variying sizes we rely on dividing the address space into small fixed sized allocation units, called pages (or page frame). A 64-byte (4-page x 16) address space and corresponding mapping to 128-bytes (8-page frames x 16) is shown here



Paging: ADVANTAGES

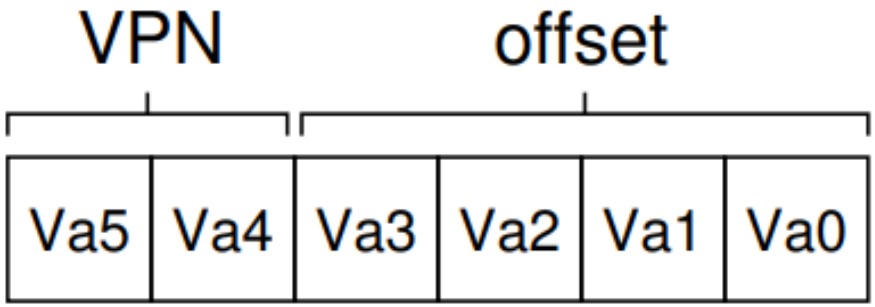
Paging, has a number of advantages over our previous approaches.

- Flexibility: we won't, for example, have to make assumptions about how the heap and stack grow and how they are used.
- Simplicity of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space from above into our 8-page physical memory, it simply finds four free pages; perhaps the OS keeps a free list of all free pages for this, and just grabs the first four free pages off of this list.

To record where each virtual page of the address space is placed in physical memory, the operating system 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 they live (VP 0 $\rightarrow$ PF 3; VP $\rightarrow$PF7; etc.)

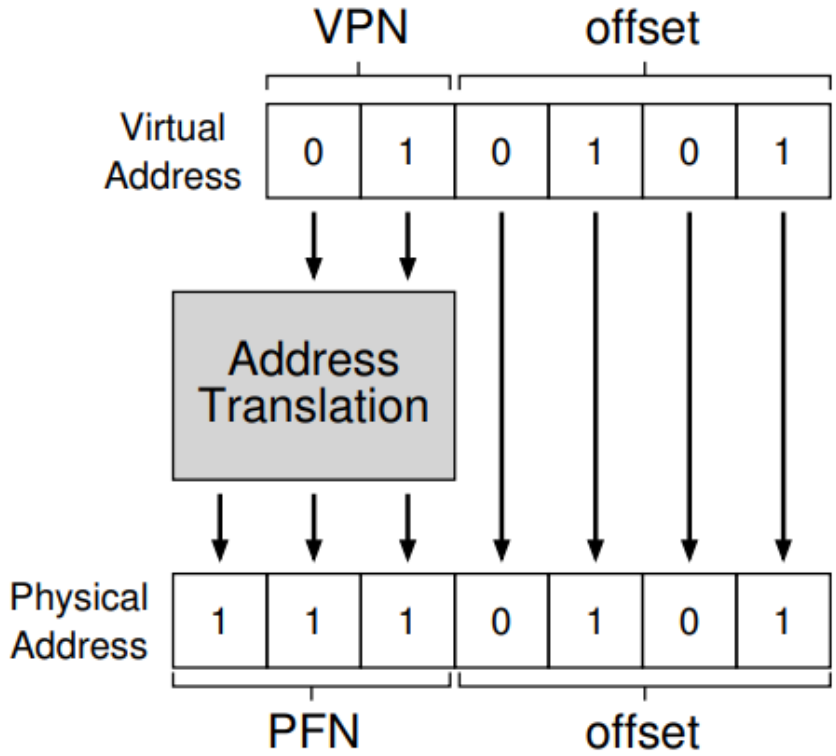
Page Tables

Note that the page-table is a per-process data structure (except the inverted page table). If another process were to run, the OS would have to manage a different page table for it, as its virtual pages obviously map to different physical pages.
Address Translation : Let us take the 64 address space (4-pages of 16 bytes each) as we have presented in the beginning. Length of an address would be 6-bits VA(0 to 5). We need two higher bits (VA5 and VA4) to represent the page and rest 4-bits (VA3, VA2, VA1 and VA0) for the offset within that page (see figure).



Page Tables ... contd.

Suppose we need translation of address 21. Note that it is in page 1 (VPN $= 1$) at an offset of 5 (see figure). The translation mechanism should be able to choose the correct physical page frame number (PFN) consulting the corresponding page-table entry (note that page VPN 1 is mapped to PFN 7). This precisely the role to be played by the page-table (PT). In its simplest form the PT could be an array and here index 1 would store 7.



Page Tables ... where to store.

For keep fragmentation at bay we go for more pages (smaller is better) – so we have big or very big page table to deal with. Question is where to keep it as it requires to be in main-memory for faster translation but main memory is a very important resource. Note,

- For a typical 32- bit machine with 4 KB page; we have
- 2^{20} pages. That is 20 bit of VPN and 12 bit offset. Now,
- with 4-bytes per page table entry (PTE) the size of the PT would be 4-MB for each process. With 100 processes running;
- there would be 400 MB of page table occupying a major portion of main memory; and thus not acceptable.

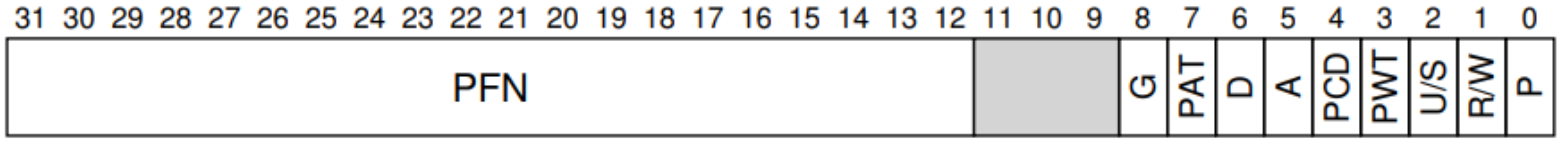
Page tables are so big we do not provide any special on-chip cache in the MMU to hold the PT of the currently running process.
Instead we save the page table for each process elsewhere.

Page Table Entry (PTE)

Primary purpose of the page table is to get the PFN from VPN. So, it can be any data structure; might be an array as well where VPN indexes to PTE to get PFN. However, in real-life PTE we find many bits to implement error checking, paging strategies, cache control and other administrative actions. Some of the common bits are

- Vaild (V) – indicates if the translation is valid or not
- Dirty (D) – tracking wheather the page is changed (written into) once it is brought to the memory.
- Present (P) – to know if the page is present in main memory or in disk
- Referenced (R) – A page has been accessed or not

Figure shows the contents of the PTE for X86-32 bit system. Note that the bits G, PAT, PCT, and PWT are for cache control. The U/S bits determines user access to the page while R/W allows writing to this page.



Paging is a slow process

Consider reading a memory location at virtual address 21 (like the previous example) of our 64-byte system with four 16-bytes pages. Note that the corresponding physical address is 117 (PFN 7 + offset of 5 = 7 x 16 + 5= 117). For translation locating the page table for the current process is done through PTBR (The page table base register – holds the physical base address of the page table for the current process). The PTE (or page table entry) address arithmetic is shown below.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
[Note that in our case  VPN_MASK is  0x110000 and the SHIFT is 4]
Offset does not require any translation and can be computed as
offset = VirtualAddress & OFFSET_MASK
and the physical address  can be extracted
from the PFN as PhysAddr = (PFN << SHIFT) | offset
```

Steps to get PA from VA in paging for memory access

```
// 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)
```

Now, for each memory reference we need 2 memory references. So, *paging* is a slow process by a factor of 2 (or more). Expediting the translation must be done.

Memory Trace

Let us see the memory access for paging using a memory trace for the following program (array.c)

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

Compilation and running the code in UNIX may be done using the command

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Finally the *objdump* of the array initialisation is shown below [For simplicity we have assumed all instructions are 4-bytes long; which is not true for x86]

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

Memory Trace ...contd.

- The first instruction $movl\ 0x0,\ (\%edi, \%eax, 4)$ moves 0 to the VA of the array. This is computed as *edi* holds the base address (assumed) of the array with which we add the contents of *eax* * 4 (a scale factor of 4 is used as it is an integer array)
- eax is increased by 1 ($incl\ \%eax$) to get the next array index; and it is
- compared with $0x03e8 = 1000$ using $jne\ 0x03e8, \%eax$
- if not we go back to repeat moving 0 else we move to the next instruction beyond the for loop

To understand which memory accesses this instruction sequence makes (at both the virtual and physical levels), we'll have to assume something about where in virtual memory the code snippet and array are found, as well as the contents and location of the page table.

Memory Trace ...contd.

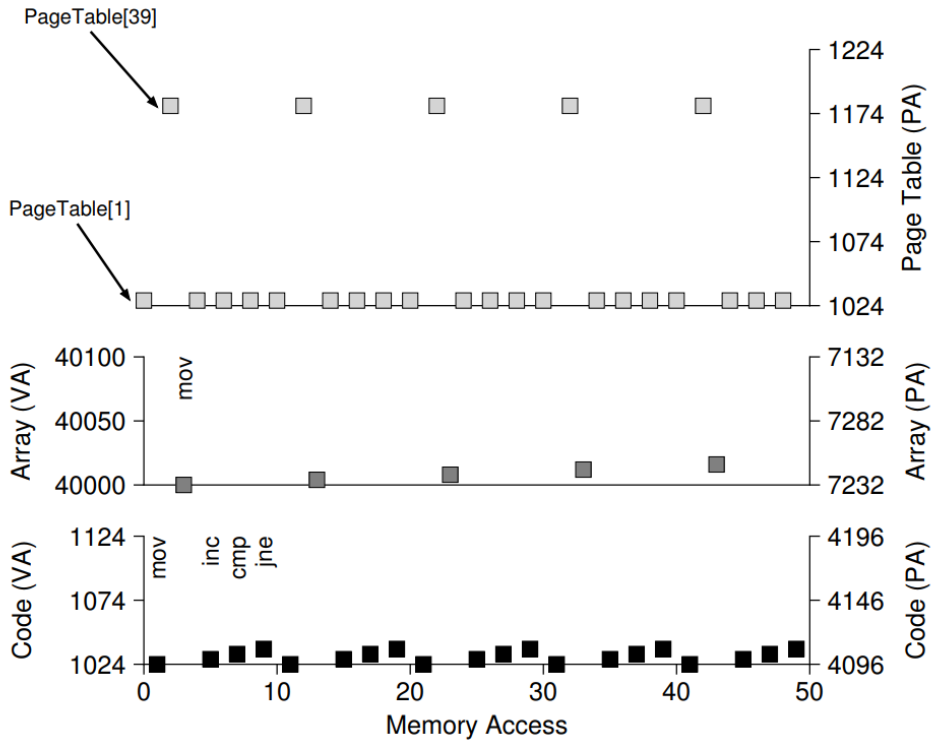
- Assume virtual address space of size 64 KB and a page size of 1 KB.
- page table is a linear array starting at PA location 1 KB = 1024
- the code lives on location 1024 (see the address of the 1st instruction) of VA. So, it is on VPN 1 (VPN 0 is for 0 to 1023). Also, assume
- VPN 1 maps to PFN 4. Let that the array (4000 bytes) and starts from VA 40000 and ends at 40000+1000*4-1. The corresponding VPN is from 39 to 42; also assume
- VPN 39 ->PFN 7; VPN 40->PFN 8; VPN 41 ->PFN 9; VPN 42->PFN 10

During execution

- each instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides within, and one to the instruction itself to fetch it to the CPU for processing.
- one explicit memory reference in the form of the mov instruction; this adds another page table access first (to translate the array virtual address to the correct physical one) and then the array access itself

Memory Trace ...contd.

The bottom one shows the instruction memory references on the y-axis (with VAs on the left, and the actual PAs on the right); the middle one shows array accesses; finally, the topmost graph shows page table memory accesses (just physical, as the PT in this example resides in PM). The x-axis, for the entire trace, shows memory accesses across the first five iterations of the loop (there are 10 memory accesses per loop, which includes 4 instruction fetches, 1 explicit update of memory, and 5 page table accesses to translate those 4 fetches and 1 explicit update).



Paging : 1st Summary

We got the basic concept of paging which is advantageous than segmentation as it offers

- Lower external fragmentation, as paging is intentional break-up of the free space into small fixed size chunks
- It is quite flexible, enabling the sparse use of virtual address spaces.

However, paging has its blemishes as well; particularly if it is implemented carelessly

- May lead to slower access
- Waste of memory to hold the (big) page tables

TLB – contd.

Paging can be the basic mechanism for the implementation of virtual memory. However, multiple memory access required for translation and keepin the page table small or avoiding storage of the big Page table in main memory are the main challenges.

- A special hardware cache named **Trnaslation Lookaside Buffer** (TLB) is used for faster translation.
- A TLB (though a better name is perhaps the address translation cache or ATC) is a part of any modern MMU.
- A virtual address is first searched in TLB to check the presence of the desired translation (the part of PA to be precise) – this reduces extra memory read and the need to consult the page table.
- TLB miss is costly, however for a well designed system Hit ratio may be more than 80%.
- TLB makes paging effective

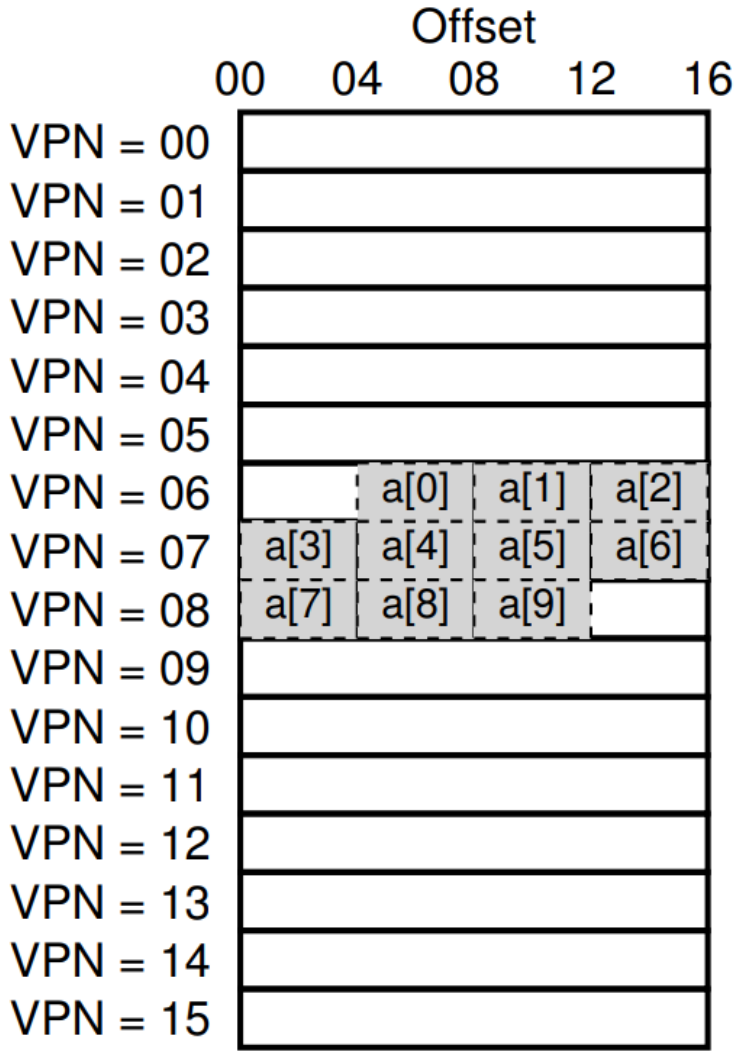
TLB – Logical Flow

```
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
        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()
```

TLB – access example

let's examine a simple virtual address trace and see how a TLB can improve its performance. Assume,

- we have an array of Ten (10) 4-byte integer, starting at virtual address 100
- a small 8-bit virtual address space, with 16-byte pages;
- virtual address breaks down into a 4-bit VPN (there are 16 virtual pages)
- and a 4-bit offset (16 bytes per pages).
- Array elements are stored accross 3 virtual pages (6, 7 and 8)



TLB – access example ... contd.

Assume an example of memory access as shown below; pretend the sum requires the memory access and other (like i etc.) are not.

```
sum += a[i]; // i varies from 0 to 9
```

Considering the array access trace note that it would be

```
MISS HIT HIT  MISS HIT HIT HIT MISS HIT HIT
```

The first reference i.e., a[0] would result in a TLB miss – for the next two i.e. a[1], a[2] are hits as they are in the same VP. a[3] results in a MISS followed by three consecutive hits for a[4] to a[6]. Next is again a miss as a[7] is in next VP; followed by 2 hits (for a[8] and a[9]). So, out of 10 accesses there will be 3 misses or a 30% penalty or a 70% hit rate (as array elements are tightly packed – spatial locality). Naturally, the number of hits would be more for a real life VP containing many more data items packed in the same VP. Note that for another array access (a[0] to a[9]) after the first one we will have 0 misses (due to temporal locality)

TLB ... Handling miss

A TLB miss severly reverses the gain in having TLB unless we manage the miss penalty is an efficient way. We have two options i) h/w based or ii) s/w based solution. In a h/w based solution (like x86 architecture) we have

- One page table base register (PTBR: CR3 register in X86 – that uses fixed multi level PT) to know where the PTs are located in memory.
- A miss in the TLB leads to the h/w searching through the PT,
- finds the correct page-table entry and extracts the desired translation,
- updates the TLB with the translation,
- and retry the instruction.

The drawback with the h/w approach is that it is not very flexible and you cannot have any data structure to implement the PT.
Many modern CPUs (MIPS R10000 and SUN SPARC V9) thus offer s/w based TLB misses. Note that in case of s/w handling of TLB the OS simply raises an exception on TLB miss the rest is handled by s/w.

Handling miss ... s/w controlled

To get more flexibility TLB miss may be handled in s/w and most modern CPU does this way at the cost of slightly more translation time. The event chains are as follows

- On a miss a hardware exception is raised which pauses the current instruction stream, raises the privilege level to kernel mode, and
- jumps to a trap handler that is specially written for handling of TLB misses.
- the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap;
- at this point, the hardware re-executes the instruction (resulting in a TLB hit) that causes the miss.

It may be noted that the return from TRAP this time is different than the normal TRAP where simply the next instruction in the instruction stream is executed (like a procedure return). Here, the instruction which causes the TRAP has to be executed. So, in this case a different PC value needs to be stored by the h/w so as to re-run the instruction that causes the miss.
The s/w approach offers flexibility and allows any data structure to implement the PT without any concern for the underlying h/w.

Contents of a TLB

The TLB (h/w) usually contains 32 to 128 entries and must be in the form of a fully associative cache form for parallel searching of all entries as any translation can be stored at any location. A typical TLB entry might look like

```
VPN   |  PFN   | other bits
```

The bits coming under the 'other' category are

- Valid bit – indicating that it is a valid translation or not.
- Protection bits – dictate how a page would be accessed (for example code should be r-x and heap would be rw-)
- Address space identifies or ASID (elaborated later)
- dirty bits; etc.

Note that context-switching needs be handled carefully as TLB contains the translation of VPN to PFN for the currently running process. On a switch the OS must be careful not to use traslation for the new process done for some other old process(es). We will examine this issue next.

Context switch

The TLB contains virtual-to-physical translations that are only valid for the currently running proces (say P1). As a result, when switching from one process to another (say, P2) , the hardware or OS (or both) must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

- Assume, VPN 10 of P1 is mapped to PFN 100.
- OS soon performs a context switch and run P2 Assume, VPN 10 of P2 is mapped to PFN 170
- If entries for both processes were in the TLB, the contents of the TLB would be:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

Context switch ... contd.

The problem here is that the same VPN is translated to multiple PFNs. VPN 10 translates to either PFN 100 (P1) or PFN 170 (P2), but the hardware can't distinguish which entry is meant for which process.
There are a number of possible solutions to this problem.

- flush the TLB on context switches, thus emptying it before running the next process.
- as a process will never accidentally encounter the wrong translations in the TLB.
- However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages. If the OS switches between processes frequently, this cost may be high.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an *address space identifier (ASID)* field in the TLB.

ASID in Context switch

- ASID is similar to a process identifier (PID), but usually it has fewer bits (e.g., 8 bits for the ASID versus 32 bits for a PID) in the context of TLB.
- The h/w also needs to know which process is currently running and thus the OS must, on a context switch, set some privileged register to the ASID of the current process.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

- Assume two processes share a page (a code page, for example). Process 1 is sharing physical page 101 with Process 2; P1 maps this page into the 10th page of its address space, whereas P2 maps it to the 50th page of its address space.
- Sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

Replacement Policy

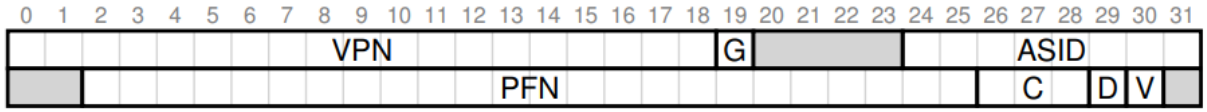
Like any other cache TLB requires replacement policy. Note with a new entry being made in the TLB an old one is replaced. Certainly the focus would be to increase the hit rate,

- One common approach is to evict the least-recently-used or LRU entry. The idea here is to take advantage of locality in the memory-reference stream; thus, it is likely that an entry that has not recently been used is a good candidate for eviction as (perhaps) it won't soon be referenced again.
- Another typical approach is to use a random policy. Randomness sometimes makes a bad decision but has the nice property that there are not any weird corner cases like a loop accessing n+ 1 pages, a TLB of size n, and an LRU replacement policy.

A real TLB entry : MIPS

We inspect a real TLB entry used for MIPS R4000 processor. It is a s/w controlled TLB that uses 64-bit TLB entry.

- R4000 supports 32-bit address space with 4-KB pages. So, a 20-bit VPN and 12-bit offset would be expected for a typical Virtual address.
- However, a 19-bit VPN is used which is half the address space reserved for the user (rest is for the kernel). The VPN translates to a 24-bit PFN; i.e., supporting a physical address of 64 GB (2^{24} 4KB pages)
- A global bit (G) indicates globally shared pages and ignore the ASID
- A Dirty bit (D) indicates the page is changed (written into). A valid bit V incates a valid translation present in the entry.
- Three cache coherence bits (Cs) are also used to determine how a page is cached.
- Page mask bits (not) shown are also used to change the page size. Some bits are reserved and not shown here
- Priviledged instructions like TLBP, TLBR, TLBWI and TLBWRare available to mange the TLB by the OS.



TLB: Summary

A dedicated on-chip TLB as an address-translation cache reduces the Page table search in most cases. Thus, in the common case, the performance of the program will be almost as if memory isn't being virtualized at all and certainly essential to the use of paging in modern systems.
However, TLB has problems as well

- If the no. of pages used by the program exceeds the no. of pages the TLB can hold in a short period of time there will be a large number of TLB misses (this is known as the TLB coverage problem) – To increase the TLB coverage larger page size can be used
- For physically-indexed-caches TLB access can be troublesome for the CPU pipeline

Smaller Tables

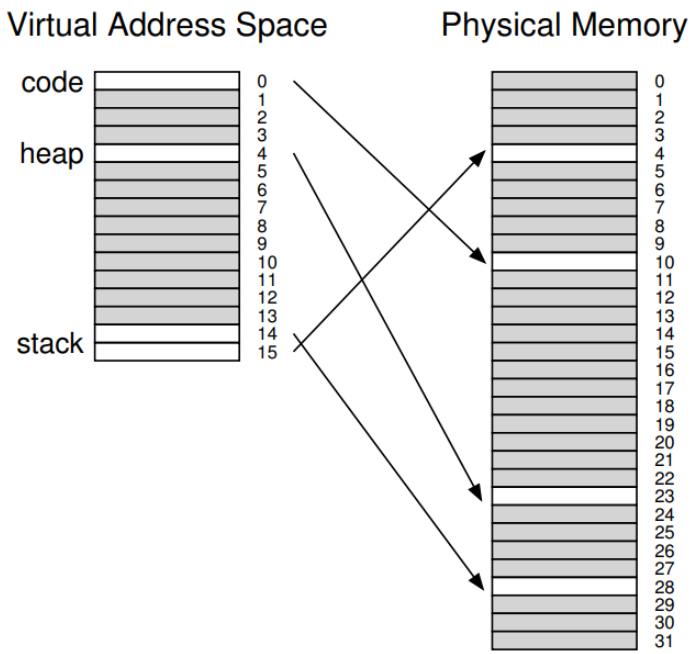
Page tables are too big and consumes a lot of memory. As an example for a 32-bit address space and 4 KB pages we will have $2^{32}/2^{12}$ or 1 MB Pages. For a 4-byte PTE thus we need 4MB space in the memory. Page table is per process data structure and we may have 100 (or more) active processes. So, what is the solution? A simple approach could be –

- Use big pages (instead of 4 KB say 16 KB or more)
- So, the number of pages reduced by a factor of 4.
- However, the page size increases by a factor of 4 and leads to a lot of internal fragmentation and this simple approach fails.
- Thus most systems use relatively small pages (4KB is quite common; e.g., X86 or 8KB as in SPARCv9)

What about using a hybrid approach of segments that are paged – this is used in MULTICS; the forerunner to UNIX.

Wastage in PT

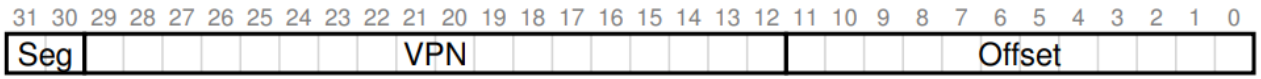
Consider a small 16 KB address space with 1 KB page; also a process with small heap and stack space in use. An example of the mapping to a 32 KB space and the corresponding PT are given below. What a considerable waste; most of the PTE entries are just invalid.



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

A hybrid approach

We may use segments (say, code, data and stack) for each process and maintain 3 page tables instead of a single PT mapping all the address space for the process. There would be 3 separate limit/bound register as well. In segmentation we use a base register to hold the address of the segment. Here, the base points to the physical address of the PT of that segment. The bound register indicates the number of pages in that segment. In the hardware, assume that there are thus three base/bounds pairs, one each for code, heap, and stack. When a process is running, the base register for each of these segments contains the physical address of a linear page table for that segment; thus, each process in the system now has three page tables associated with it. On a context switch, these registers must be changed to reflect the location of the page tables of the newly running process. For a 32-bit address and 4 KB page we may have 4 segment (code, stack, heap and unused) with the division of the address space as



A hybrid approach ... contd.

On a TLB miss (assuming a hardware-managed TLB, i.e., where the hardware is responsible for handling TLB misses), the hardware uses the segment bits (SN) to determine which base and bounds pair to use. The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry (PTE):

```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

We still have the problems

- of inflexibility – as we are using segments
- for a large but sparsely-use heap leads to a lot of page table wastage.
- external fragmentation to arise again.
- While most of memory is managed in page-sized units, page tables now can be of arbitrary size (in multiples of PTEs). Thus, finding free space for them in memory is more complicated.

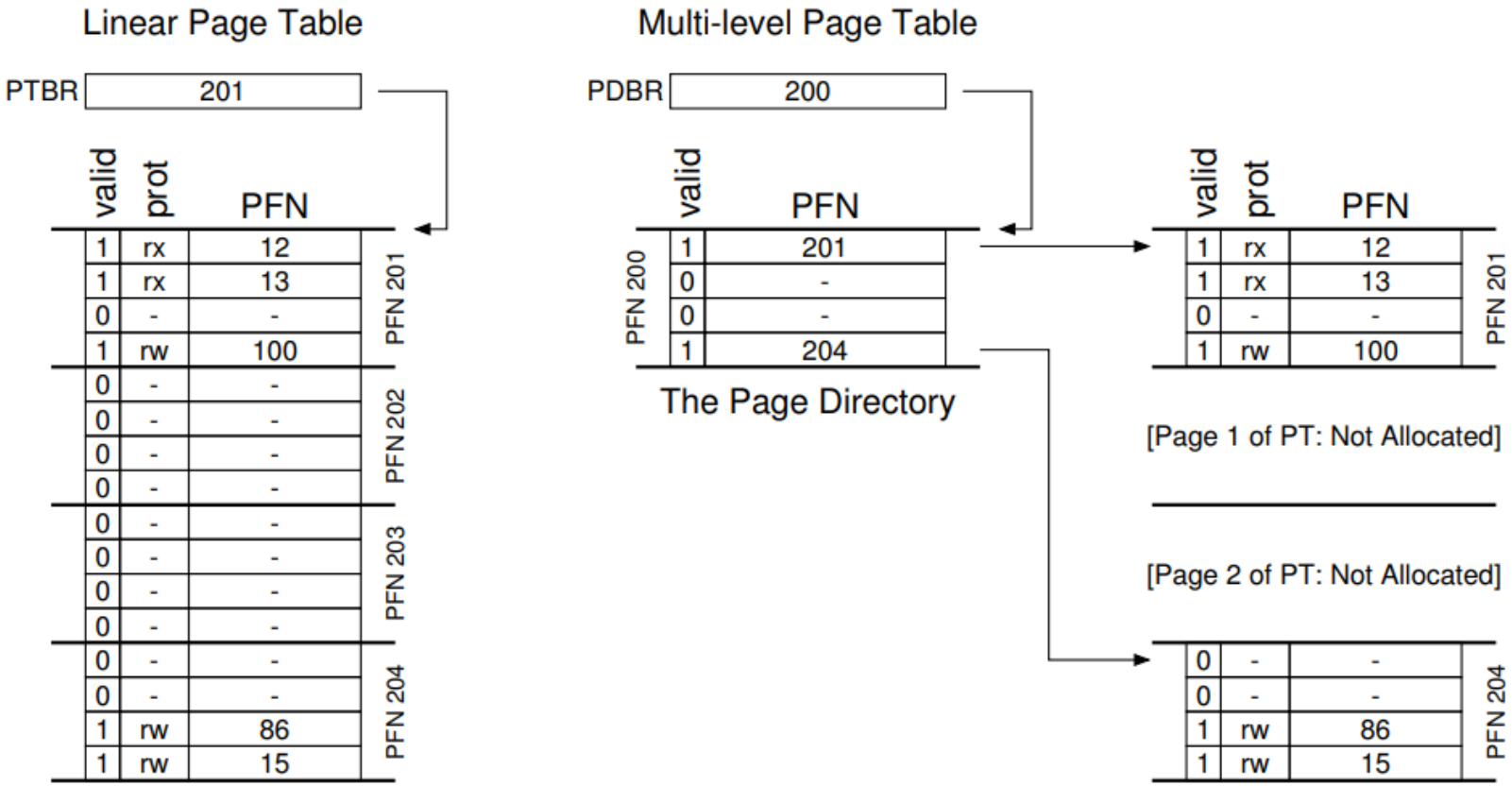
Multi-level page table

Paging approach is fine but how to get rid of all those invalid regions in the page table instead of keeping them all in memory is the crux of the problem. The solution is a multi-level (X86 uses it) page table, as it turns the linear page table into something like a tree. The basic idea behind a multi-level page table is simple.

- First, chop up the page table into page-sized units; then,
- if an entire page of PTEs is invalid, don't allocate that page of the page table at all.
- To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the page directory.
- The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Multi-level page table ... contd.

The page directory marks just two pages of the page table as valid (the first and last); thus, just those two pages of the page table reside in memory. And thus you can see one way to visualize what a multi-level table is doing: it just makes parts of the linear page table disappear (freeing those frames for other uses), and tracks which pages of the page table are allocated with the page directory.



Multi-level page table ... contd.

The page directory consists of a number of page directory entries (PDE). A PDE (minimally) has a valid bit and a PFN, if the PDE entry is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid, i.e., in at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to one. If the PDE entry is not valid (i.e., equal to zero), the rest of the PDE is not defined. Multi-level page tables have advatages.

- First, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces.
- Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table.

In contrast the size of a linear PT is 4MB (32-bit address, 4KB page and 4-bytes per PTE) finding such a large chunk of unused contiguous free physical memory can be quite a challenge. With multi-level we have added one extra level of indirection through the page-directory that enable us to place the pages of the PTs anywhere in memory.

Multi-level page table ... contd.

Multilevel page table approach has its pitfalls as well

- on a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table.
- here the complexity is more. On a TLB miss, page table look-up is more involved than a simple linear page-table lookup.

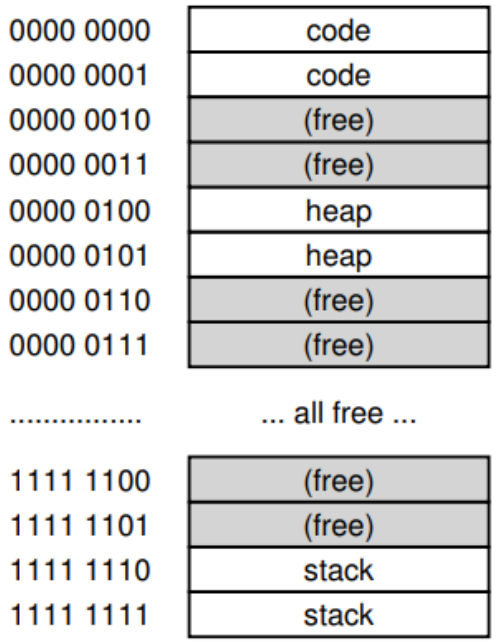
Multi-level table is an example of a time-space trade-off. We wanted smaller tables (and got them), but not for free; although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

Often we are willing to increase complexity in order to improve performance or reduce overheads; in the case of a multi-level table, we make page-table lookups more complicated in order to save valuable memory.

Multi-level PT ... an example

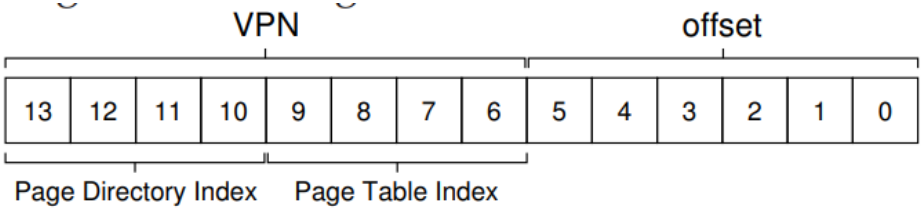
Consider an address space of size 16 KB, with 64-byte pages. So, a 14-bit VA space, with 8 bits for the VPN and 6 bits for the offset. A linear page table would have $2^8 = 256$ entries, even if only a small portion of the address space is in use. In this example, virtual pages 0 and 1 are for code, virtual pages 4 and 5 for the heap, and virtual pages 254 and 255 for the stack; the rest of the pages of the address space are unused.

To build a two-level page table for this address space, we start with our full linear page table and break it up into page-sized units. PT has 256 entries; assume each PTE is 4 bytes in size. Page table is 1KB (256 $\times$ 4 bytes) in size. Given that we have 64-byte pages, the 1-KB page table can be divided into 16 64-byte pages; each page can hold 16 PTEs.



Multi-level PT ... an example

VA 4:4:6 for 16 directory entries:16 Page Table entries : 64 byte Page



Page Directory		**Page of PT (@PFN:100)**			**Page of PT (@PFN:101)**		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	–	0	—
——	0	23	1	r-x	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	80	1	rw-	–	0	—
——	0	59	1	rw-	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	–	0	—
——	0	–	0	—	55	1	rw-
101	1	–	0	—	45	1	rw-

Multi-level PT ... Translation

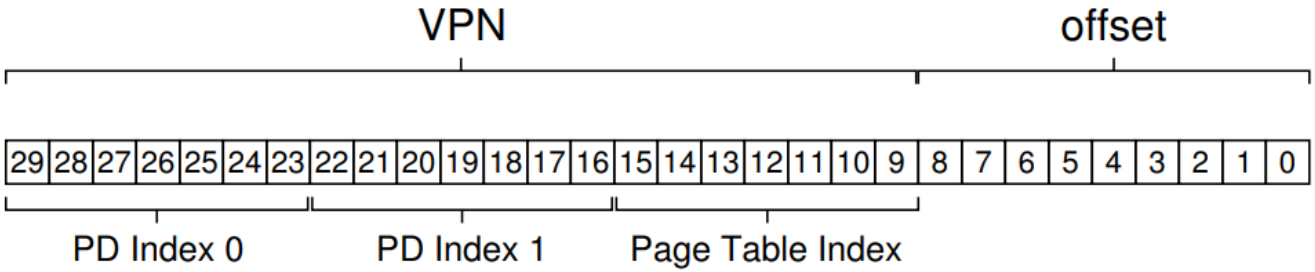
- PFN 100 and 101 are the two valid frames contained in the page directory. PFN 100 contains the mapping of the first 16 VPNs. The page has 4 valid entries (0, 1 for code and 4, 5 for heap) PFN 101 contains the mapping of the last 16 VPNs. And the valid entries are in our example, VPNs 254 and 255 (the stack segment).
- Multilevel structure reduces waste as instead of 16 pages (linear PT) here we use only 3: 1 for the PD, and two for the chunks of the PT that have valid mappings. The savings for large (32-bit or 64-bit) address spaces is much greater.
- Translation: Take the address 11 1111 1000 0000. The 8-bit from left indicates 254 (0x3F8). The page directory index (MSnibble; i.e., bit 10 to bit 13) is 1111 or 15 in decimal points to PFN 101. And the Page Table Index (bit 6 to 9) is 1110 or 14 gives to 55 (0x37) and tells us that page 254 of our virtual address space is mapped at physical page 55.
- By concatenating PFN=55 (or hex 0x37) with offset=000000, we can thus form our desired physical address and issue the request to the memory system: PhysAddr = (PTE.PFN << SHIFT) + offset = 00 1101 1100 0000 = 0x0DC0.

Multi-level PT ... More than two level

In some cases we need more than 2-levels to store the page directory in a page. Let's take a simple example

- A 30-bit VA and 512 byte page. So, we have 21-bit virtual page component and 9-bit offset.
- For a 4-byte PTE we can store 512/4=128 entries in a single page and thus Page table index would be 7-bit long.
- So, the page directory index would be 30-(9+7) = 14-bit long. So, we need $(2^{14}/2^9) \times 4 = 128$ pages to hold the PD.
- Thus a the basic idea of fitting all the components (PD and PT etc) cannot be achieved.

The solution lies by having multi-component PDs. By using multi-level PD (each fits into a page) we have a nice solution albeit increased translation complexity and more penalty on TLB miss.



Page table ... swapping

So far our assumption is that the PTs are always kept in some kernel physical memory space. Even with our many tricks to reduce the size of page tables, it is still possible, however, that they may be too big to fit into memory all at once. Thus, some systems place such page tables in kernel virtual memory, thereby allowing the system to swap some of these page tables to disk when memory pressure gets a little tight.

Summary: We have now seen how real page tables are built; not necessarily just as linear arrays but as more complex data structures. The trade-offs such tables present are in time and space – the bigger the table, the faster a TLB miss can be serviced, as well as the converse – and thus the right choice of structure depends strongly on the constraints of the given environment.

BEYOND PHYSICAL MEMORY

So, far the tacit assumption is that all the address spaces of all the active processes do fit in the physical memory as in reality we need to support many processes with huge address space.

- Thus the OS needs to stash at least a part of the big address space that are not being used in some suitable device.
- Naturally that device should have the capacity more than the physical memory and on-demand the stored things need be brought to the main memory in a reasonable time. This leads to the natural choice – the disks (modern systems rely more on electronic disks for obvious reasons)
- So, the OS provides an illusion of unlimited memory space that holds a super big address space bigger than the physical memory.
- This is usually achieved by using a part of the disk as swap space which holds pages to be moved in-and-out to and from the physical memory pages.

The swap space

The idea is to extend the memory by providing more space in a low cost high capacity (low speed) device. Some salient points ...

- the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the disk address of a given page.
- The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time. For the time being it may be cosidered very large. [Note that code/RO data binaries are also swapped in and out from main memory to their respective disk file locations (acting as natural swap space)].

The figure shows an example of 4 physical memory pages (shared by process 0, 1 and 2) versus 8-page swap space (part of process 0, 1, and 2 are present in the swap space, 1 page is free and process 3 is fully stashed here and not active)