

Paging

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseau

Moshe Sulamy

Tel-Aviv Academic College

Role of Hardware

- Role of hardware:
 - CPU provides **privileged mode**
 - Privileged instruction sets **base, bound**
 - **MMU** translates every memory access
 - MMU generates **exception** to OS on illegal access

Role of OS

- Role of OS:
 - Maintain free list of memory
 - Allocate and free space to processes
 - Maintain where space is allocated (in PCB)
 - Set translation information (base & bound)
 - Update information on context switch
 - Handle trap due to illegal memory access

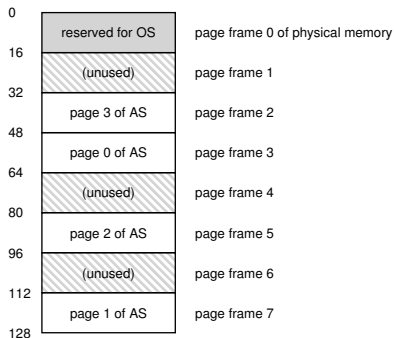
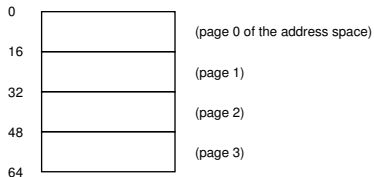
Paging

- Consider another approach:
 - Divide space into fixed-size pieces
 - Each fixed-size unit is a **page**
 - Divide physical memory into **page frames**

How can we virtualize memory with pages to avoid the problems of segmentation?

Paging

- For example:
 - 64-bytes address space, 16-byte pages, 128-bytes RAM



- Advantages:
 - **Flexibility:** abstracting address space effectively
 - **Simplicity:** ease of free-space management

- **Page table**

- Records where each virtual page is placed in physical memory
- Per-process structure
- **Address translation** for virtual pages

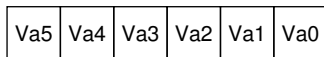
Paging

- To **translate** a virtual address:
 - Split into **virtual page number (VPN)** and **offset**
 - For 64-bytes, virtual address size is 6 bits:

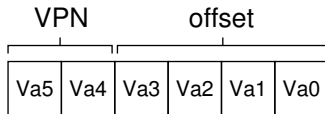
Va5	Va4	Va3	Va2	Va1	Va0
-----	-----	-----	-----	-----	-----

Paging

- To **translate** a virtual address:
 - Split into **virtual page number (VPN)** and **offset**
 - For 64-bytes, virtual address size is 6 bits:

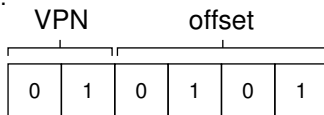


- Address space is 4 pages, thus:



Paging

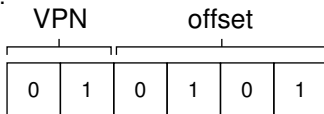
- Let's translate an address!
 - Virtual address 21:



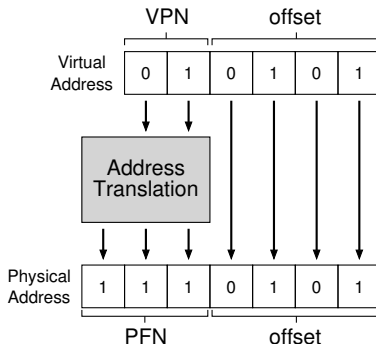
Paging

- Let's translate an address!

- Virtual address 21:



- 5th byte of page 1 → **physical frame number (PFN) 7**:



Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes			

Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes	4		
1KB			

Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes	4		
1KB	10		
1MB			

Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes	4		
1KB	10		
1MB	20		
512 bytes			

Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes	4		
1KB	10		
1MB	20		
512 bytes	9		
4KB			

Where Are Page Tables Stored?

Page Size	Low Bits		
16 bytes	4		
1KB	10		
1MB	20		
512 bytes	9		
4KB	12		

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	
1KB	10	20	
1MB	20	32	
512 bytes	9	16	
4KB	12	32	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	6
1KB	10	20	
1MB	20	32	
512 bytes	9	16	
4KB	12	32	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	6
1KB	10	20	10
1MB	20	32	
512 bytes	9	16	
4KB	12	32	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	6
1KB	10	20	10
1MB	20	32	12
512 bytes	9	16	
4KB	12	32	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	6
1KB	10	20	10
1MB	20	32	12
512 bytes	9	16	5
4KB	12	32	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits
16 bytes	4	10	6
1KB	10	20	10
1MB	20	32	12
512 bytes	9	16	5
4KB	12	32	20

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	
1KB	10	20	10	
1MB	20	32	12	
512 bytes	9	16	5	
4KB	12	32	20	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	
1MB	20	32	12	
512 bytes	9	16	5	
4KB	12	32	20	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	
512 bytes	9	16	5	
4KB	12	32	20	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	4K
512 bytes	9	16	5	
4KB	12	32	20	

Where Are Page Tables Stored?

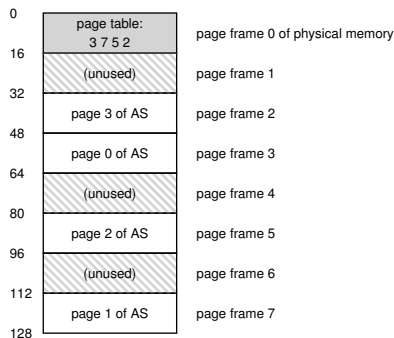
Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	4K
512 bytes	9	16	5	32
4KB	12	32	20	

Where Are Page Tables Stored?

Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	4K
512 bytes	9	16	5	32
4KB	12	32	20	1M

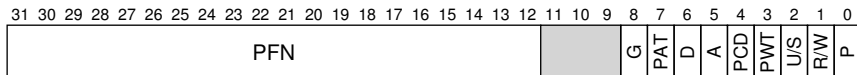
Where Are Page Tables Stored?

- Typical 32-bit address space with 4KB pages:
 - 20-bit VPN and 12-bit offset
 - 4 bytes per **page table entry (PTE)**
 - 4MB of memory for each page table
- 100 processes: 400MB just for address translations!



What's Actually In The Page Table?

- Simplest form: **linear page table**
 - Array, indexed by VPN, looks up PTE to find PFN
 - **Valid bit**: whether the translation is valid
 - Unused space is **invalid**, access will trap into OS
 - Sparse address space: no physical frame for invalid
 - **Protection bits**: read, write, execute
 - **Present bit**: in physical memory or disk (discussed later)
 - **Dirty bit**: whether page has been modified
 - **Reference bit**: indicating page has been accessed



Issues

- Too slow
 - To find PTE, starting location of page table is needed
 - Page table too big to store in MMU
 - Extra memory reference for every memory reference
- Internal fragmentation
 - Page size may not match needed size
- Takes up too much memory

Issues

- Too slow
 - To find PTE, starting location of page table is needed
 - Page table too big to store in MMU
 - Extra memory reference for every memory reference
- Internal fragmentation
 - Page size may not match needed size
- Takes up too much memory

How can we speed up address translation,
and avoid the extra memory reference?

- **Translation-lookaside buffer**
 - Part of the **MMU**
 - Hardware **cache** of popular translations
- On each memory reference:
 - **TLB hit**: translation performed quickly
 - Extract PFN for VPN in hardware
 - **TLB miss**: consult **page table**
 - Update the **TLB**
 - Retry the instruction

- Example: accessing a 10 elements array in a loop

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- Example: accessing a 10 elements array in a loop

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
- 3 misses and 7 hits

TLB

- Example: accessing a 10 elements array in a loop

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
- 3 misses and 7 hits

TLB **hit rate** is 70%

TLB

- Example: accessing a 10 elements array in a loop

	Offset				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
- 3 misses and 7 hits

TLB **hit rate** is 70%

- TLB improves performance due to **spatial locality**

- **Spatial Locality**

- If a program accesses memory at address x , it will likely soon access memory near x

- **Spatial Locality**

- If a program accesses memory at address x , it will likely soon access memory near x

- **Temporal Locality**

- Recently accessed instruction or data will likely be re-accessed soon
- Access array after loop: even better performance

TLB Miss

- Who handles a TLB miss?
 - Hardware or software (OS)?

TLB Miss

- Who handles a TLB miss?
 - Hardware or software (OS)?
- In the past: hardware
 - **Page table base register**

TLB Miss

- Who handles a TLB miss?
 - Hardware or software (OS)?
- In the past: hardware
 - **Page table base register**
- Modern: **software-managed TLB**
 - TLB miss raises an exception
 - **Trap handler** will lookup translation in the page table and update TLB

TLB Contents

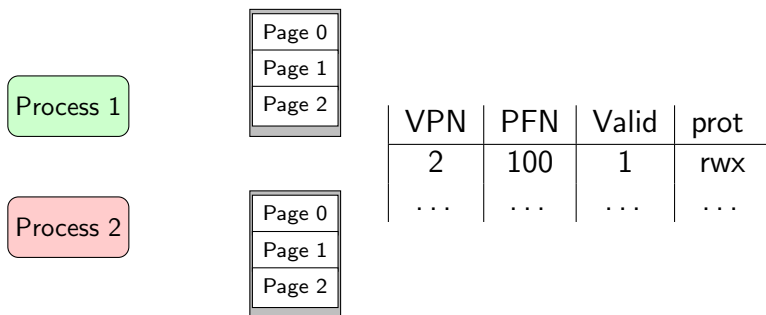
- Typical TLB: 32, 64, or 128 entries
- **Fully associative**
 - Any translation can be anywhere in the TLB
 - Hardware searches entire TLB in parallel

VPN | PFN | other bits

- Other bits:
 - **Valid** bit
 - **Protection** bits
 - **Address-space identifier**
 - **Dirty** bit
 - ...

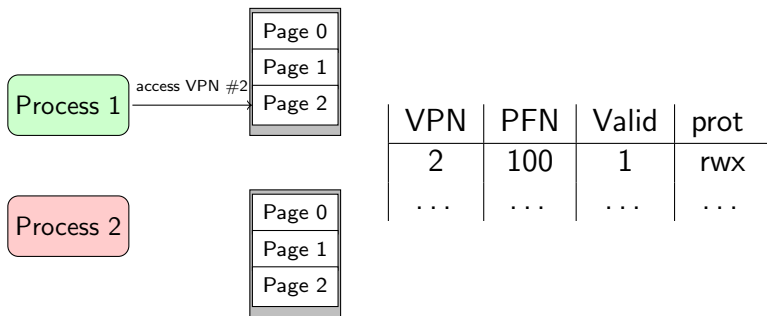
Context Switches

- On context switch, TLB contents invalid



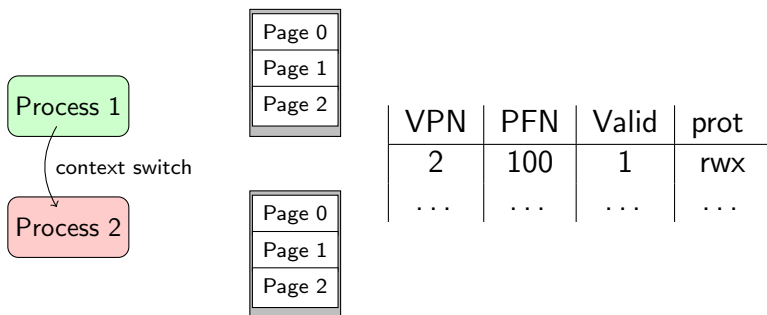
Context Switches

- On context switch, TLB contents invalid



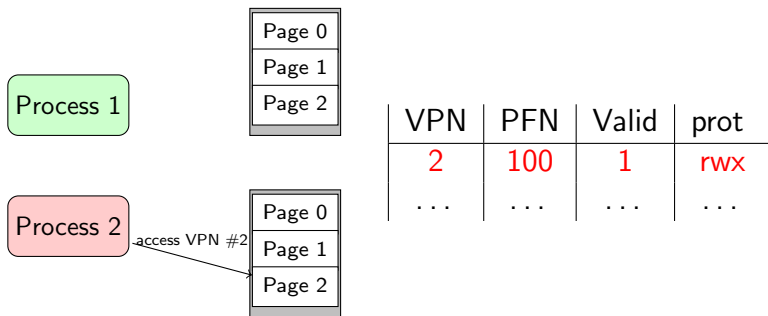
Context Switches

- On context switch, TLB contents invalid



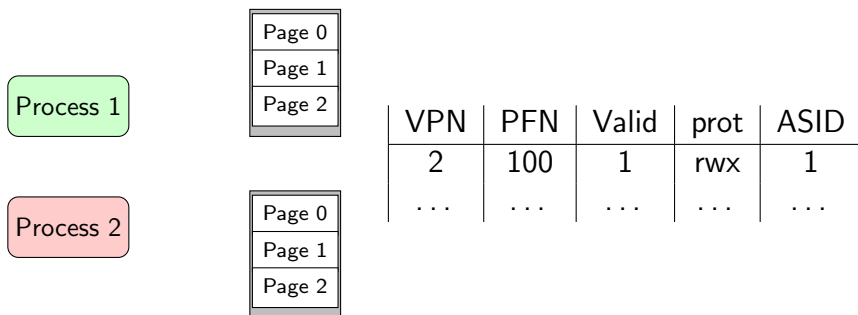
Context Switches

- On context switch, TLB contents invalid



Context Switches

- On context switch, TLB contents invalid



Share A Page

- **ASID** also allows us to share pages:

VPN	PFN	Valid	prot	ASID
10	101	1	r-x	1
-	-	-	-	-
50	101	1	r-x	2
-	-	-	-	-

Replacement Policy

- When inserting a new TBL entry:
 - Have to **replace** an old one
 - Which TLB entry should be replaced?

Replacement Policy

- When inserting a new TBL entry:
 - Have to **replace** an old one
 - Which TLB entry should be replaced?
- Common approaches:
 - **Least-recently-used (LRU)**
 - **Random** policy
 - When is it better?

Replacement Policy

- When inserting a new TBL entry:
 - Have to **replace** an old one
 - Which TLB entry should be replaced?
- Common approaches:
 - **Least-recently-used (LRU)**
 - **Random** policy
 - When is it better?
 - Consider TLB of size n , loop over $n + 1$ pages
 - LRU misses on every access

Smaller Tables

- Tackle the second paging problem:
 - Page tables consume too much memory
 - 32-bit address space (2^{32}), 4KB pages (2^{12}), 4 byte address
 - Page table size is: $\frac{2^{32}}{2^{12}} \cdot 4\text{bytes} = \mathbf{4MB}$
 - For every process!

How can we make page tables smaller?

Bigger Pages

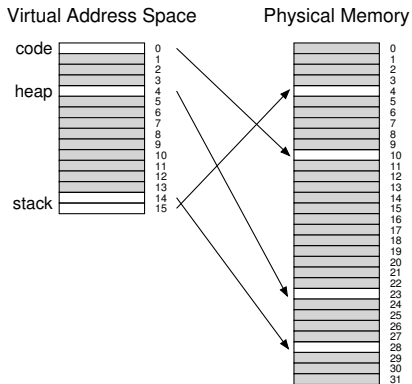
- Same address space, **16KB** pages
 - 18-bit **VPN** and 14-bit **offset**
 - $\frac{2^{32}}{2^{14}} \cdot 4\text{bytes} = \mathbf{1MB}$ per page table
- Major problem:

Bigger Pages

- Same address space, **16KB** pages
 - 18-bit **VPN** and 14-bit **offset**
 - $\frac{2^{32}}{2^{14}} \cdot 4\text{bytes} = \mathbf{1MB}$ per page table
- Major problem: **internal fragmentation**

Paging and Segments

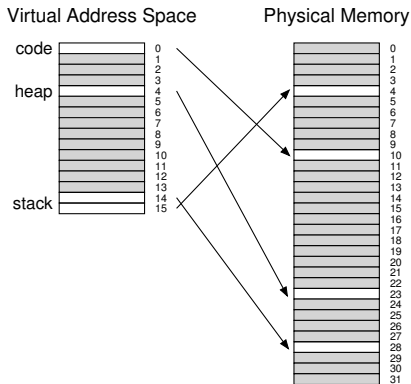
Another attempt:



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
-	0	-	-	-
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

Paging and Segments

Another attempt:

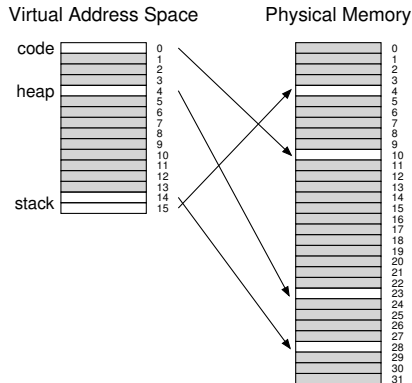


PFN	valid	prot	present	dirty
10	1	r-x	1	0
—	0	—	—	—
—	0	—	—	—
—	0	—	—	—
15	1	rw-	1	1
—	0	—	—	—
—	0	—	—	—
3	1	rw-	1	1
23	1	rw-	1	1

- Most of the page table is **unused**

Paging and Segments

- Page table for each segment
- **Base** and **bound** to physical page table
 - The bounds register indicates end of page table
 - i.e., how many valid pages it has



Paging and Segments

- Not without problems
 - Inherits segmentation issues
 - Large, sparsely-used heap can end up with page table waste
 - **External fragmentation**
 - Page tables of arbitrary size
 - Hard to find space for arbitrary size page tables

Multi-Level Page Tables

- Turn the page table into a tree
 - Split page table into page-sized units
 - Entire page of entries invalid? Don't allocate it
 - Track in a **page directory**

Multi-Level Page Tables

Linear Page Table

PTBR 201 →

	valid	prot	PFN	
	1	rx	12	PFN 201
	1	rx	13	
	0	-	-	
	1	rw	100	
	0	-	-	PFN 202
	0	-	-	
	0	-	-	
	0	-	-	
	0	-	-	PFN 203
	0	-	-	
	0	-	-	
	0	-	-	
	0	-	-	PFN 204
	0	-	-	
	1	rw	86	
	1	rw	15	

Multi-level Page Table

PDBR 200 →

	valid	PFN	
PFN 200	1	201	→
	0	-	
	0	-	
	1	204	

The Page Directory

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

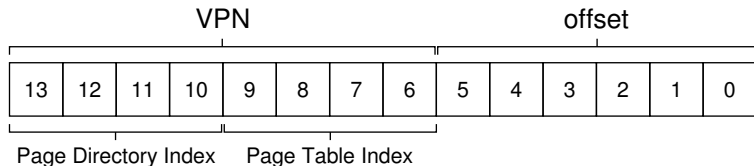
	valid	prot	PFN	
	1	rx	12	PFN 201
	1	rx	13	
	0	-	-	
	1	rw	100	

	valid	prot	PFN	
	0	-	-	PFN 204
	0	-	-	
	1	rw	86	
	1	rw	15	

Multi-Level Page Tables

- **Page directory**

- One **page directory entry (PDE)** per page of page table
- Valid bit and PFN



Multi-Level Page Tables

- Pros:
 - Allocate space in proportion to use
 - Each portion fits within a page
 - Easier to manage memory
- Cons:
 - Add a **level of indirection (time-space trade-off)**
 - Increased complexity

More Than Two Levels

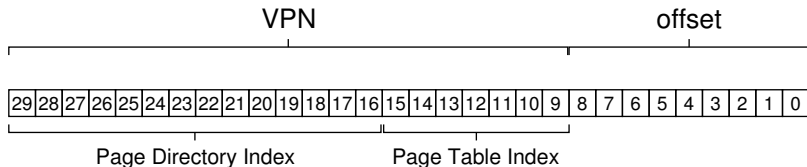
- In some cases, a deeper tree is possible (and needed)
- Let's assume (for example):
 - 30-bit virtual address space, 512 bytes page

More Than Two Levels

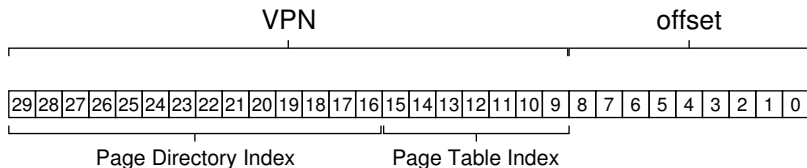
- In some cases, a deeper tree is possible (and needed)
- Let's assume (for example):
 - 30-bit virtual address space, 512 bytes page
 - 21-bit VPN and 9-bit offset
 - How many entries in a page?

More Than Two Levels

- In some cases, a deeper tree is possible (and needed)
- Let's assume (for example):
 - 30-bit virtual address space, 512 bytes page
 - 21-bit VPN and 9-bit offset
 - How many entries in a page?
 - 512 bytes, PTE of 4 bytes: 128 PTEs on a single page

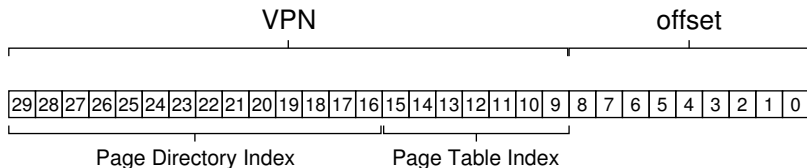


More Than Two Levels

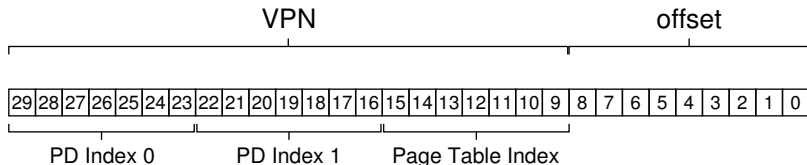


- 14 bits left for page directory
 - 2^{14} entries, spans 128 pages

More Than Two Levels



- 14 bits left for page directory
 - 2^{14} entries, spans 128 pages
 - Can add a directory to the directory!



Inverted Page Tables

- Keep a single page table
 - Entry for each physical page
 - Keeps which process is using the page, virtual page it maps to
 - Use hash table to speed up lookups

Swapping

- Relax assumption that physical memory suffices
- Page tables may be too big to fit into memory
- Use **kernel virtual memory**
 - Virtual memory allows us to **swap** pages to disk
 - Our next topic

Summary

- Fixed size pages, mapped to physical page frames
- Translation with **TLB**
 - **TLB hit**: fast
 - **TLB miss**: slow, update **TLB** and retry instruction
- **Multi-level page tables**
 - Divide page table into pages
 - The **page directory**