Paging

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Paging

Improving on segmentation

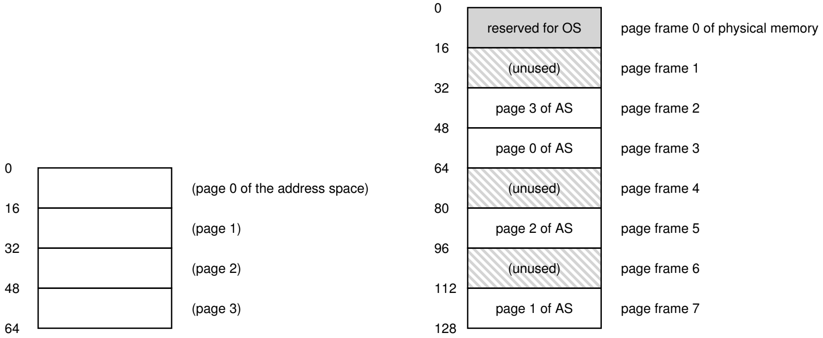
- Transparent to user mode code.
- No need for compactification. Ever.
- Basic idea: Small fixed size parts of process spread ‘randomly’ in physical memory.

Paging

- Divide process address space into fixed-size pieces
- Each fixed-size unit is a (virtual) **page**
- Divide physical memory into **page frames** (physical page)

Paging

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



Paging

- **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:

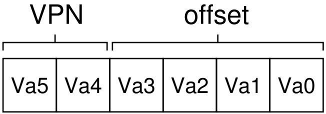
Va5	Va4	Va3	Va2	Va1	Va0

Paging

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

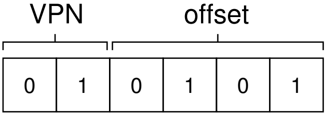
Va5	Va4	Va3	Va2	Va1	Va0

  - Address space is 4 pages, thus:



Paging

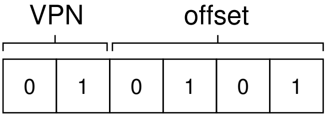
- Let's translate an address!
  - Virtual address 21:



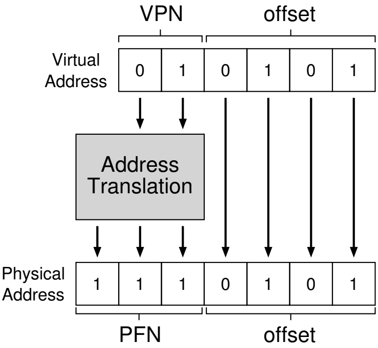
VPN		offset			
0	1	0	1	0	1

Paging

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



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



Bitology

Page Size	Low Bits		
16 bytes			

Bitology

Page Size	Low Bits		
16 bytes	4		
1KB			

Bitology

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

Bitology

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

Bitology

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

Bitology

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

Bitology

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		

Bitology

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		

Bitology

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		

Bitology

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		

Bitology

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		

Bitology

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	

Bitology

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	

Bitology

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	

Bitology

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	

Bitology

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	

Bitology

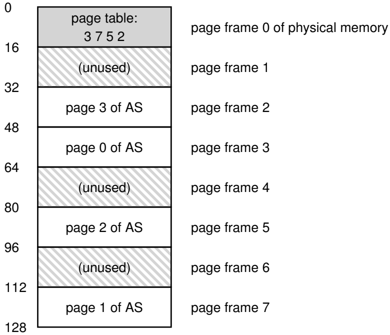
Page Size	Low Bits	V.Addr Bits	High Bits	# V.Pages
16 bytes	4	10	6	64
1KB	10	20	10	1K
1MB	20	32	12	4K
512 bytes	9	16	5	32
4KB	12	32	20	

Bitology

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)**
  - 4GB programs yields 4MB of memory for each page table
  - Promil overhead. However: Huge continuity demand!
  - Danger of compactification creeping back in.
  - 100 processes: 400MB just for address translations!



0
page table:
3 7 5 2
page frame 0 of physical memory
16
(unused)
page frame 1
32
page 3 of AS
page frame 2
48
page 0 of AS
page frame 3
64
(unused)
page frame 4
80
page 2 of AS
page frame 5
96
(unused)
page frame 6
112
page 1 of AS
page frame 7
128

What's Actually In The Page Table?

- Simplest form: **linear page table**
  - Array, indexed by VPN, looks up PTE to find PFN
  - Register of array length!
  - **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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																							G	PAT	D	A	PCD	PWT	U/S	R/W	P

Issues

- HUGE CONTINUITY DEMAND.
- Too slow
  - To get 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

Issues

- HUGE CONTINUITY DEMAND.
- Too slow
  - To get 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

Solving the continuity demand

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

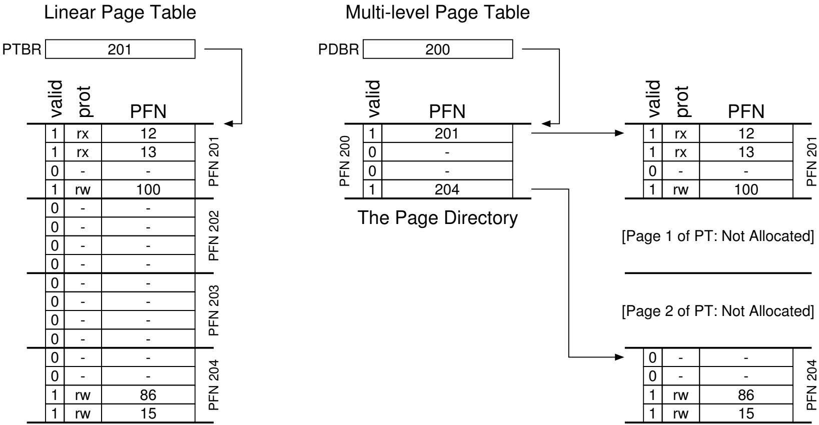
- Same address space, **16KB** pages
  - 18-bit **VPN** and 14-bit **offset**
  - $\frac{2^{32}}{2^{14}} \cdot 4\text{bytes} = \mathbf{1MB}$ per page table

- Major problem: **internal fragmentation**

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

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

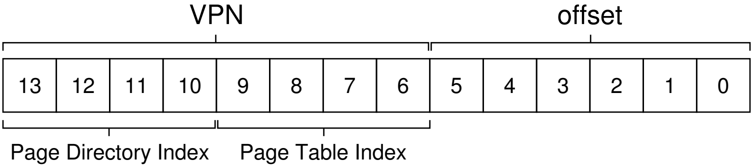
[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

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



VPN

offset

13 12 11 10 9 8 7 6 5 4 3 2 1 0

Page Directory Index

Page Table Index

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

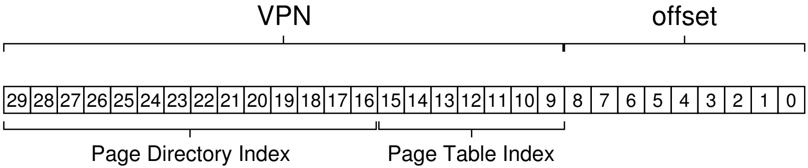
- In some cases, a deeper tree is possible (and needed)
- Let's assume (for example):
  - 30-bit virtual address space, 512 bytes page

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?

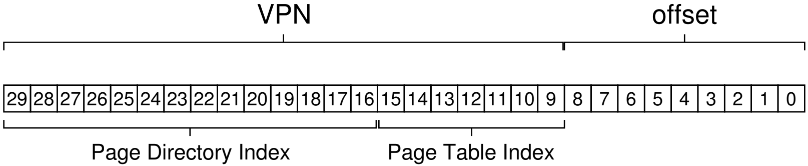
More Than Two Levels

- 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



29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

More Than Two Levels



VPN

offset

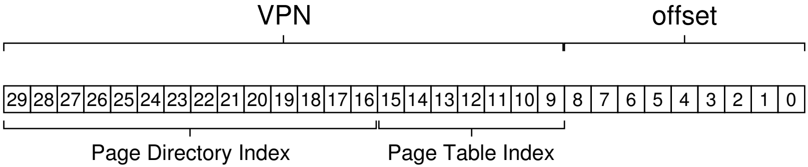
29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Page Directory Index

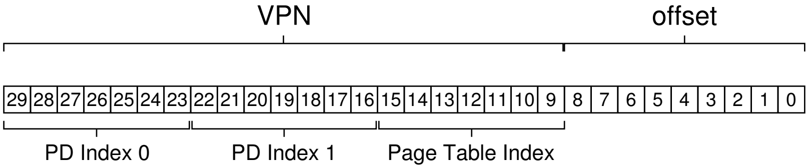
Page Table Index

- 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!



Translation Lookaside Buffer (TLB)

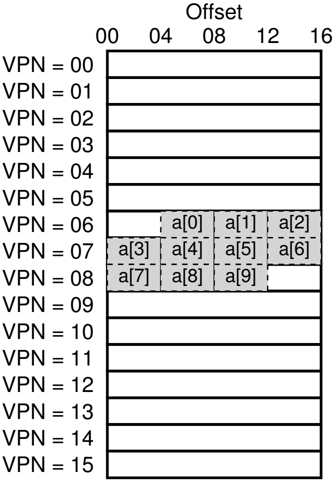
- Mitigating the extra memory accesses needed for translation.
- A special purpose cache in the MMU.
- They key is the virtual address.
- The value is the physical address.
- Like all caches:
  - Associative or set associative
  - Replacement policy when full.
- Always remember it is just a chache. No functionality change.

TLB

- **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

TLB

- Example: accessing a 10 elements array in a loop



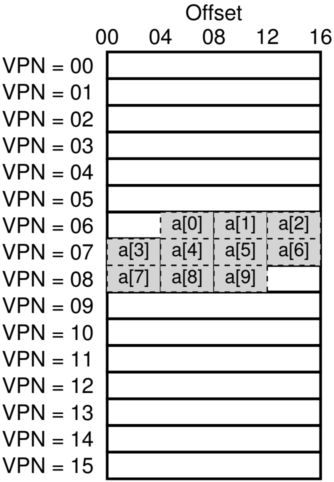
Offset

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				

TLB

- Example: accessing a 10 elements array in a loop

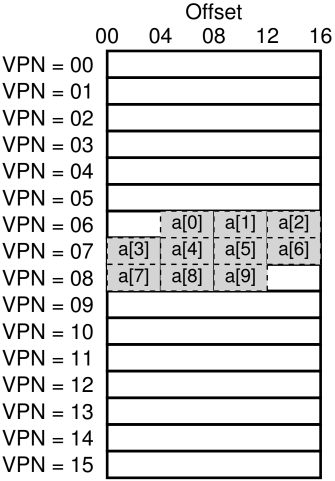


	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 02					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

- **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

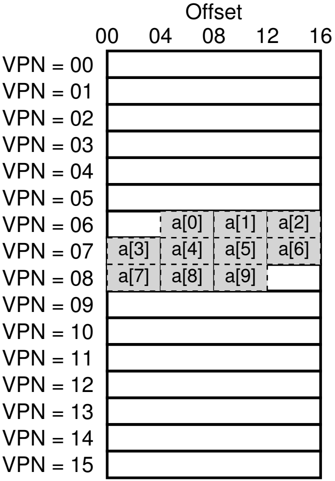


- **miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
- 3 misses and 7 hits

TLB **hit rate** is 70%

TLB

- Example: accessing a 10 elements array in a loop



- **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**

Locality

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

Locality

- **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)?
- Usually hardware (x86, riscv). Can be software (some mipses)

TLB Miss

- Who handles a TLB miss?
  - Hardware or software (OS)?
- Usually hardware (x86, riscv). Can be software (some mipses)
- If **software-managed TLB**
  - TLB miss raises an exception
  - **Trap handler** will lookup translation in the page table and update TLB

TLB Contents

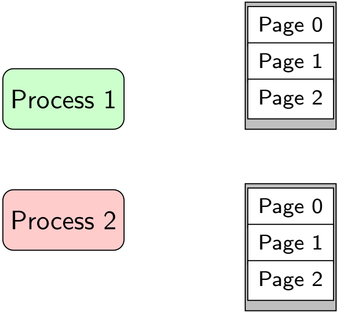
- Typical TLB: Nowadays can be several throusands entries
- Full associative, set associative, or direct are possible.

VPN	PFN	other bits

- Usually a copy of the PTE.

Context Switches

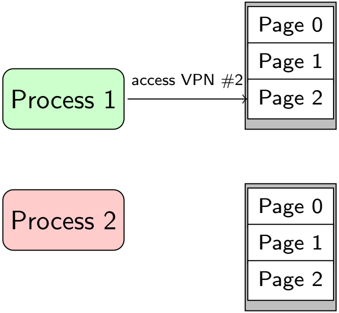
- On context switch, TLB contents usually are flushed



VPN	PFN	Valid	prot
2	100	1	rwx
...	...	...	...

Context Switches

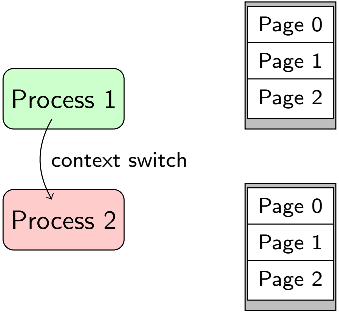
- On context switch, TLB contents usually are flushed



VPN	PFN	Valid	prot
2	100	1	rwx
...	...	...	...

Context Switches

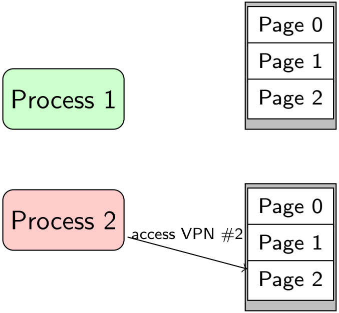
- On context switch, TLB contents usually are flushed



VPN	PFN	Valid	prot
2	100	1	rwx
...	...	...	...

Context Switches

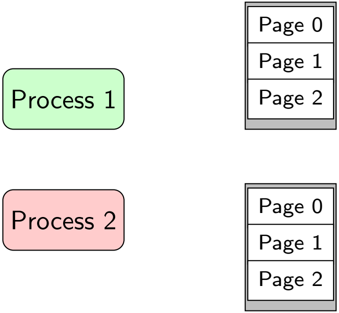
- On context switch, TLB contents usually are flushed



VPN	PFN	Valid	prot
2	100	1	rwx
...	...	...	...

Context Switches

- On context switch, TLB contents usually are flushed



VPN	PFN	Valid	prot	ASID
2	100	1	rwx	1
...	...	...	...	...

Mitigating TLB flush

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

Replacement Policy

(This is the same as every other cache!)

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

Replacement Policy

(This is the same as every other cache!)

- When inserting a new TLB 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

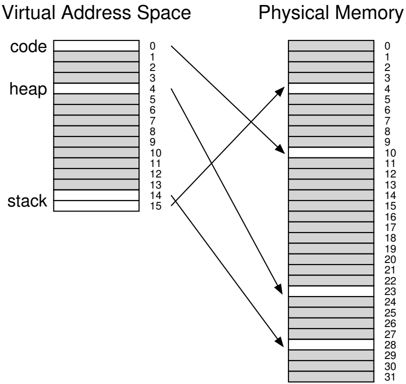
(This is the same as every other cache!)

- When inserting a new TLB 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

Abomination

Paging and Segments

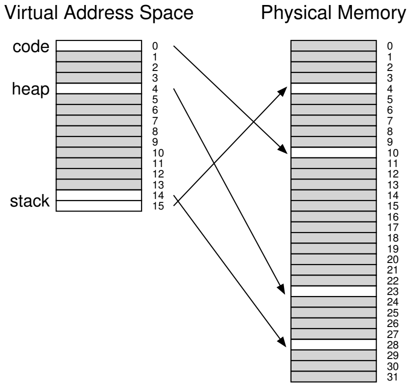
Another attempt:



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:

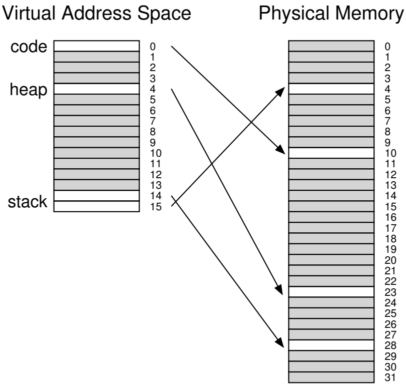


PFN	valid	prot	present	dirty
10	1	r-x	1	0
–	0	–	–	–
–	0	–	–	–
–	0	–	–	–
15	1	rw-	1	1
–	0	–	–	–
–	0	–	–	–
3	1	rw-	1	1
23	1	rw-	1	1

- 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

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**