TLB - Faster translation (ch. 19)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Translation Lookaside Buffer (TLB)

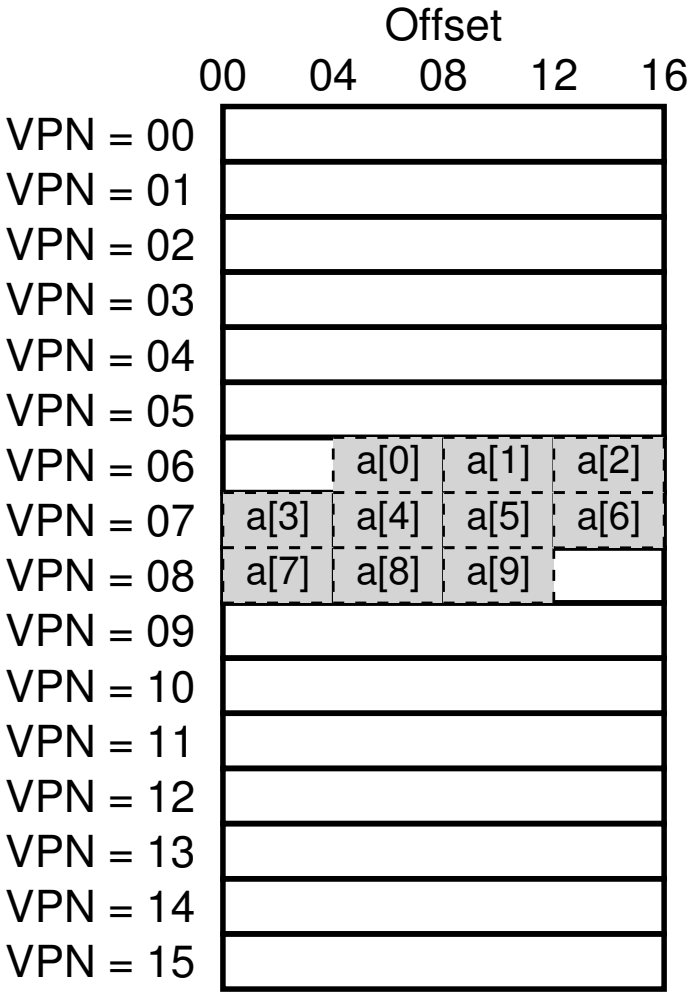
- Mitigating the extra memory accesses needed for translation.
- A special purpose cache in the MMU.
- They key is the virtual address.
- The value is the pte.
- Like all caches:
  - Associative or set associative
  - Replacement policy when full.
- Always remember it is just a chache. No functionality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

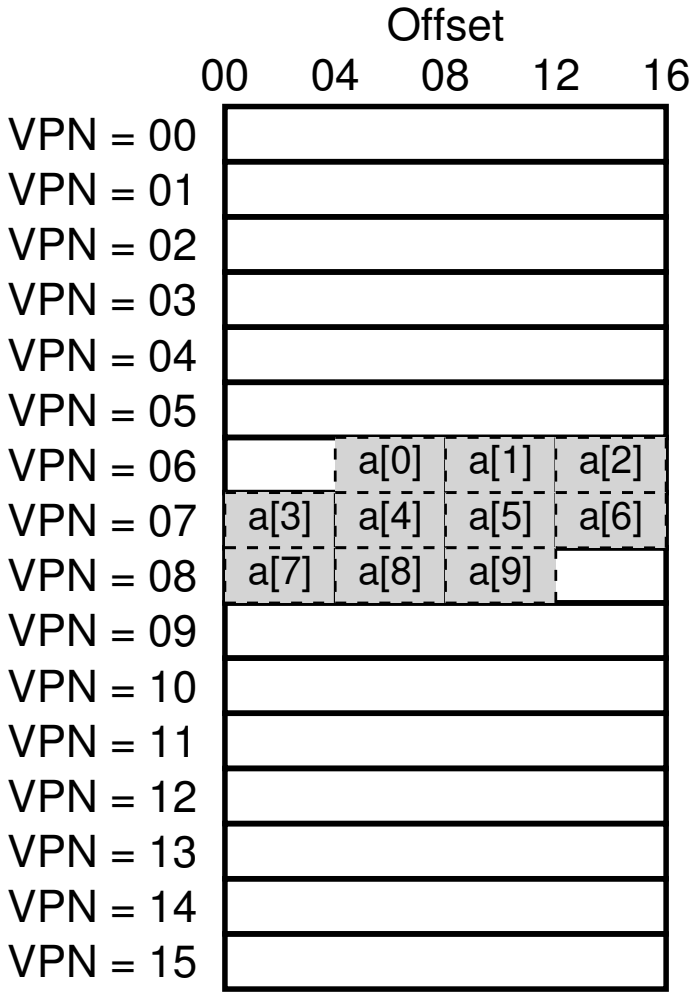
TLB

- Example: accessing a 10 elements array in a loop



TLB

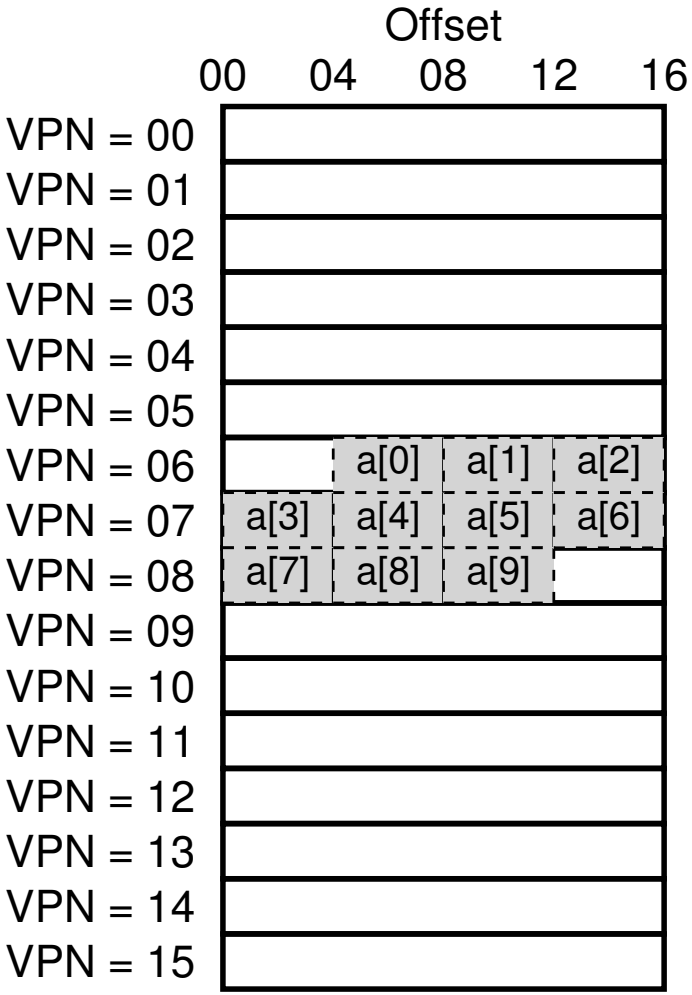
- Example: accessing a 10 elements array in a loop



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

TLB

- Example: accessing a 10 elements array in a loop

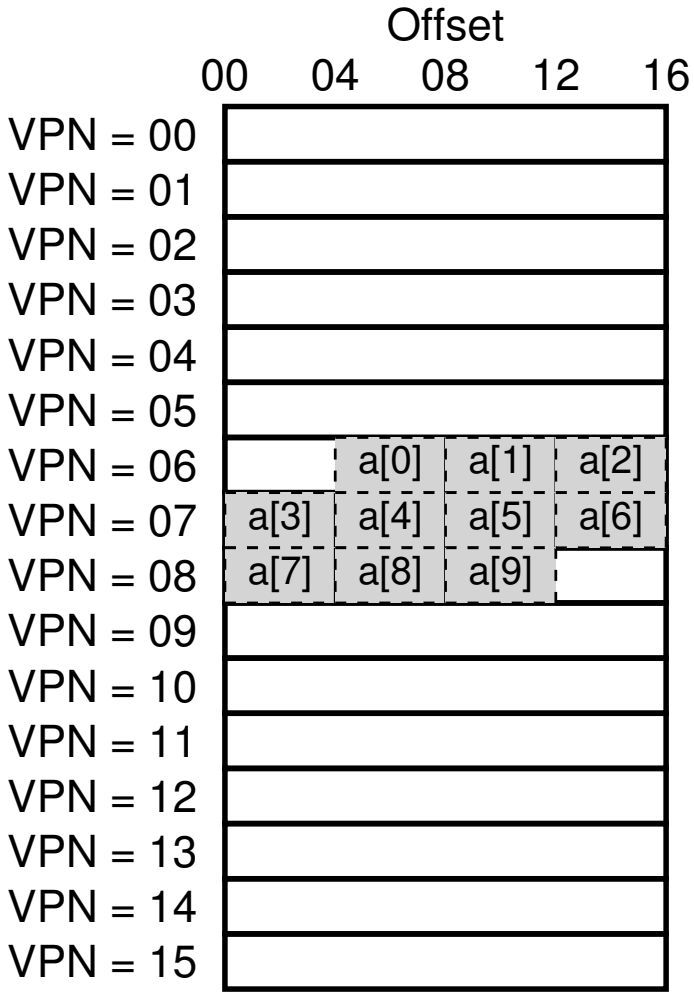


- **miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
- 3 misses and 7 hits

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

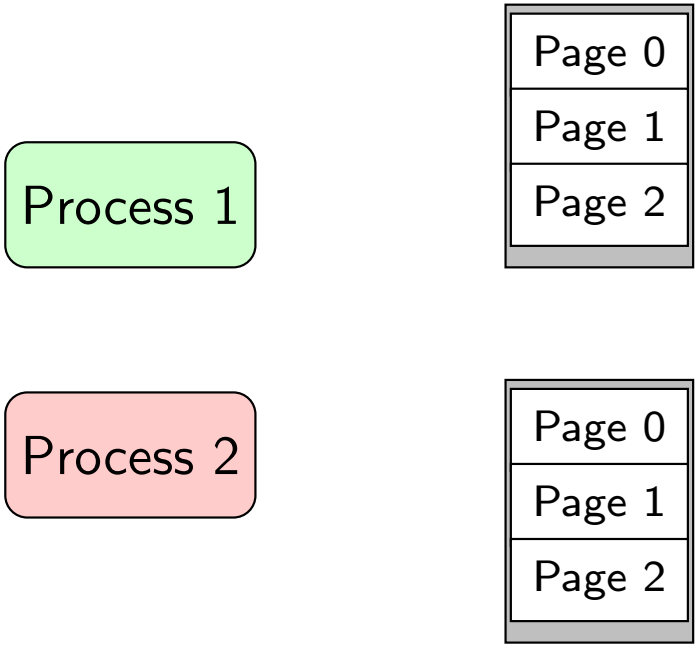
- Typical TLB: Nowadays can be several throusands entries
- Full associative, set associative, or direct are possible.

$$\text{VPN} \mid \text{PFN} \mid \text{other bits}$$

- Usually a copy of the PTE.

Context Switches

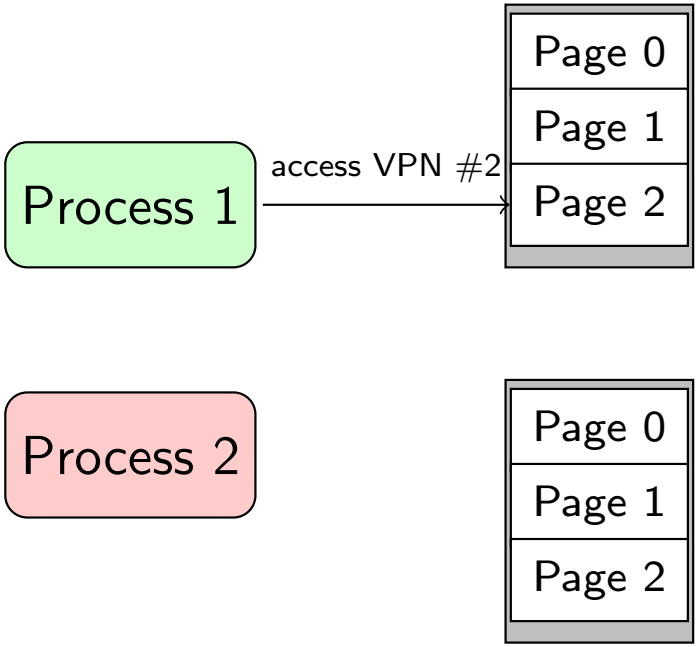
- On context switch, TLB contents usually are flushed



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

Context Switches

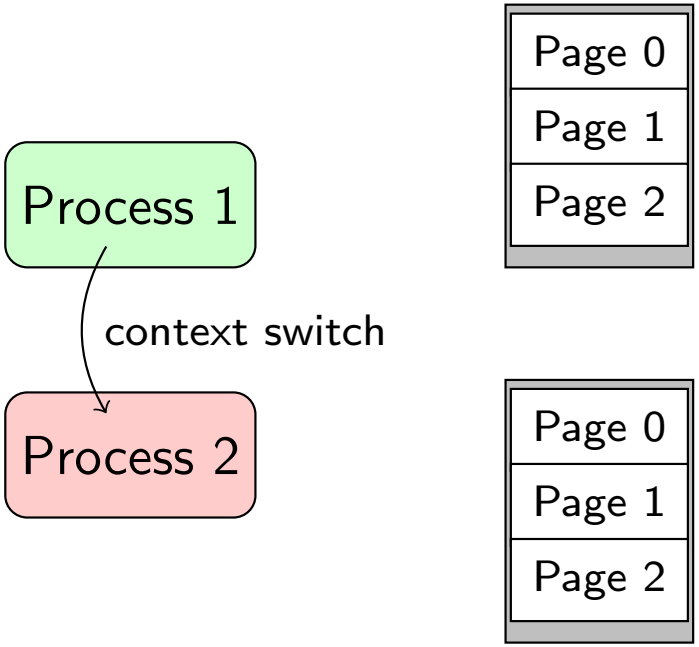
- On context switch, TLB contents usually are flushed



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

Context Switches

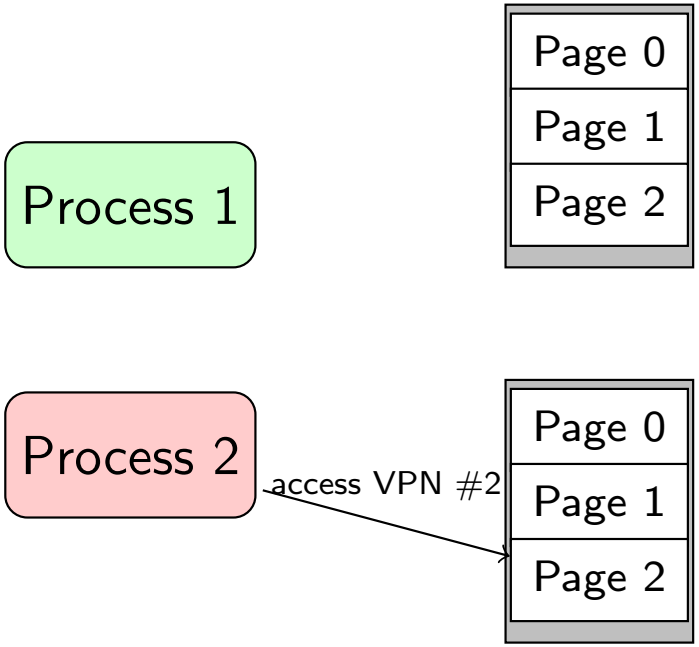
- On context switch, TLB contents usually are flushed



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

Context Switches

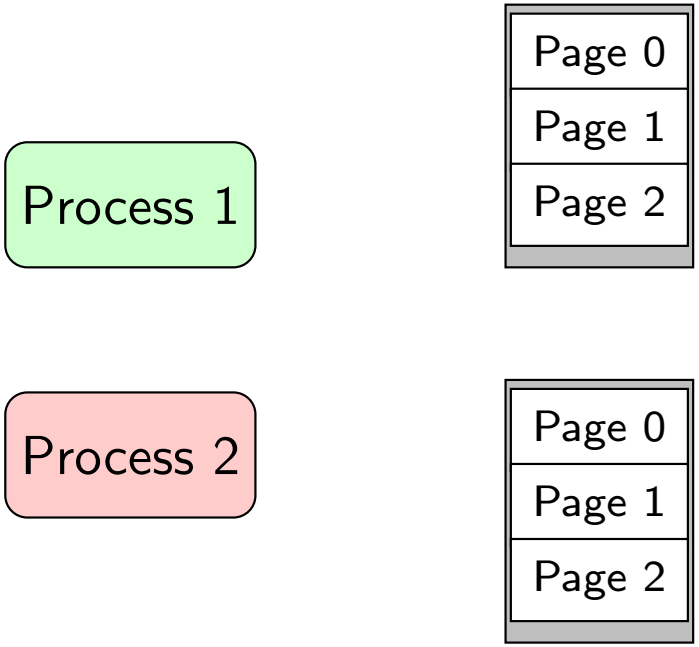
- On context switch, TLB contents usually are flushed



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

Unless there are ASIDs.

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

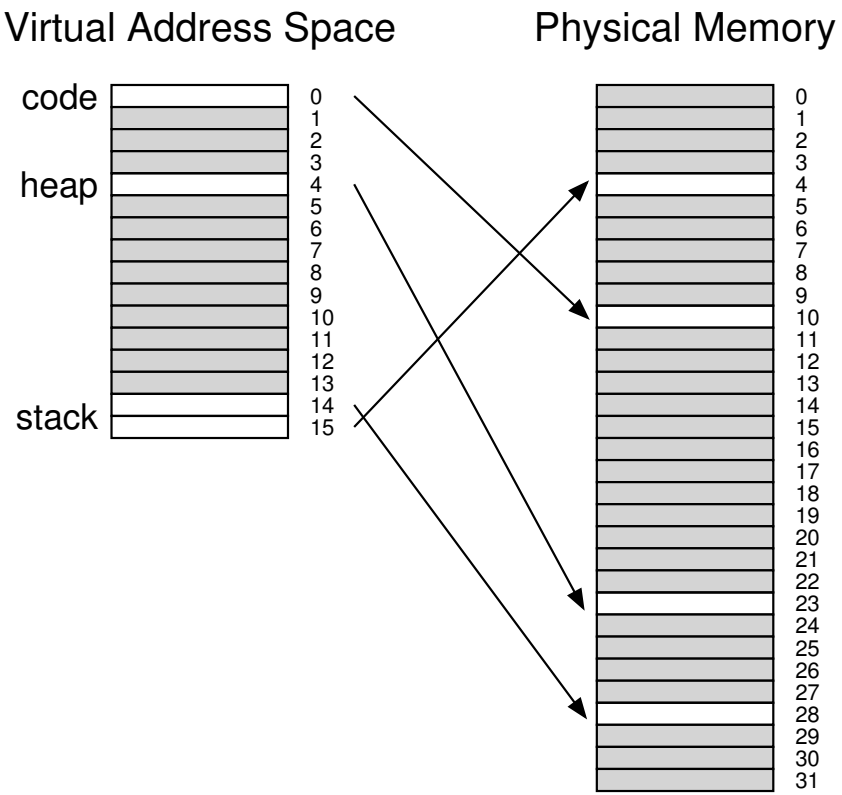
(This is the same as every other cache!)

- 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

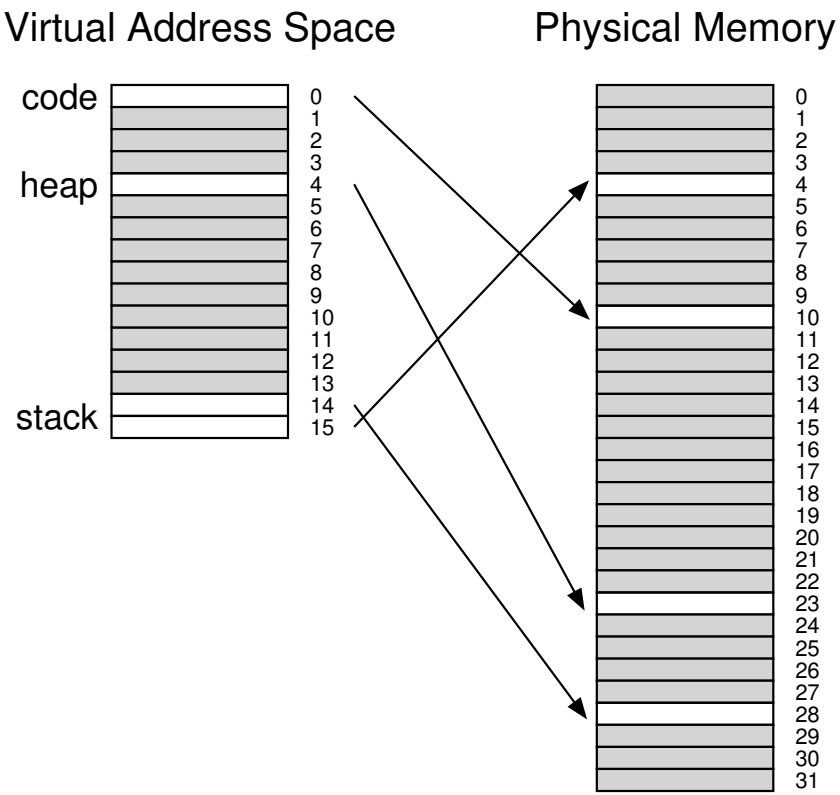
Another attempt:



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:

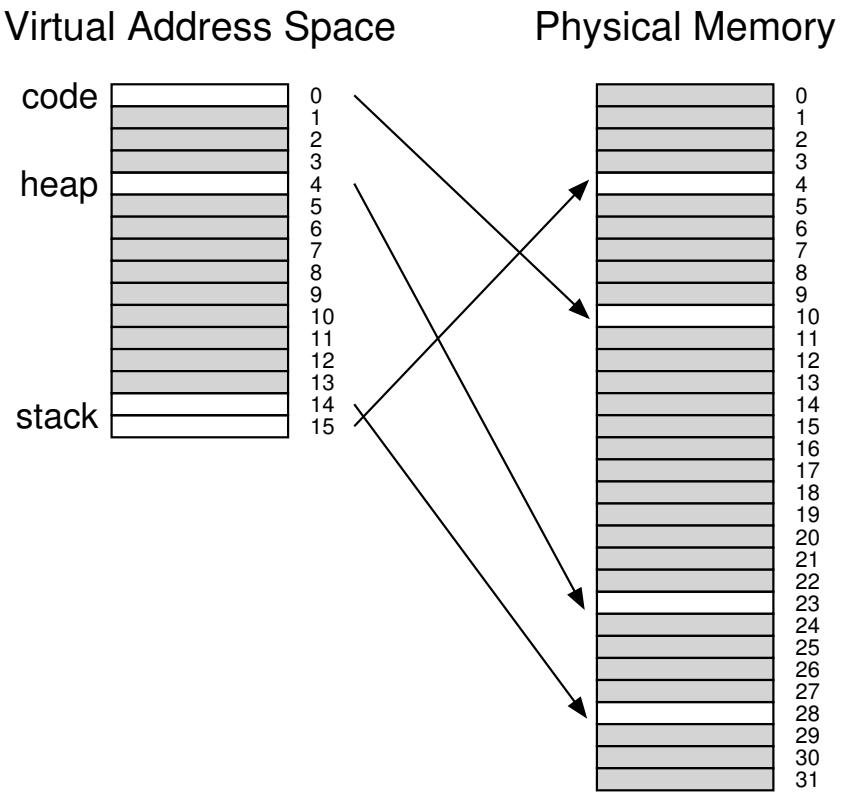


PFN	valid	prot	present	dirty
10	1	r-x	1	0
–	0	–	–	–
–	0	–	–	–
–	0	–	–	–
15	1	rw-	1	1
–	0	–	–	–
–	0	–	–	–
3	1	rw-	1	1
23	1	rw-	1	1

- 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

Interesting idea. Unreasonable limitation.

- 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**