Operating Systems

Autumn 2024

Virtual Memory and Paging

Virtual memory

- Splits memory between various processes
- But gives each process the illusion that it has the full address space
  - Code uses virtual memory addresses (aka logical addresses)
  - CPU somehow translates to physical addresses assigned to that process
- Virtual memory also gives the illusion that memory is huge
  - OS swaps memory to disk if there is no space in physical memory (More on this in later lectures.)

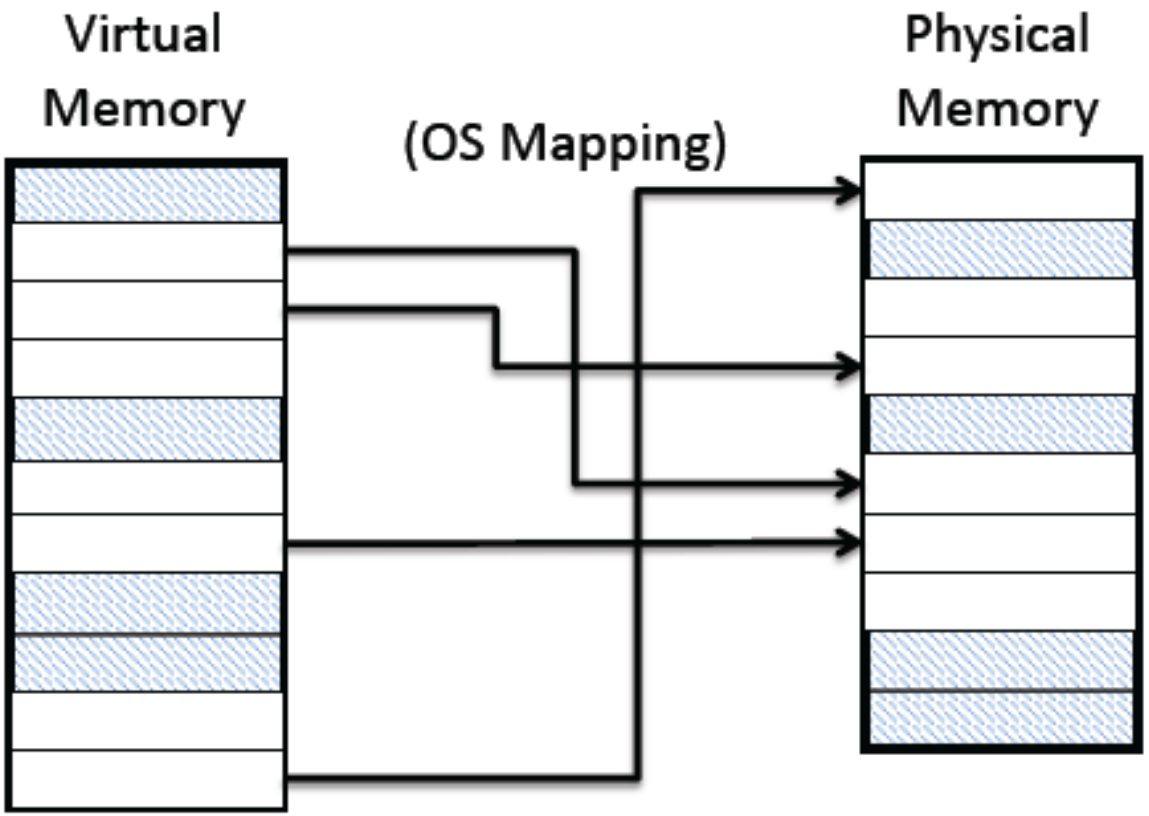
Paging vocabulary

- For each process, the virtual address space is divided into fixed-size pages (virtual pages)
- For the system, the physical memory is divided into fixed-size frames (physical pages)
- The size of a page is equal to that of a frame
  - Often 4 KB in practice

Main idea

- ANY virtual page can be stored in any available frame
  - Makes finding an appropriately-sized memory gap very easy – they're all the same size
- For each process, OS keeps a table mapping each virtual page to physical frame

Main idea



Addressing

- Virtual addresses
  - High-order bits: page ♯
  - Low-order bits: offset within the page
- Physical addresses
  - High-order bits: frame ♯
  - Low-order bits: offset within the frame

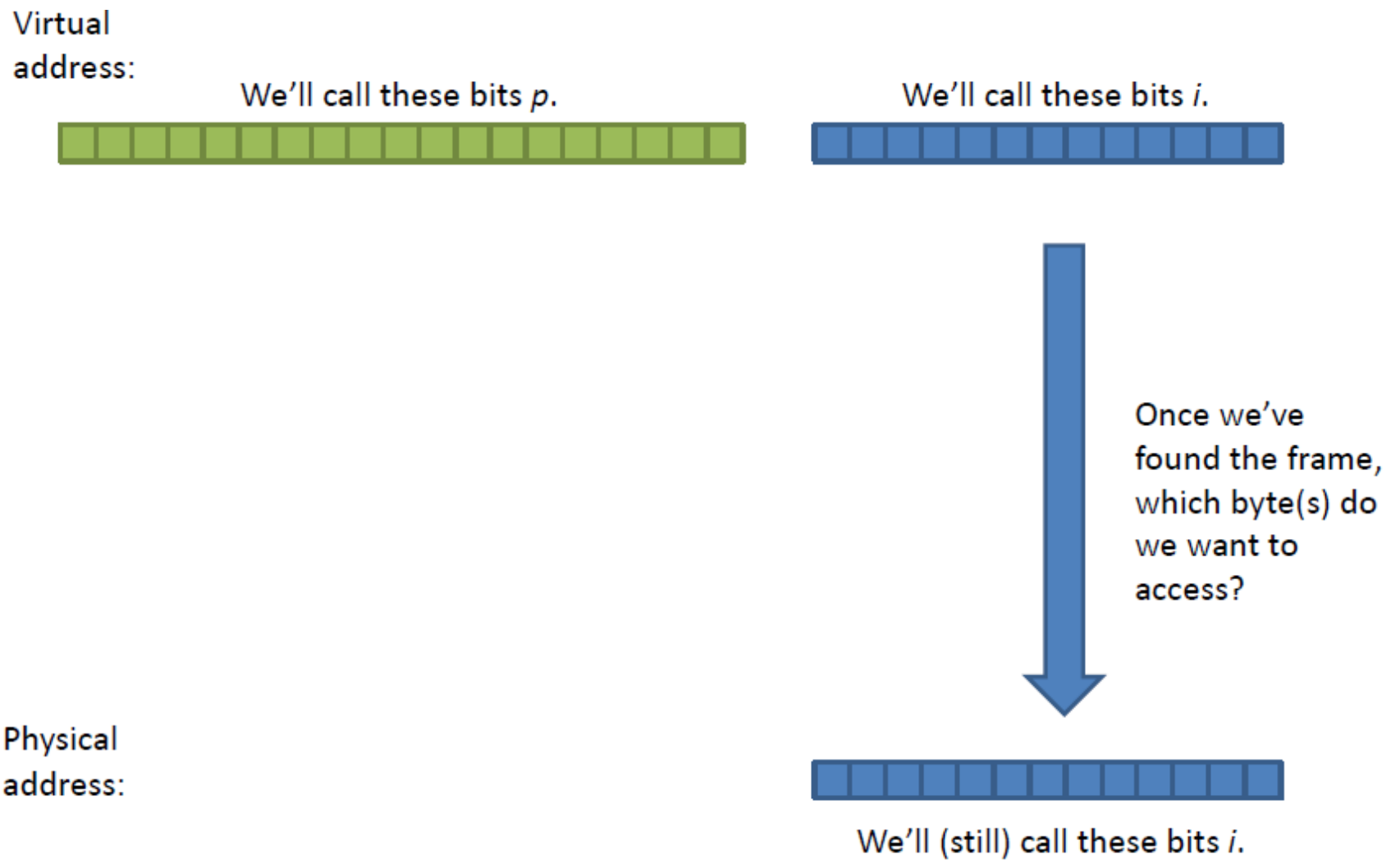
Example: 32-bit virtual addresses

- Suppose we have 8-KB (8192-byte) pages
- We need enough bits to individually address each byte in the page
  - How many bits do we need to address 8192 items?
  - $2^{13} = 8192$, so we need 13 bits
  - Lowest 13 bits: offset within page

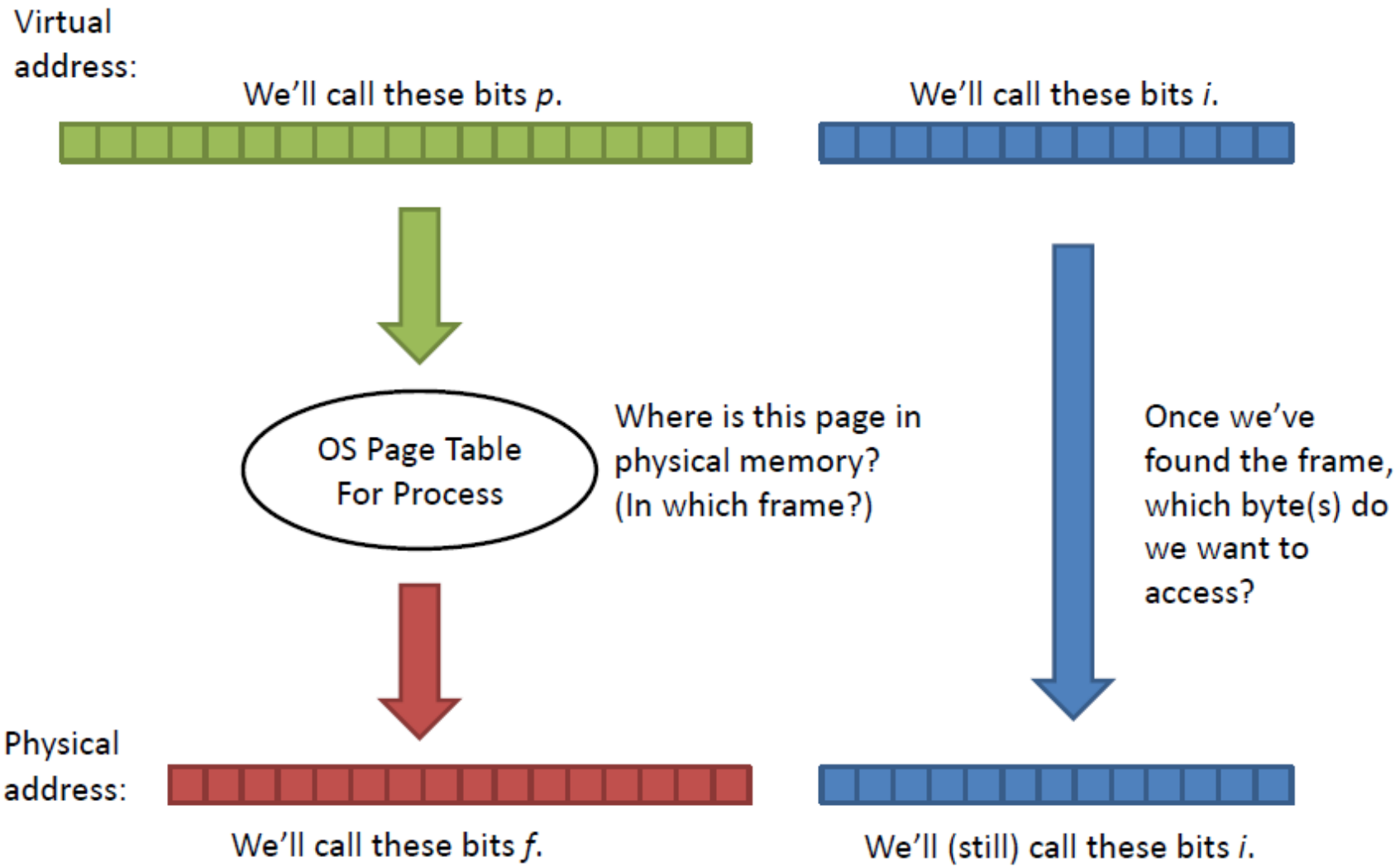
Example: 32-bit virtual addresses

- Suppose we have 8-KB (8192-byte) pages
- We need enough bits to individually address each byte in the page
  - How many bits do we need to address 8192 items?
  - $2^{13} = 8192$, so we need 13 bits
  - Lowest 13 bits: offset within page
- Remaining 19 bits: page number

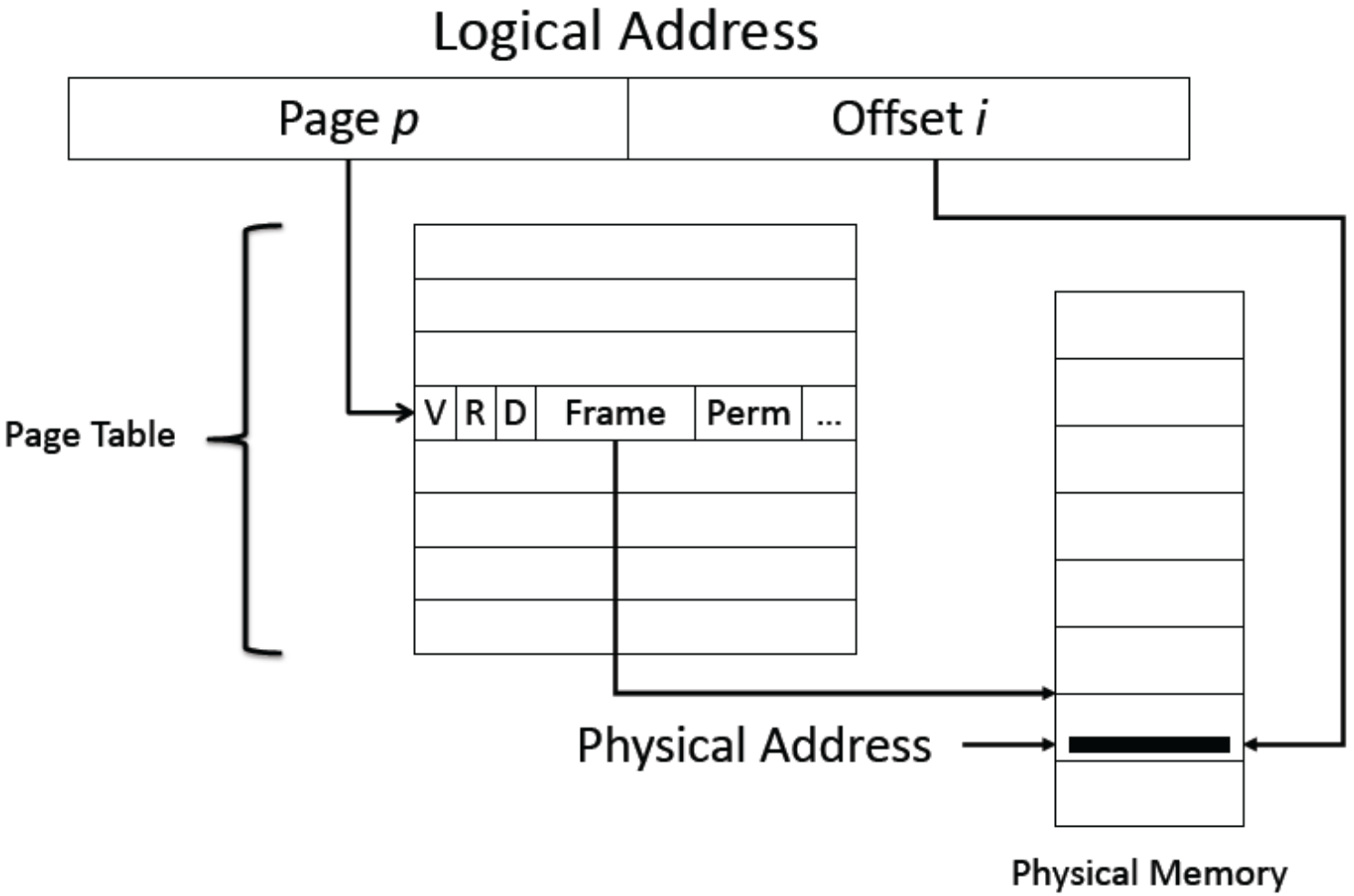
Address translation



Address translation



Address translation



Where are the page tables stored?

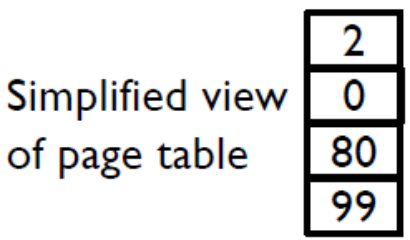
- For now simply assume it's all stored in OS memory
  - a special Page Table Base Register (PTBR) stores the physical address of the page table
  - OS kernel changes the PTBR value when switching processes

Memory access with paging

14-bit addresses

```
0x0010:  movl 0x1100, %edi
```

Assume PT is at phy addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12



Fetch instruction at logical addr `0x0010`

Access page table to get ppn for vpn 0 (Mem ref 1)

Learn vpn 0 is at ppn ____

Fetch instruction at _____ (Mem ref 2)

Memory access with paging

14-bit addresses

```
0x0010:  movl 0x1100, %edi
```

Assume PT is at phy addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view of page table

2
0
80
99

Fetch instruction at logical addr `0x0010`

Access page table to get ppn for vpn 0
Mem ref 1: `0x5000`

Learn vpn 0 is at ppn 2

Fetch instruction at `0x2010` (Mem ref 2)

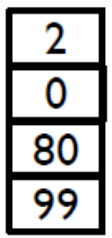
Memory access with paging

14-bit addresses

```
0x0010:  movl 0x1100, %edi
```

Assume PT is at phy addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view of page table



Exec. load from logical addr `0x1100`

Access page table to get ppn for vpn 1 (Mem ref 3)

Learn vpn 1 is at ppn ____

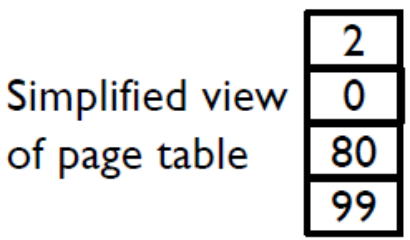
Fetch instruction at _____ (Mem ref 4)

Memory access with paging

14-bit addresses

```
0x0010:  movl 0x1100, %edi
```

Assume PT is at phy addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12



Exec. load from logical addr `0x1100`

Access page table to get ppn for vpn 1
Mem ref 3: `0x5004`

Learn vpn 1 is at ppn 0

movl from `0x0100` (Mem ref 4)

Summary: page translation steps

Hardware (MMU): for each memory reference

1. extract VPN (virt page num) from VA (virt addr)
2. calculate addr of PTE (page table entry)
3. read PTE from memory
4. extract PFN (page frame num)
5. build PA (phys addr)
6. read contents of PA from memory into register

Which steps are expensive?

Example: Array iterator

```
int sum=0;
for (i=0; i < N; i++) {
    sum += a[i];
}
```

Assume a starts at 0x3000
Ignore instruction fetches and access to i

Virtual address	Physical address
load 0x3000 // a[0]	load 0x100C
	load 0x7000
load 0x3004 // a[1]	load 0x100C
	load 0x7004
load 0x3008 // a[2]	load 0x100C
	load 0x7008
load 0x300C // a[3]	load 0x100C
	load 0x700C

Example: Array iterator

Virtual address	Physical address
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
	load 0x700C

Access page table to get ppn for vpn 3

Mem ref 1: `0x100C`

Learn vpn 3 is at ppn 7

Example: Array iterator

Mem ref 2: access `a[i]`

Virtual address	**Physical address**
load 0x3000	load 0x100C
	load 0x7000 // a[0]
load 0x3004	load 0x100C
	load 0x7004 // a[1]
load 0x3008	load 0x100C
	load 0x7008 // a[2]
load 0x300C	load 0x100C
	load 0x700C // a[3]

Paging latency

- Every memory access now requires an additional read to get the physical page number from the page table
- RAM access is slow ($\sim$50ns), so this is very bad!

Translation lookaside buffer (TLB)

- TLB is the solution to our paging latency problems
- TLB caches recently used PTEs
  - In other words, it's a small fraction of the current page table that is stored on-chip, in fast memory
  - Usually "fully associative"

Why does TLB help?

- Because programs don't access random addresses
- We're likely to need the same translations in the future
- Temporal locality – programs reuse the exact same memory addresses
- Spatial locality – programs typically will access memory near recently-used memory

Cache dynamics

- A cache hit is when data is found in the cache
  - This is the fast case, and hopefully the most common
  - A cache miss is when data is not found in the cache
- Our attempt to take a shortcut failed
  - Must carry out the request normally (access page table in RAM)
  - When done, store the data in the cache for next time
  - To make space in the cache, we must chose an existing entry to evict (remove)

Example: Array iterator

```
int sum=0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Assume a starts at 0x1000
Ignore instrution fetches and access to i

Assume following virtual address stream:

load 0x1000

load 0x1004

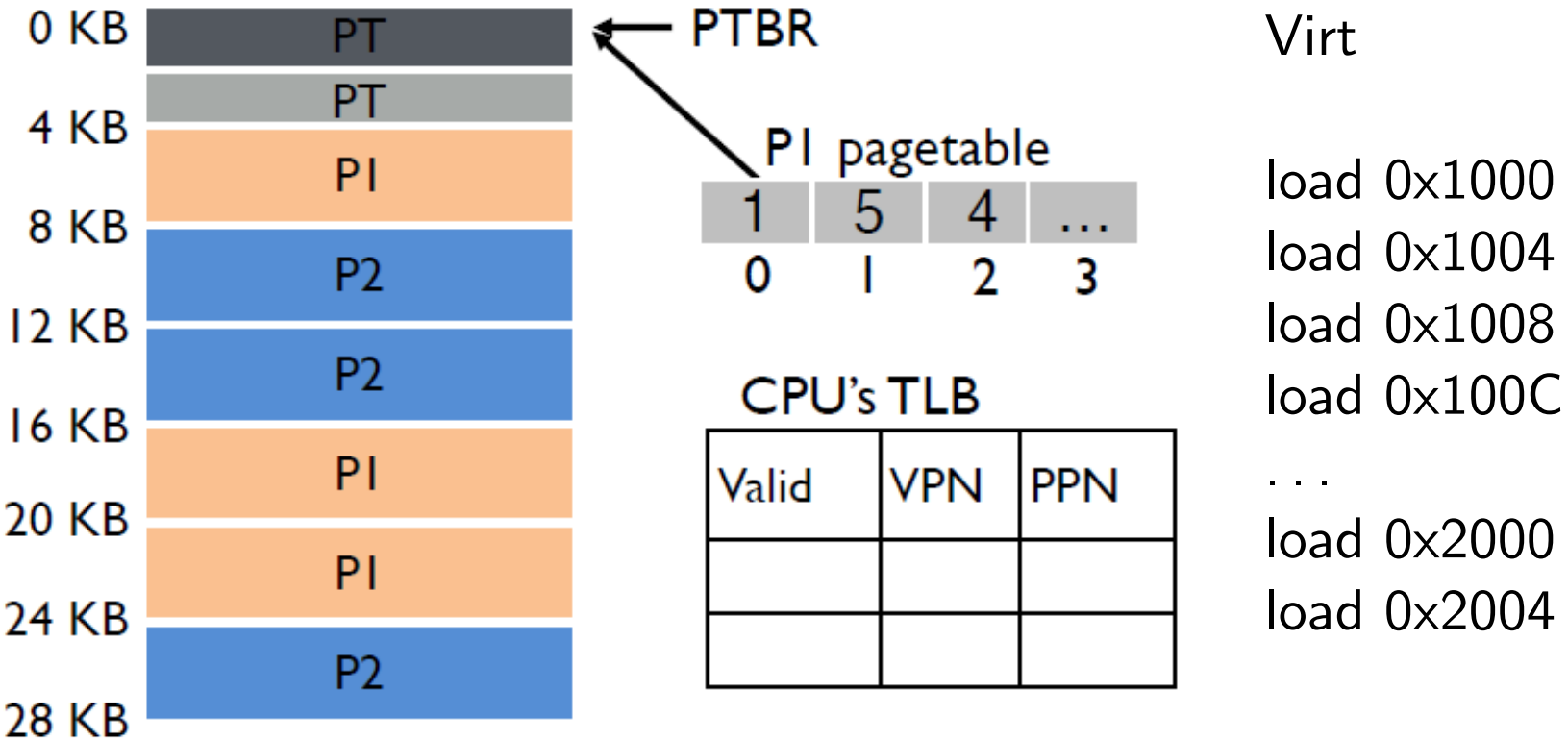
load 0x1008

load 0x100C

...

What will TLB behavior look like?

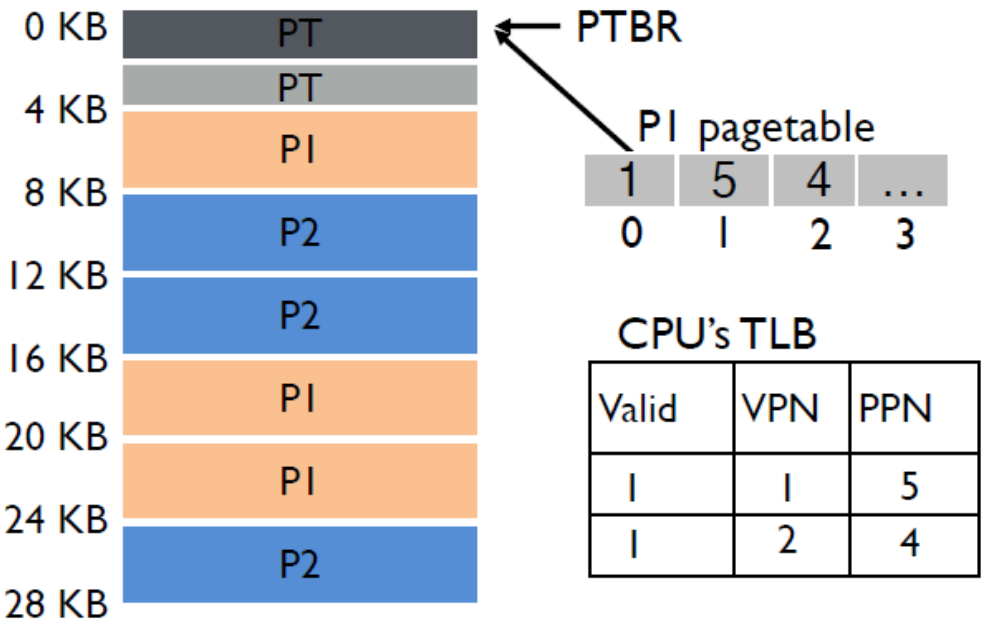
TLB Accesses: Sequential example



Virt

load 0x1000
load 0x1004
load 0x1008
load 0x100C
...
load 0x2000
load 0x2004

TLB Accesses: Sequential example



Virt	Phys
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB hit)
	load 0x5004
load 0x1008	(TLB hit)
	load 0x5008
load 0x100C	(TLB hit)
	load 0x500C
...	
load 0x2000	load 0x0008
	load 0x4000
load 0x2004	(TLB hit)
	load 0x4004

Performance of TLB

```
int sum=0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Assume 4KB pages

Assume int's are of 4 bytes

Miss rate of TLB
= ♯ TLB misses / ♯ TLB lookups

♯ TLB lookups?
= number of accesses to a

♯ TLB misses?
= number of unique pages accessed

Miss rate?

Hit rate?

= (1 - miss rate)

Performance of TLB

```
int sum=0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Assume 4KB pages

Assume int's are of 4 bytes

Miss rate of TLB
= ♯ TLB misses / ♯ TLB lookups

♯ TLB lookups
=2048

♯ TLB misses
= 2

Miss rate
= 2/2048 = 0.1%

Hit rate
= 99.9%

Summary: page translation steps (with TLB)

Hardware (MMU): for each memory reference

1. extract VPN (virt page num) from VA (virt addr)
2. check TLB for VPN
   if miss:
   1. calculate addr of PTE (page table entry)
   2. read PTE from memory, add to TLB
3. extract PFN from TLB (page frame num)
4. build PA (phys addr)
5. read contents of PA from memory

Exercise

- Assume a virtual memory system with:
  - a linear page table per process
  - no TLB
  - no swapping (all referenced pages are found in memory)
  - one memory access takes M time units
- How long does it take to execute a single load instruction (e.g. mov VirtualAddress, register), in terms of memory accesses?

Exercise

- Assume a virtual memory system with:
  - a linear page table per process
  - no TLB
  - no swapping (all referenced pages are found in memory)
  - one memory access takes M time units
- How long does it take to execute a single load instruction (e.g. mov VirtualAddress, register), in terms of memory accesses?
- Now assume the same system but with a TLB. What is the fastest the load instruction above will execute, assuming TLB hits?

TLB issue

- TLB entries may not be valid after a context switch

Process 1's Page Table

VPN	PFN
i	d
j	b
k	a

TLB

VPN	PFN
i	d
J	b
k	a

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

VPNs are the same, but PFN mappings have changed!

Potential Solutions

- Strategy 1: clear the TLB (mark all entries as invalid) after each context switch
- Problem: forces each process to start with a cold cache

Potential Solutions

- Strategy 2: Associate an ASID (address space ID) with each process
- ASID is just like a process ID in the kernel
  - CPU can compare the ASID of the active process to the ASID stored in each TLB entry
  - If they don't match, the TLB entry is invalid