

CS 471 Operating Systems

Yue Cheng

George Mason University
Spring 2019

Paging Problems

- Page tables are too slow
- Page tables are too big

Address Translation Steps

- Hardware: for each memory reference
 1. Extract **VPN** (virt page num) from **VA** (virt addr)
 2. Calculate addr of **PTE** (page table entry)
 3. Fetch **PTE**
 4. Extract **PFN** (phys page frame num)
 5. Build **PA** (phys addr)
 6. Fetch **PA** to register

- **Q: Which steps are expensive??**

Address Translation Steps

- Hardware: for each memory reference

cheap 1. Extract **VPN** (virt page num) from **VA** (virt addr)

cheap 2. Calculate addr of **PTE** (page table entry)

expensive 3. Fetch **PTE**

cheap 4. Extract **PFN** (phys page frame num)

cheap 5. Build **PA** (phys addr)

expensive 6. Fetch **PA** to register

- Q: Which steps are expensive??

Address Translation Steps

- Hardware: for each memory reference

cheap 1. Extract **VPN** (virt page num) from **VA** (virt addr)

cheap 2. Calculate addr of **PTE** (page table entry)

expensive 3. Fetch **PTE**

cheap 4. Extract **PFN** (phys page frame num)

cheap 5. Build **PA** (phys addr)

expensive 6. Fetch **PA** to register

- Q: Which expensive steps can we avoid??

Array Iterator

- A simple code snippet in array.c

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

- Compile it using gcc

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

- Dump the assembly code
 - objdump (Linux) or otool (Mac)

Trace the Memory Accesses

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
	load 0x700C
...	

Trace the Memory Accesses

Virt	Phys
load 0x <u>3</u> 000	load 0x100 <u>C</u>
	load 0x7000
load 0x <u>3</u> 004	load 0x100 <u>C</u>
	load 0x7004
load 0x <u>3</u> 008	load 0x100 <u>C</u>
	load 0x7008
load 0x <u>3</u> 00C	load 0x100 <u>C</u>
...	load 0x700C

1st mem access: Fetch PTE

Trace the Memory Accesses

Virt	Phys
load 0x <u>3</u> 000	load 0x100C
	load 0x <u>7</u> 000
load 0x <u>3</u> 004	load 0x100C
	load 0x <u>7</u> 004
load 0x <u>3</u> 008	load 0x100C
	load 0x <u>7</u> 008
load 0x <u>3</u> 00C	load 0x100C
...	load 0x <u>7</u> 00C

Map VPN to PFN: 3 -> 7

Trace the Memory Accesses

Virt	Phys
load 0x3 <u>000</u>	load 0x100C
	load 0x7 <u>000</u>
load 0x3 <u>004</u>	load 0x100C
	load 0x7 <u>004</u>
load 0x3 <u>008</u>	load 0x100C
	load 0x7 <u>008</u>
load 0x3 <u>00C</u>	load 0x100C
...	load 0x7 <u>00C</u>

2nd mem access: access **a[i]**

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
...	load 0x700C

- Note: 1. Each virt mem access → two phys mem accesses
2. Repeated memory accesses!

Translation Lookaside Buffer (TLB)

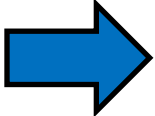
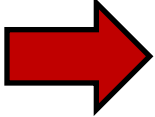
Performance Problems of Paging

- A basic memory access protocol
 1. Fetch the translation from in-memory page table
 2. Explicit load/store access on a memory address
- In this scheme every data/instruction access requires **two** memory accesses
 - One for the page table
 - and one for the data/instruction
- Too much performance overhead!

Speeding up Translation

- The two memory access problem can be solved by the use of a special **fast-lookup hardware cache** called **translation lookaside buffer** (TLB)
- A TLB is part of the memory-management unit (MMU)
- A TLB is a hardware cache
- Algorithm sketch
 - For each virtual memory reference, hardware first checks the TLB to see if the desired translation is held therein

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a **TLB hit** – extract PFN from the TLB entry, concatenate it onto the offset to form the PA  Fast path
4. If it is a **TLB miss** – access *page table* to get the translation, update the TLB entry with the translation  Slow path

Array Iterator (w/ TLB)

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Trace the Memory Accesses (w/ TBL)

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	1

CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

Virt

load 0x3000

Phys

load 0x3004

load 0x3008

load 0x300C

...

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	1

Virt

Phys

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

Miss

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt
load 0x3000

Phys
load 0x100C

load 0x3004

load 0x3008

load 0x300C

...

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt
load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Phys
load 0x100C
load 0x7000

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

Virt

load 0x3000

Phys

load 0x100C

load 0x7000

load 0x3004

load 0x3008

load 0x300C

...

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Phys

load 0x100C

load 0x7000

(TLB hit)

CPU's TLB cache

	Valid	Virt	Phys
Hit →	1	3	7
	0		
	0		
	0		

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

Phys

load 0x100C

load 0x7000

(TLB hit)

load 0x7004

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
...	load 0x700C

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
...	
load 0x1000	load 0x700C

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	1	3
0		
0		

Miss

Virt

Phys

load 0x3000

load 0x100C

load 0x7000

load 0x3004

(TLB hit)

load 0x7004

load 0x3008

(TLB hit)

load 0x7008

load 0x300C

(TLB hit)

...

load 0x700C

load 0x1000

load 0x100F

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	1	3
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
...	load 0x700C
load 0x1000	load 0x100F
	load 0x3000

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	1	3
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
...	load 0x700C
load 0x1000	load 0x100F
	load 0x3000
load 0x1004	

Trace the Memory Accesses (w/ TBL)

P1's page table

0	5
1	3
2	4
3	7

CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	1	3
0		
0		

Hit 

Virt

Phys

load 0x3000

load 0x100C

load 0x7000

load 0x3004

(TLB hit)

load 0x7004

load 0x3008

(TLB hit)

load 0x7008

load 0x300C

(TLB hit)

...

load 0x700C

load 0x1000

load 0x100F

load 0x3000

load 0x1004

(TLB hit)

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```


How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array a has 1024 items, each item is 4 bytes:

`Size(a) = 4096`

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array a has 1024 items, each item is 4 bytes:

$\text{Size}(a) = 4096$

Num of TLB miss: $4096/4096 = 1$ or 2

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array a has 1024 items, each item is 4 bytes:

$\text{Size}(a) = 4096$

Num of TLB miss: $4096/4096 = 1$

TLB miss rate: $1/1024 = 0.09\%$

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array a has 1024 items, each item is 4 bytes:

$\text{Size}(a) = 4096$

Num of TLB miss: $4096/4096 = 1$

TLB miss rate: $1/1024 = 0.09\%$

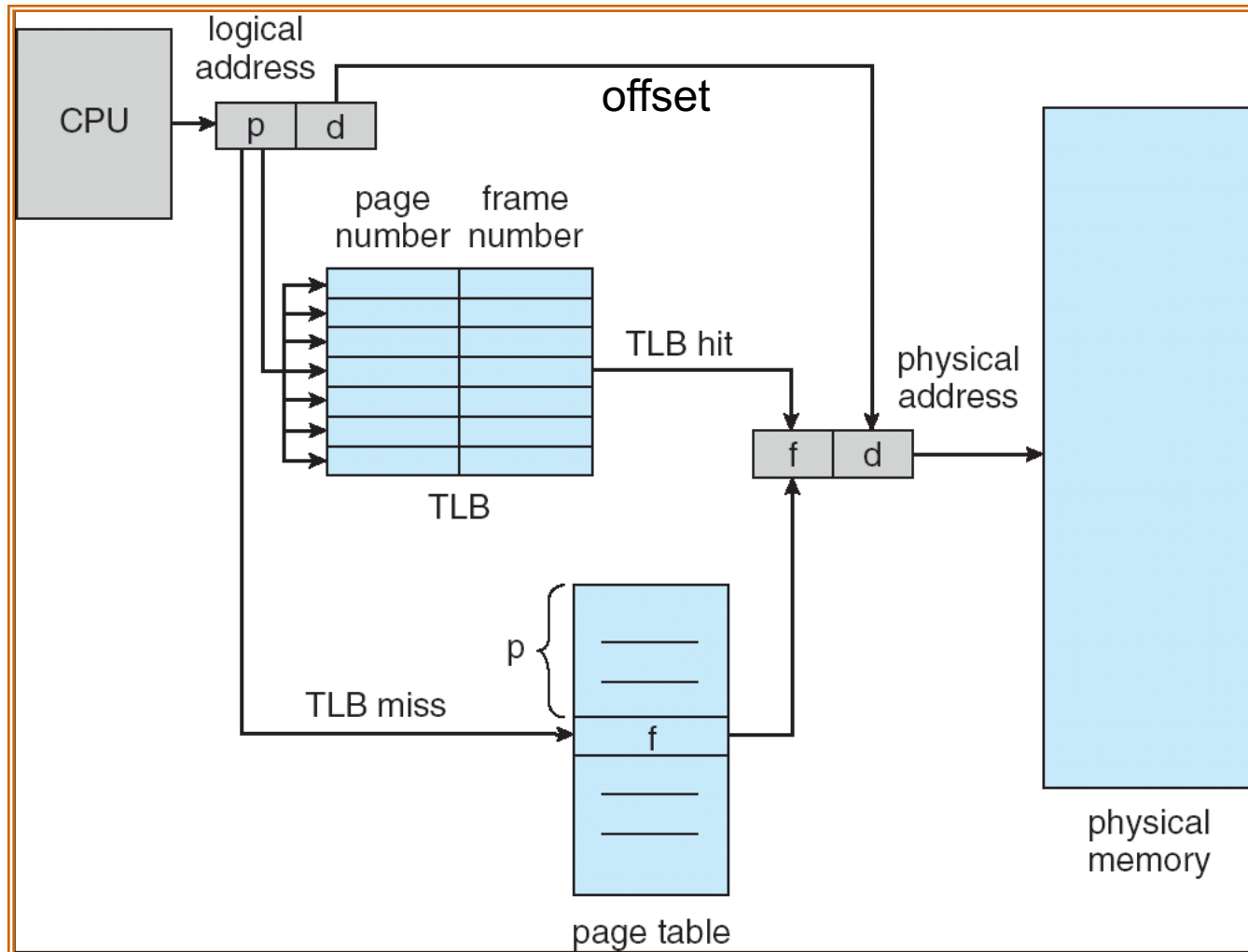
TLB hit rate : 99.91% (almost 100%)

TLB Content

- Some entries are [wired down or reserved] for permanently valid translations
- TLB is a **fully associative** cache
 - Any given translation can be anywhere in the TLB
 - Hardware searches entire TLB **in parallel** to find a match
- A typical TLB entry

VPN | PFN | other bits

Paging Hardware w/ TLB



TLB Issue: Context Switch

- TLB contains translations only valid for the **currently running** process
- Switching from one process to another requires OS or hardware to do more work

One Example

- How does OS distinguish which entry is for which process?

	VPN	PFN	valid	prot
P1	10	100	1	rwX
	—	—	0	—
P2	10	170	1	rwX
	—	—	0	—

One Simple Solution: Flush

- OS flushes the whole TLB on context switch
- Flush operation sets all valid bit to 0

One Simple Solution: Flush

- OS flushes the whole TLB on context switch
- Flush operation sets all valid bit to 0
- **Problem: the overhead is too high if OS switches processes too frequently**

Optimization: ASID

- Some hardware systems provide an **address space identifier (ASID)** field in the TLB
- Think of ASID as a **process identifier (PID)**
 - An 8-bit field

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

Page Sharing

- Leveraging ASID for supporting page sharing
- In this example, two entries from two processes with two different VPNs point to the same physical page

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

Page Sharing (cont.)

- Shared code
 - One copy of read-only (reentrant) code shared among processes (e.g., text editors, compilers, window systems)
 - Particularly important for time-sharing environments
- Private code and data
 - Each process keeps a separate copy of the code and data

TLB Replacement Policy

- Cache: When we want to add a new entry to a **full** TLB, an old entry must be evicted and replaced
- Least-recently-used (LRU) policy
 - Intuition: A page entry that has not recently been used implies it won't likely to be used in the near future
- Random policy
 - Evicts an entry at random

TLB Workloads

- **Sequential** array accesses can almost always hit in the TLB, and hence are **very fast**
- What pattern would be **slow**?

TLB Workloads

- **Sequential** array accesses can almost always hit in the TLB, and hence are **very fast**
- What pattern would be **slow**?
 - **Highly random**, with no repeat accesses

Workload Characteristics

Workload A

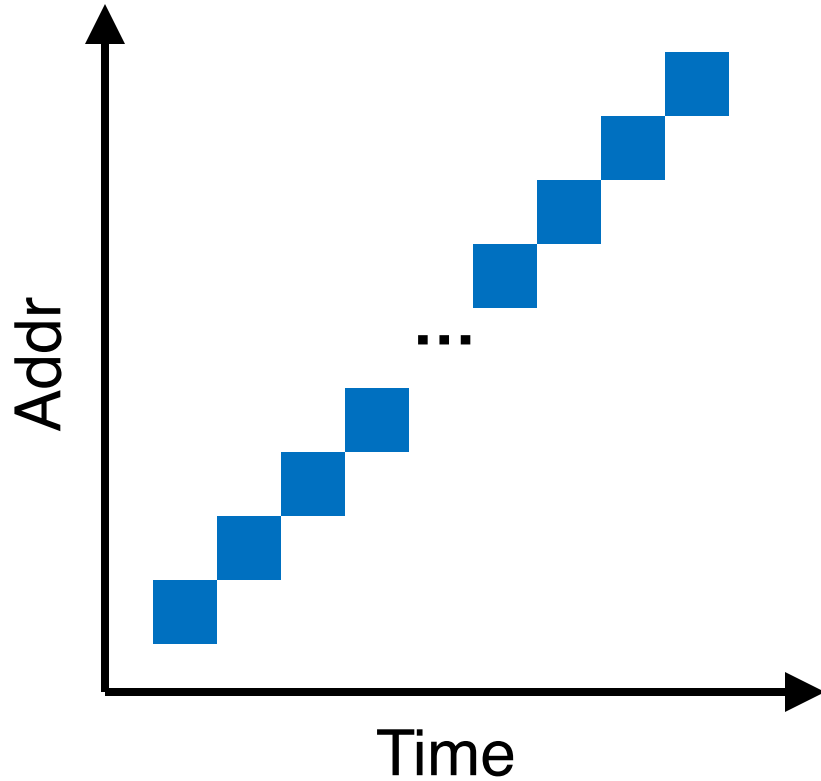
```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Workload B

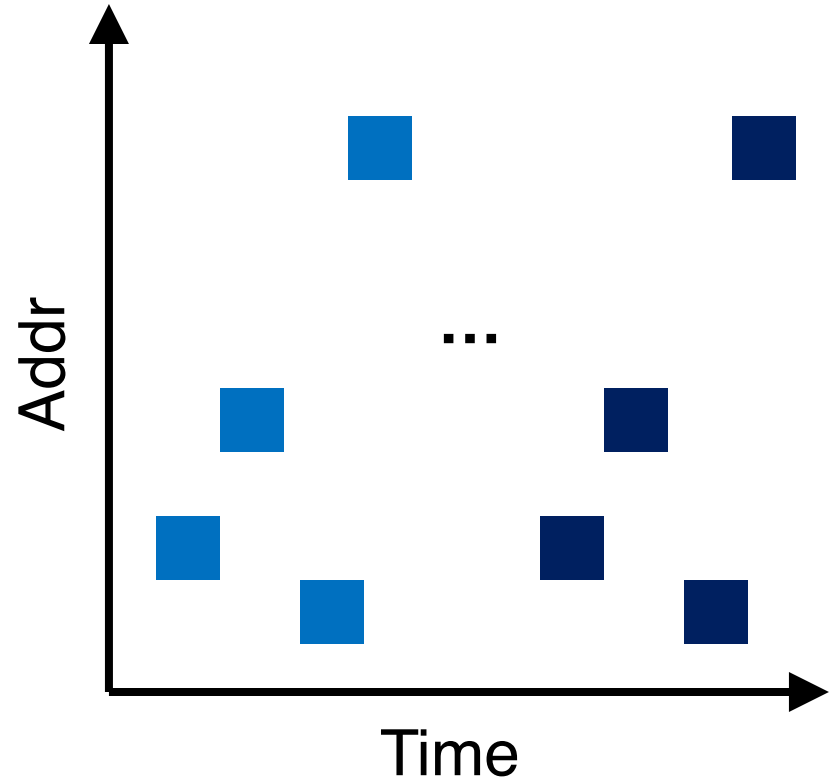
```
int sum = 0;
srand(1234);
for (i=0; i<512; i++) {
    sum += a[rand() % N];
}
srand(1234); // same seed
for (i=0; i<512; i++) {
    sum += a[rand() % N];
}
```

Access Patterns

Workload A

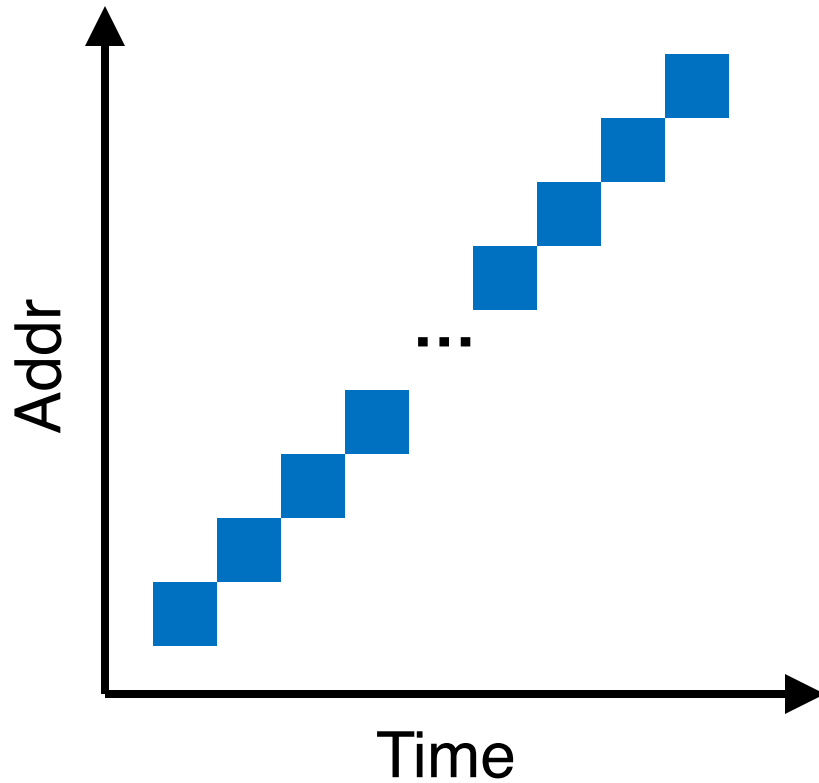


Workload B



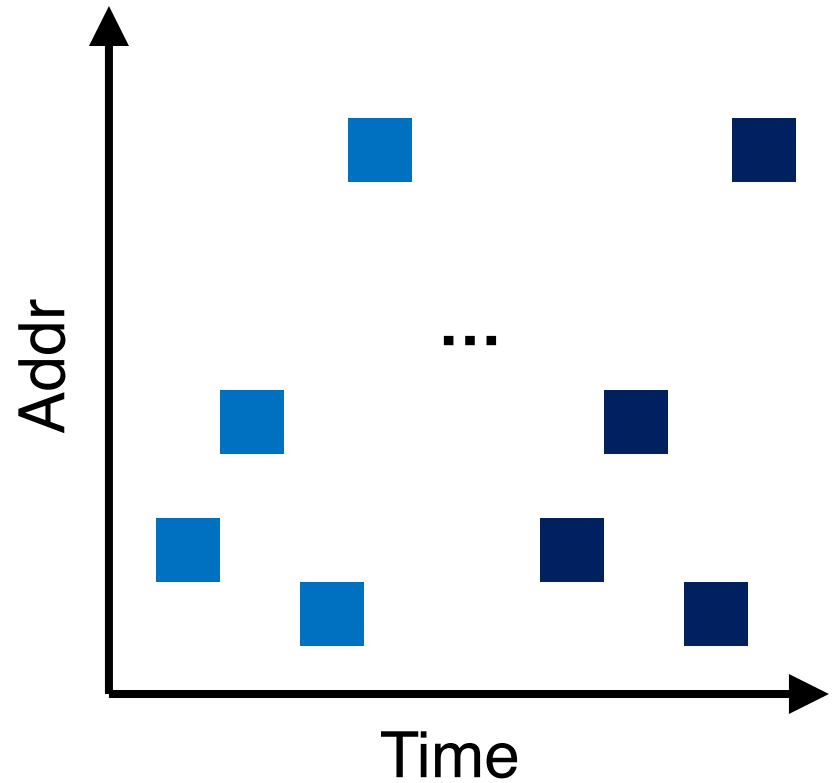
Access Patterns

Workload A



Spatial Locality

Workload B



Temporal Locality

Workload Locality

- **Spatial locality:**
 - Future access will be to nearby addresses
- **Temporal locality:**
 - Future access will be repeated to the same data

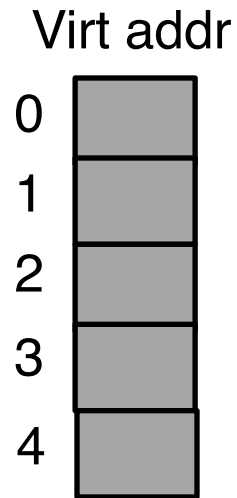
Workload Locality

- **Spatial locality:**
 - Future access will be to nearby addresses
- **Temporal locality:**
 - Future access will be repeated to the same data
- Q: What TLB characteristics are best for each type?

TLB Replacement Policy

- Cache: When we want to add a new entry to a **full** TLB, an old entry must be evicted and replaced
- **Least-recently-used (LRU)** policy
 - Intuition: A page entry that has not recently been used implies it won't likely to be used in the near future
- **Random** policy
 - Evicts an entry at random

LRU Trouble

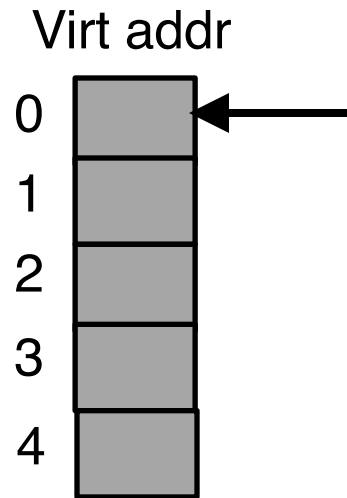


CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

The table represents the CPU's TLB cache. It has three columns: Valid, Virt, and Phys. The Valid column contains the value 0 for all four entries, indicating that no entries are currently valid in the cache. The Virt and Phys columns are empty.

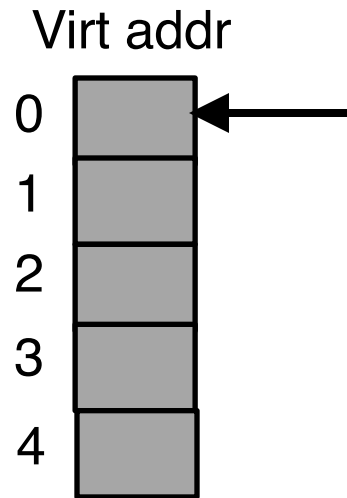
LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
0		
0		
0		

LRU Trouble

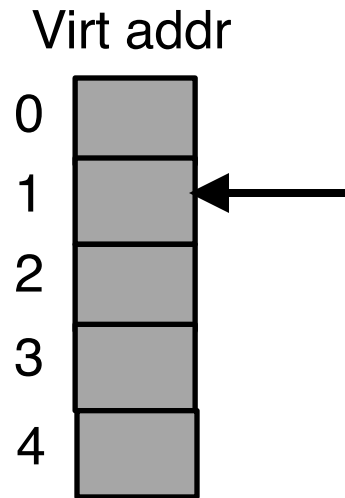


CPU's TLB cache

Valid	Virt	Phys
1	0	?
0		
0		
0		

TLB miss

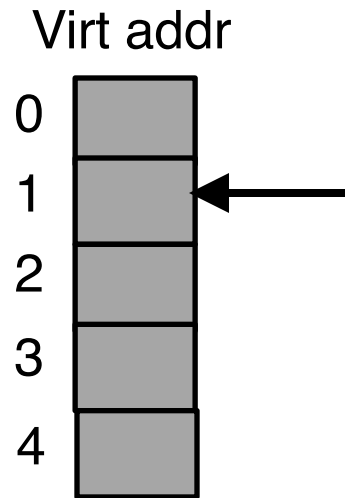
LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

LRU Trouble

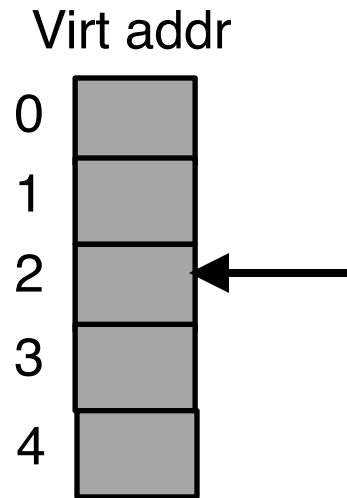


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

TLB miss

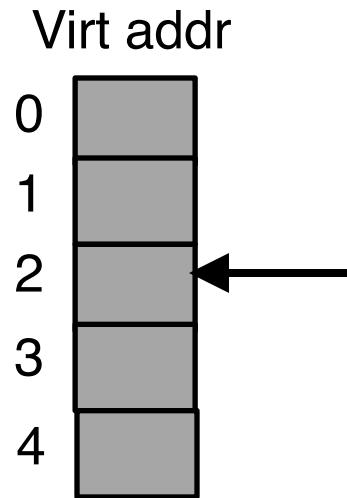
LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

LRU Trouble

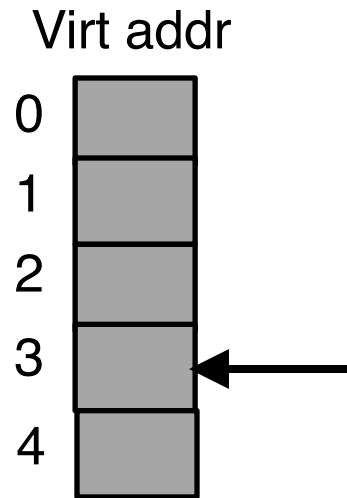


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

TLB miss

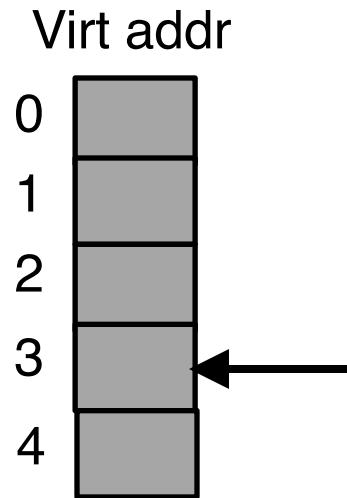
LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

LRU Trouble

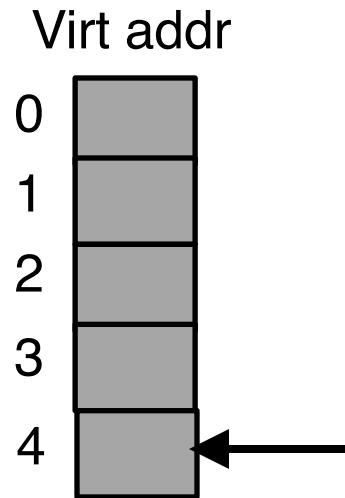


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

TLB miss

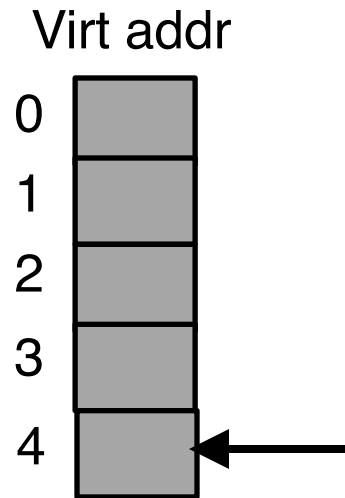
LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

LRU Trouble

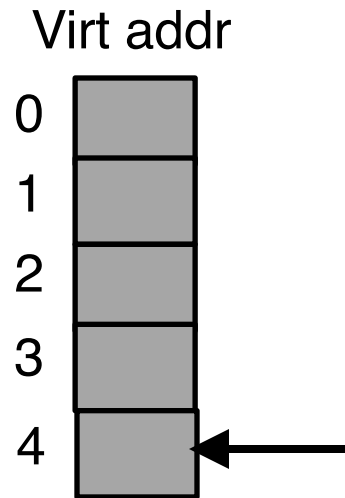


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

Now, **0** is the least-recently used item in TLB

LRU Trouble

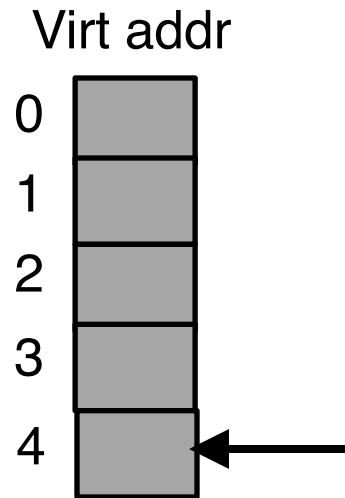


CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
1	3	?

Replace 0 with 4

LRU Trouble



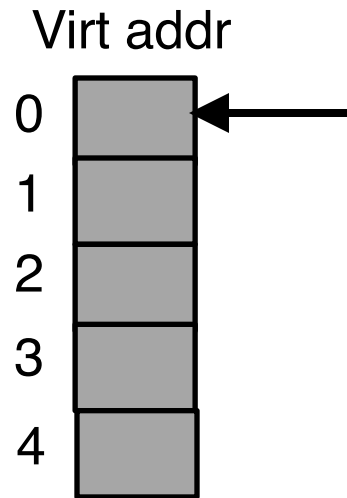
CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
1	3	?

TLB miss

Replace 0 with 4

LRU Trouble

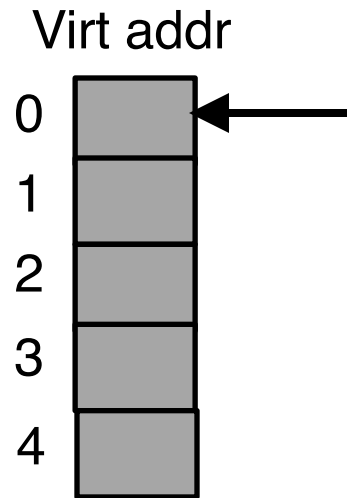


CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
1	3	?

Accessing 0 again, which was unfortunately just evicted...
Replace 1 (which is the least-recently used item at this point, with 0...

LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
1	3	?

TLB miss

Accessing 0 again, which was unfortunately just evicted...
Replace 1 (which is the least-recently used item at this point, with 0...

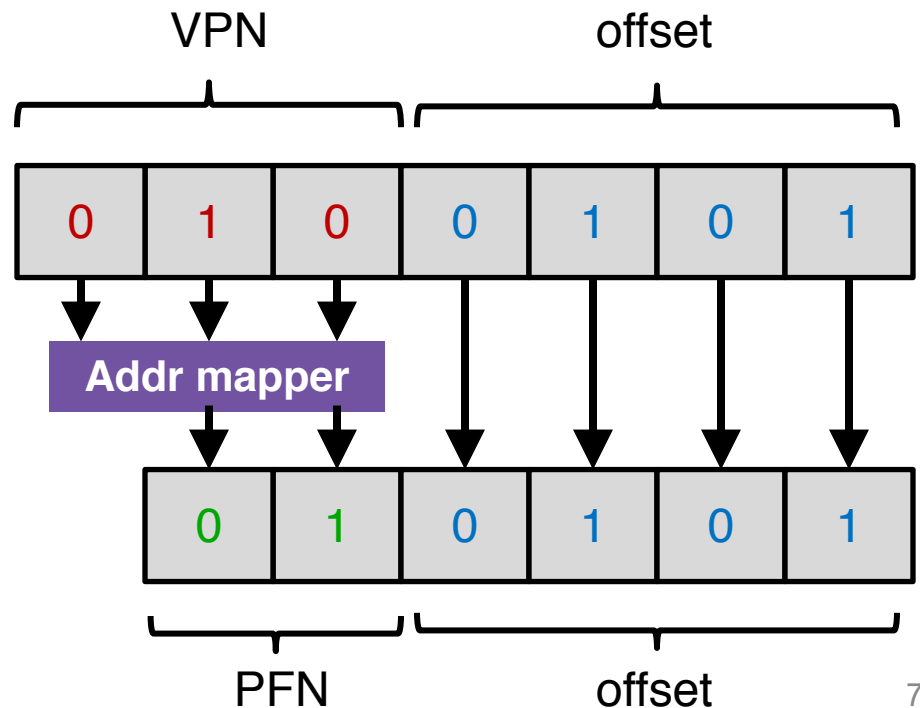
Takeaway

- LRU
- Random
- When is each better?
 - Sometimes random is better than a “smart” policy!

Review: Page Table

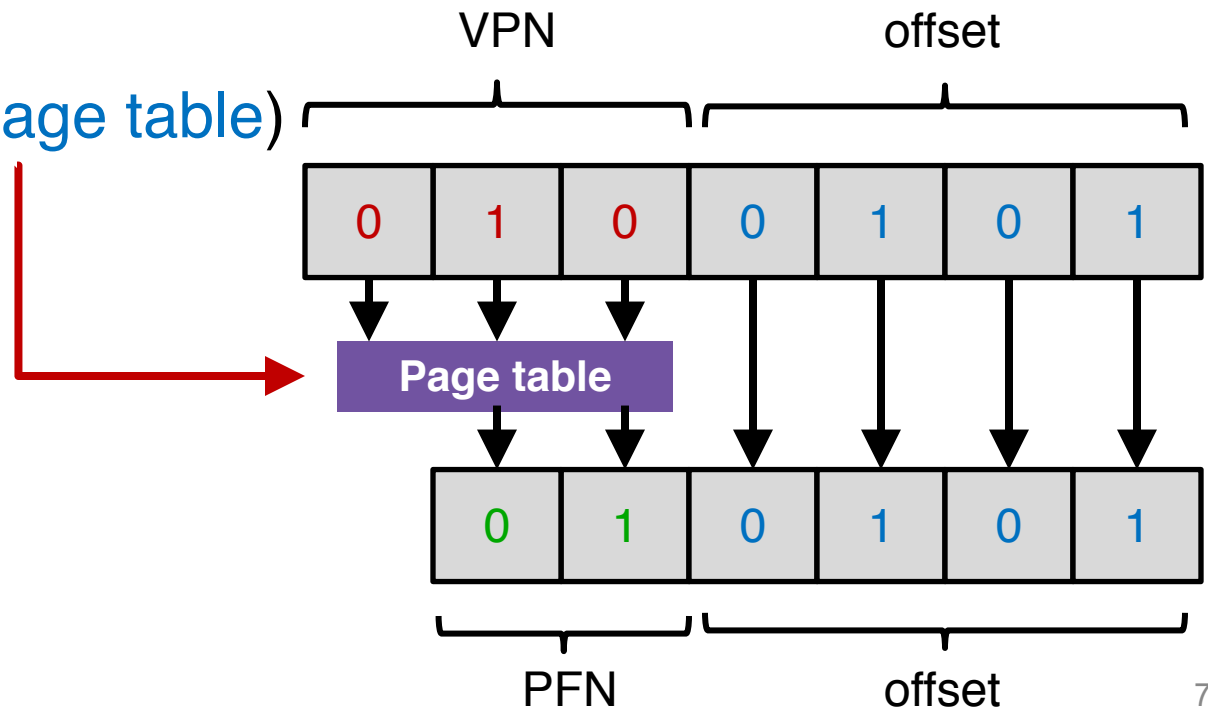
Virtual => Physical Addr Mapping

- We need a general mapping mechanism
- What data structure is good?
 - Big array

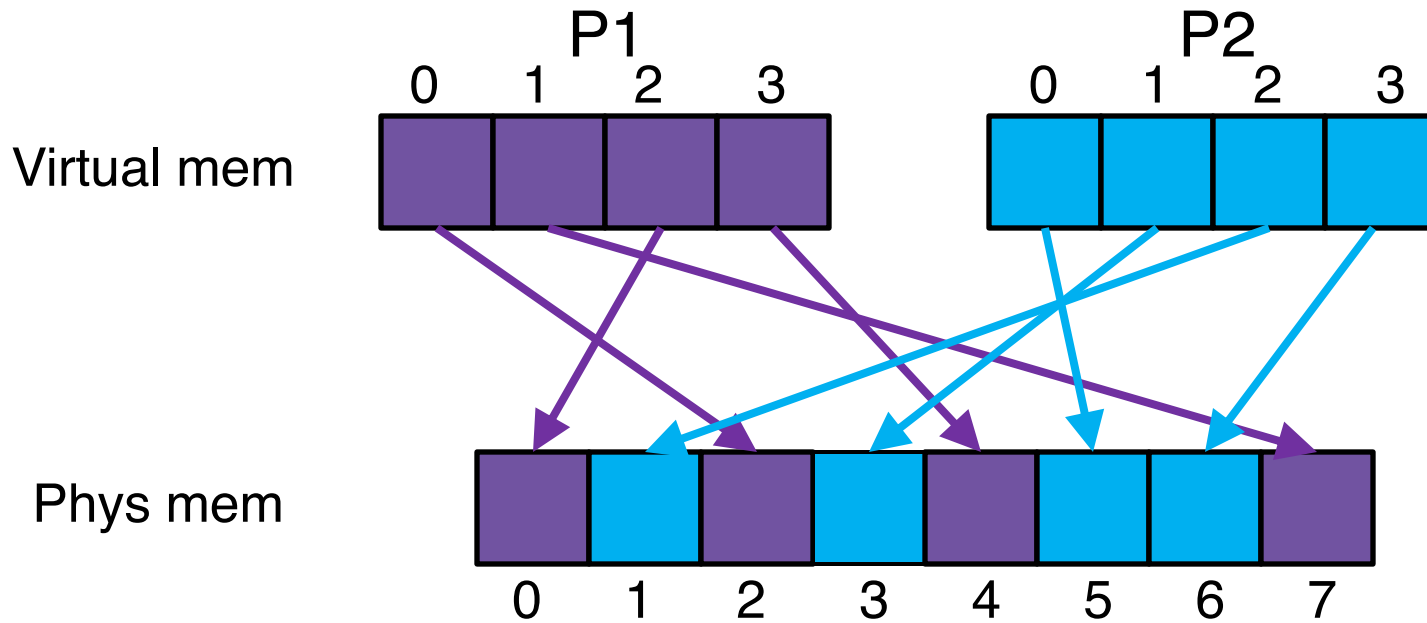


Virtual => Physical Addr Mapping

- We need a general mapping mechanism
- What data structure is good?
 - Big array
 - (aka **linear page table**)



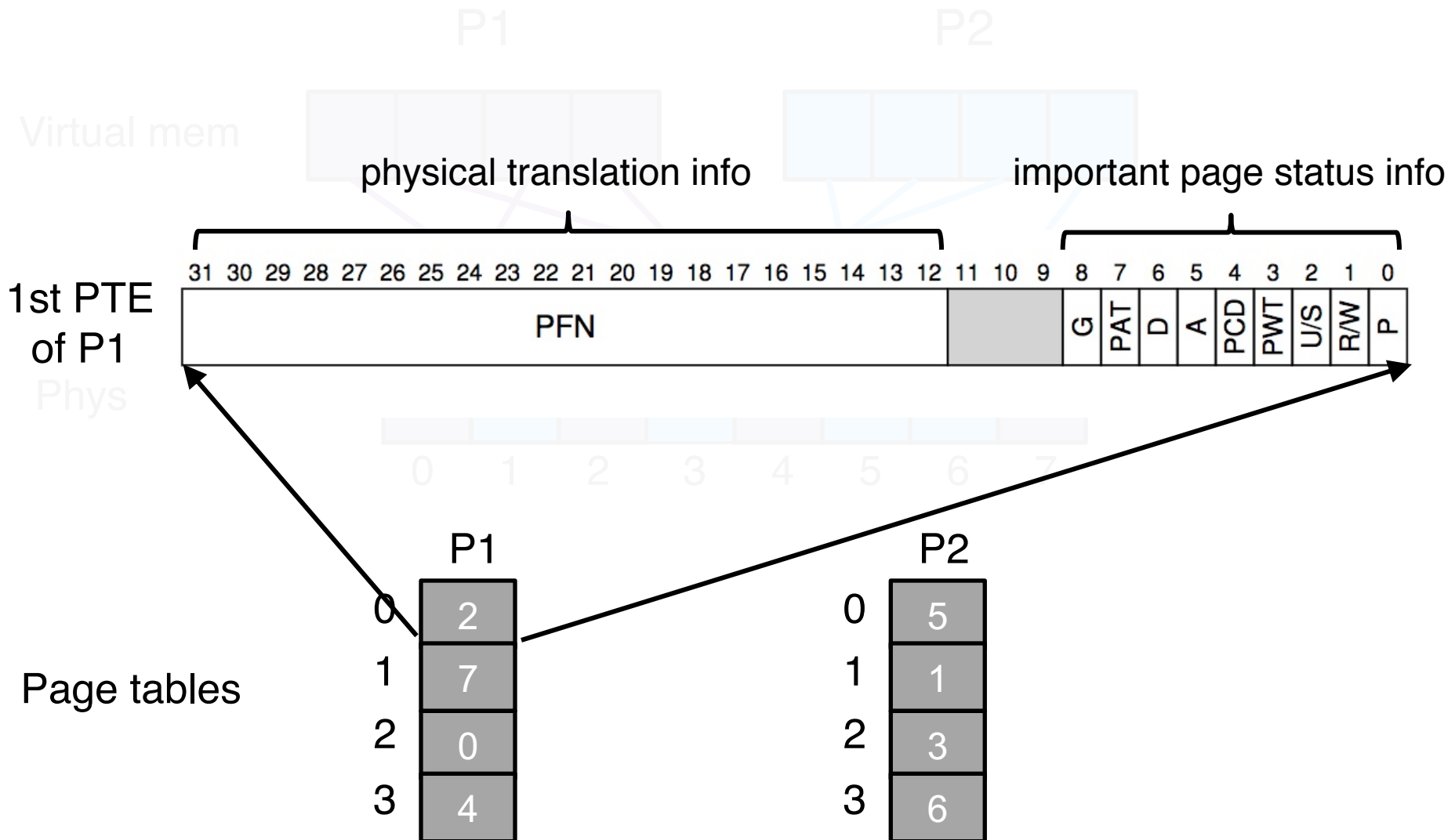
A Simple Page Table Example



Page tables

P1		P2	
0	2	0	5
1	7	1	1
2	0	2	3
3	4	3	6

A Simple Page Table Example




Paging Problems

- Page tables are too slow (**just covered**)
 - TLB to the rescue!
- Page tables are too big (**now**)

How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4KB))} * 4 = 4MB$


How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4\text{KB}))} * 4 = 4\text{MB}$



How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\underbrace{\log(4\text{KB})}_{\text{offset bits}})} * 4 = 4\text{MB}$

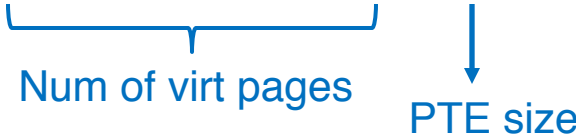
How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4KB))} * 4 = 4MB$


How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4KB))} * 4 = 4MB$

Num of virt pages

How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4\text{KB}))} * 4 = 4\text{MB}$ 

Page Tables are Too Big

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
 - How many pages: 2^{20} pages
 - How much memory: **4MB** assuming each page-table entry is of 4 bytes
 - $2^{(32-\log(4KB))} * 4 = 4MB$
- One page table for one process!
 - A system with 100 process: **400MB** only for storing page tables in memory
- Solution??

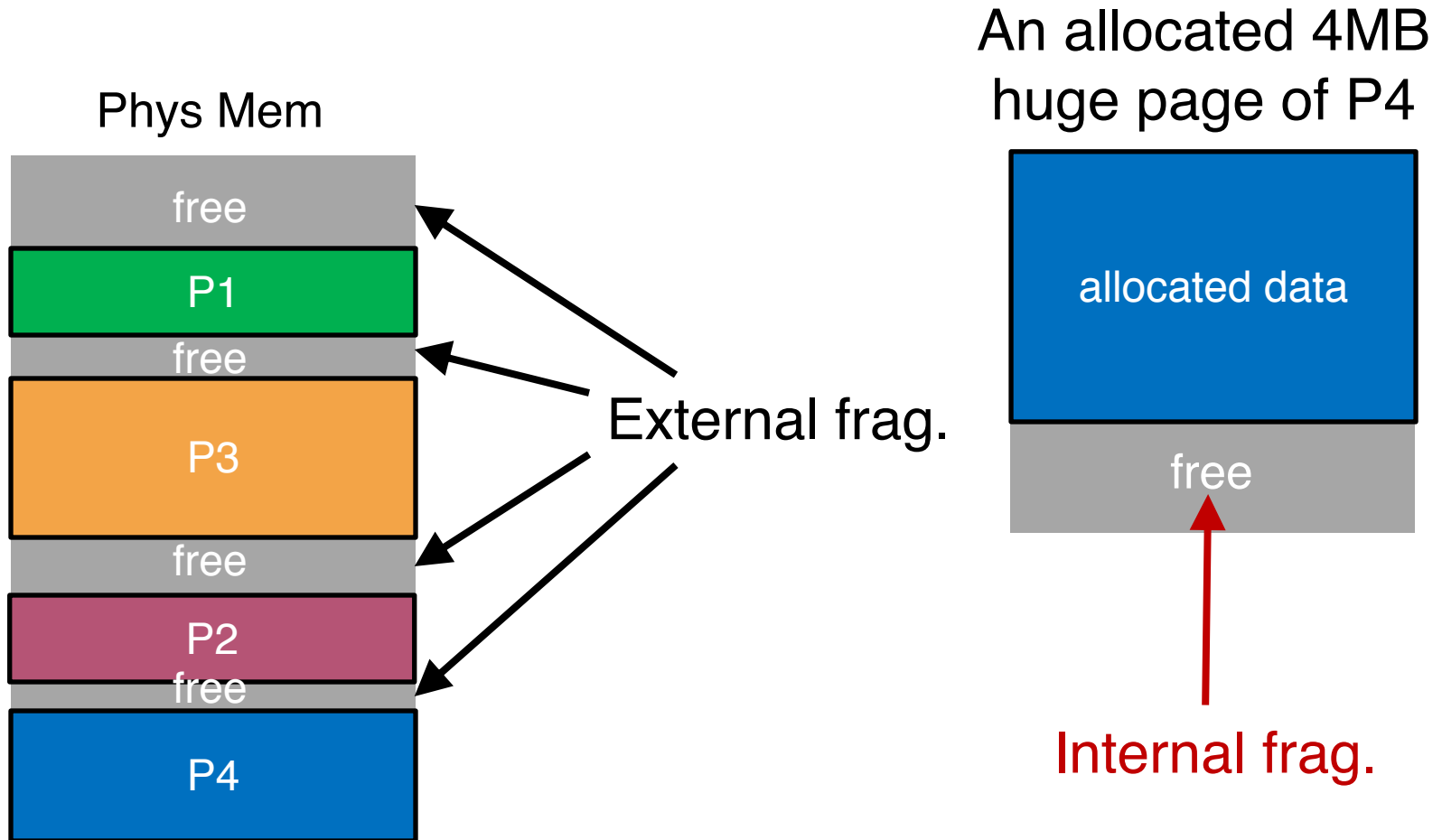
Naïve Solution

- Reduce the granularity
 - by **increasing** the page size

Naïve Solution

- Reduce the granularity
 - by **increasing** the page size
- Why are 4MB pages bad?
 - **Internal fragmentation!**

Fragmentation



Assume each process consists of multiple 4MB pages

Fragmentation



Approaches

- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

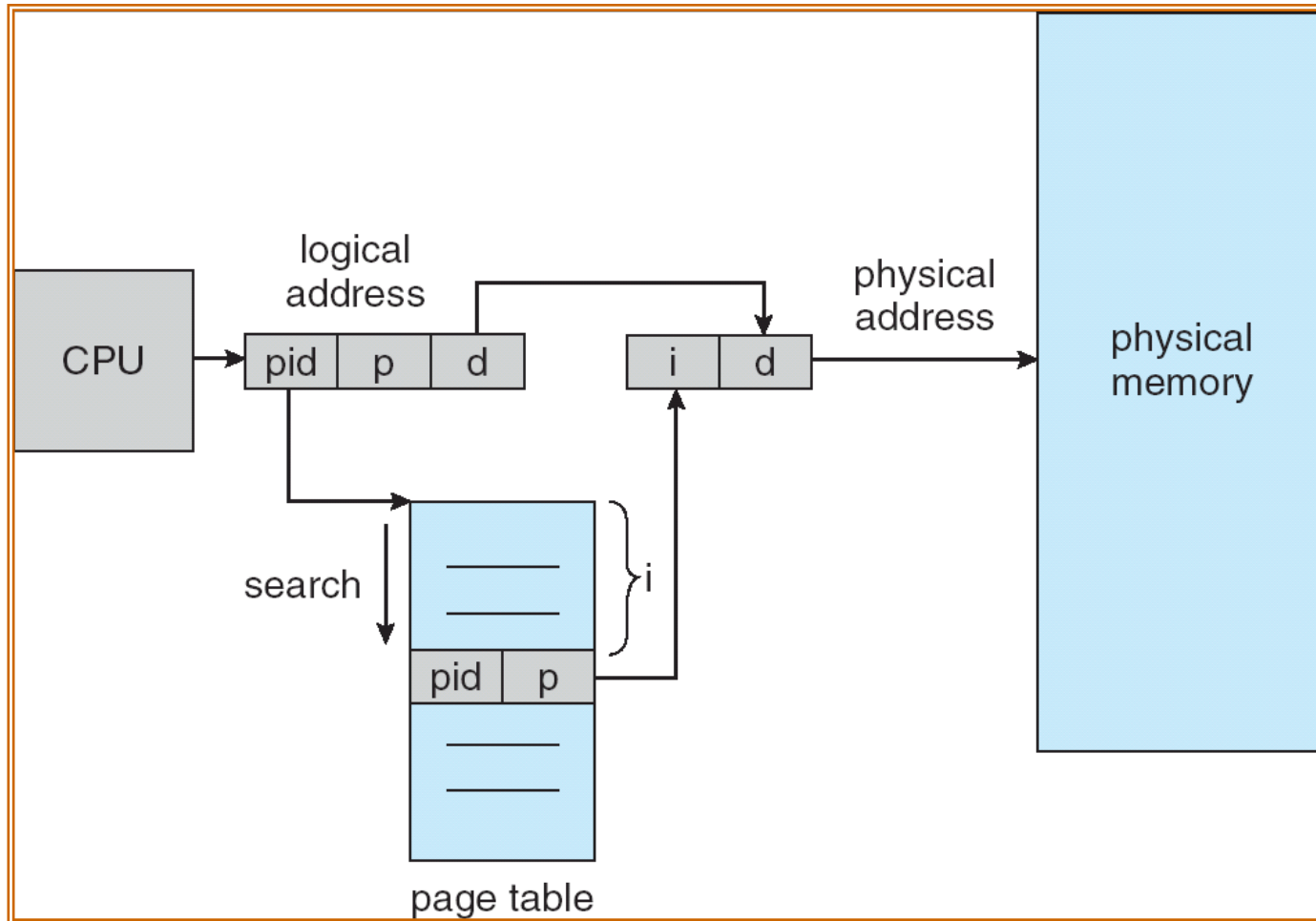
Approaches

- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

Linear Inverted Page Table

- Idea: Instead of keeping one page table per process, the system keeps a single page table that has an entry for each physical frame of the system
- Each entry tells which process owns the page, and VPN to PFN translation

Linear Inverted Page Table Example



Linear Inverted Page Table

- Idea: Instead of keeping one page table per process, the system keeps **a single page** table that has an entry for **each physical frame** of the system
- Each entry tells **which process owns** the page, and **VPN to PFN** translation
- Goal: use linear search to find the index i
 - The reason why it's called **“inverted”**
- **Pros**: Extreme memory savings
- **Cons**: A linear search is expensive
 - **Solution??**

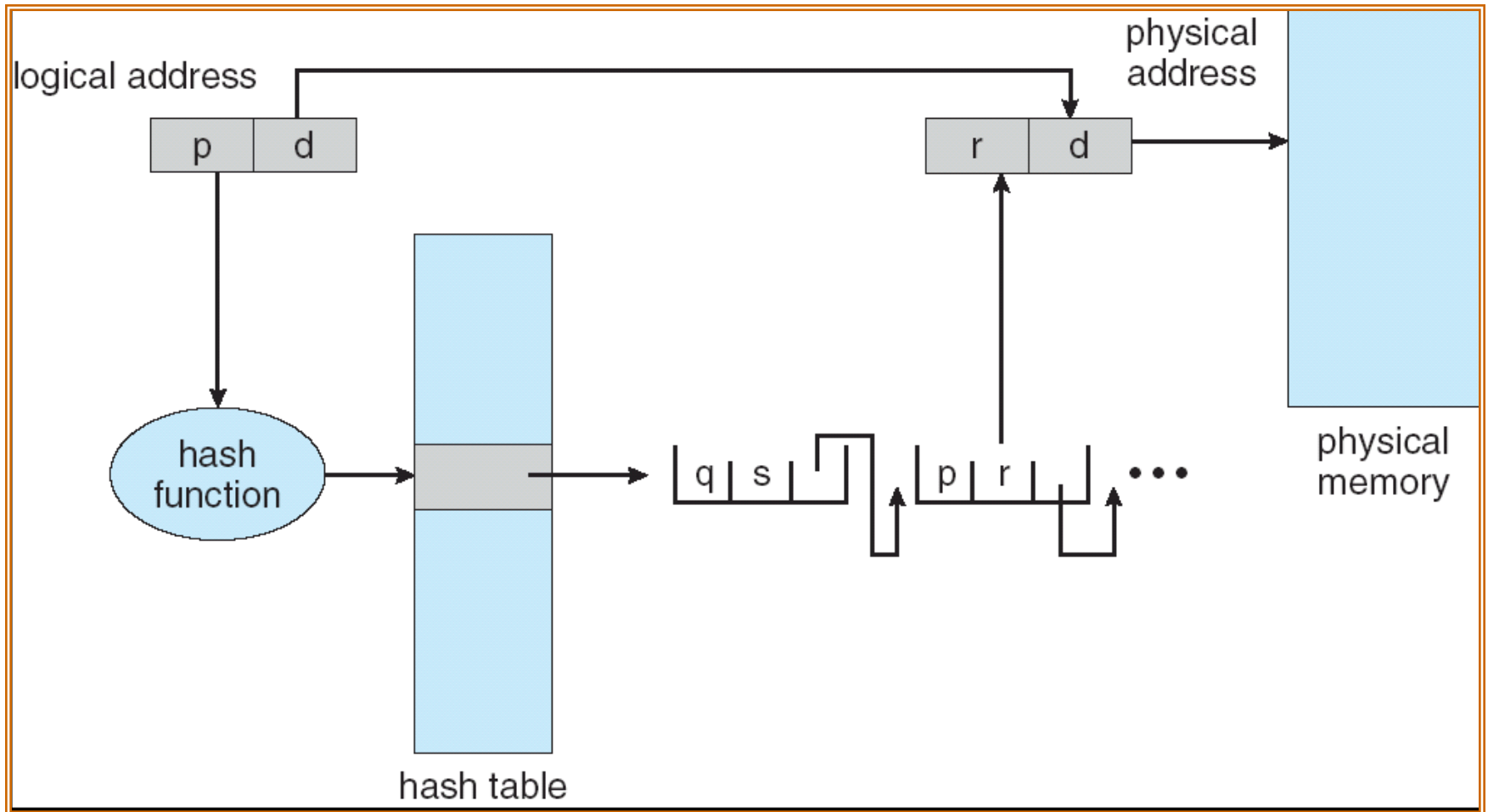
Approaches

- Approach 1: Linear Inverted Page Table
- **Approach 2: Hashed Inverted Page Table**
- Approach 3: Multi-level Page Table

Hashed Inverted Page Table

- For large address spaces, a **hashed page table** can be used, with the hash value being the **VPN**
- Idea:
 - A PTE contains a chain of elements hashing to the same location (to handle collisions) within PT
 - Each element has three fields: (a) **VPN**, (b) **PFN**, (c) **a pointer to the next element** in the linked list
 - **VPNs** are compared in this chain searching for a match. If a match is found, the corresponding **PFN** is extracted

Hashed Inverted Page Table Example



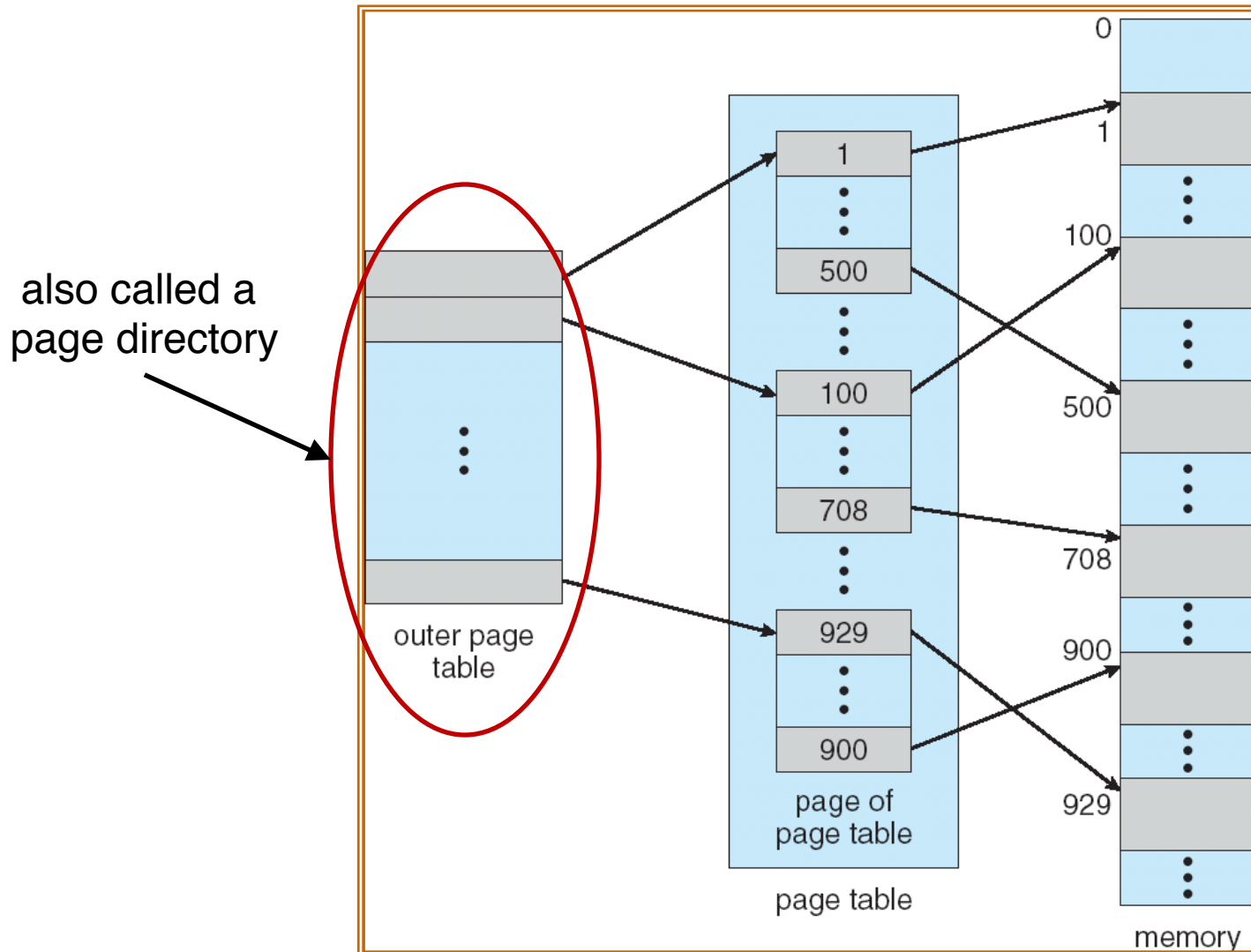
Approaches

- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- **Approach 3: Multi-level Page Table**

Multi-level Page Table

- Idea:
 - Break the page table into pages (the entire page table is **paged!**)
 - Only have pieces with >0 valid entries
 - Don't allocate the page of page table if the entire page of page-table entries is invalid
- Used by x86
- A simple technique is a two-level page table

Two-level Page Table Example



Two-level Paging

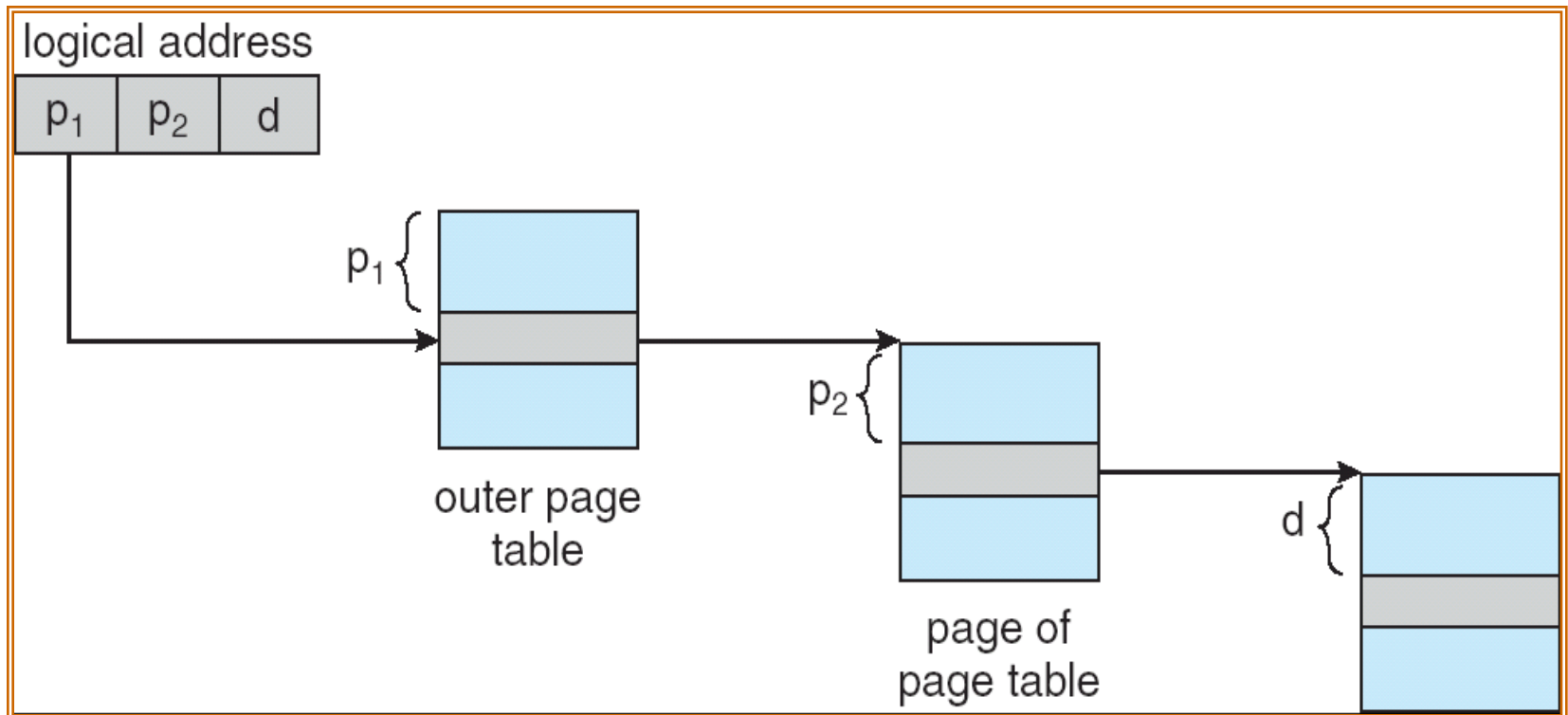
- A logical address (on 32-bit machine with 4KB page size) is divided into
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- A page table entry is 4 bytes
- Since the page table is paged, the page number is further divided into
 - p_1 : a 10-bit page directory index
 - p_2 : a 10-bit page table index
- The logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

Address Translation Scheme

- Address translation scheme for a two-level 32-bit paging architecture



> 2 Levels

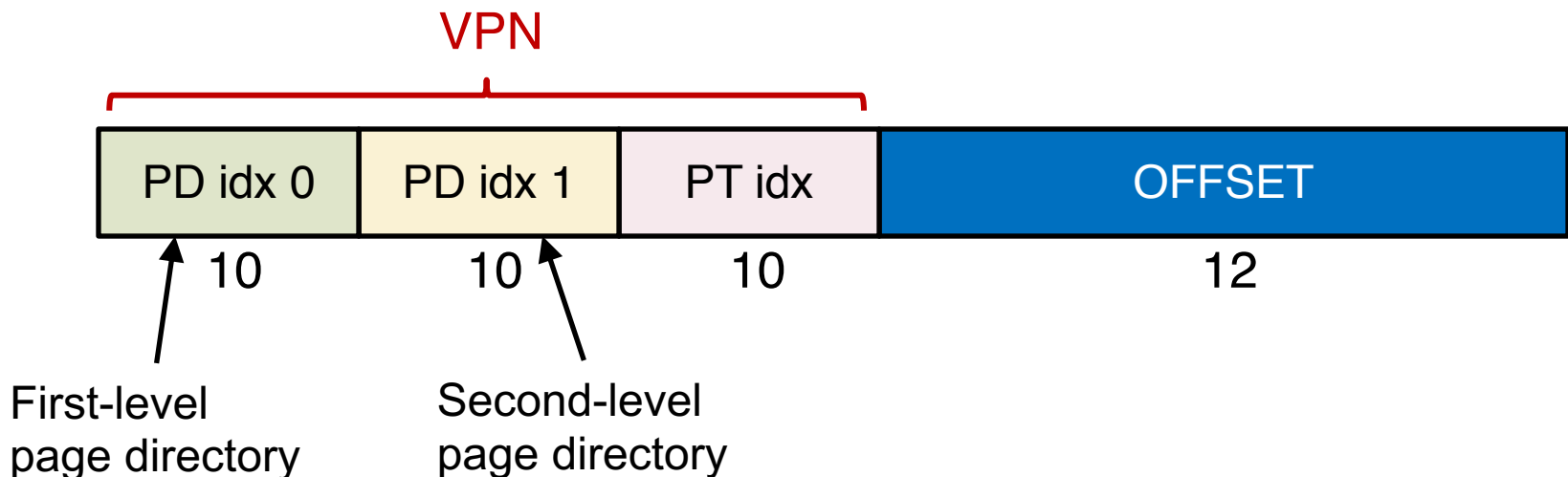
- Problem: page directory **may not fit** in a page
- Solution:
 - Split page directories into pieces
 - Use another page dir to refer to the page dir pieces

> 2 Levels

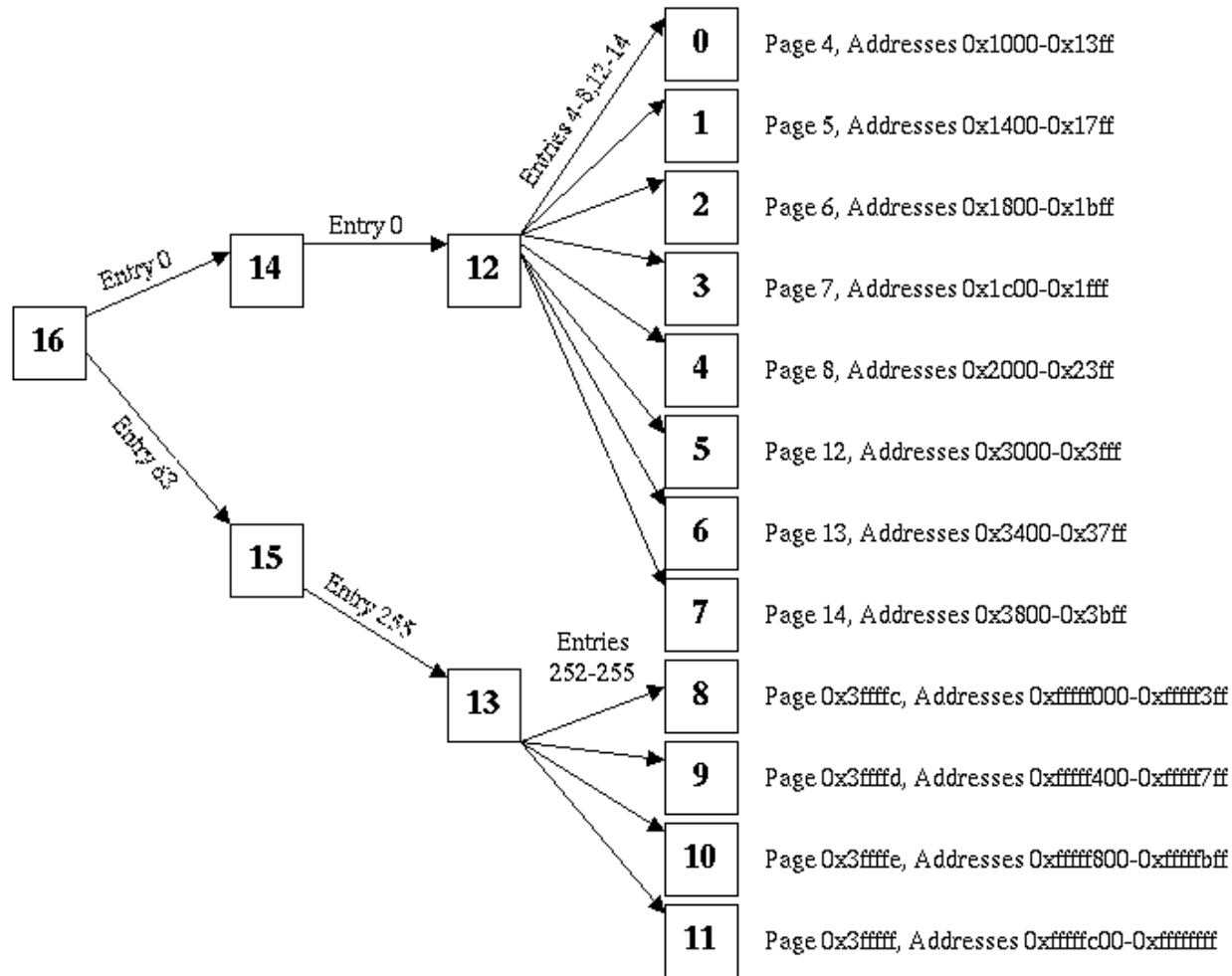
- Problem: page directory **may not fit** in a page
- Solution:
 - Split page directories into pieces
 - Use another page dir to refer to the page dir pieces
- Possible to extend to 3- or 4-level
- E.g., 64-bit Ultra-SPARC would need 7 levels of paging

> 2 Levels

- Problem: page directory **may not fit** in a page
- Solution:
 - Split page directories into pieces
 - Use another page dir to refer to the page dir pieces



Multi-level Page Table Example



<http://web.eecs.utk.edu/~mbeck/classes/cs560/560/oldtests/t2/2003/Answers.html>