

Virtual Memory

Use main memory as a "cache" for secondary (disk) storage

- Managed jointly by CPU hardware and the operating system (OS)

Programs share main memory

- Each gets a private virtual address space holding its frequently used code and data
- Protected from other programs

CPU and OS translate virtual addresses to physical addresses

- VM "block" is called a page
- VM translation "miss" is called a page fault

Virtual address $\leftrightarrow$ Physical address

(App. view)

(Managed by kernel)

Virtual Memory

Example

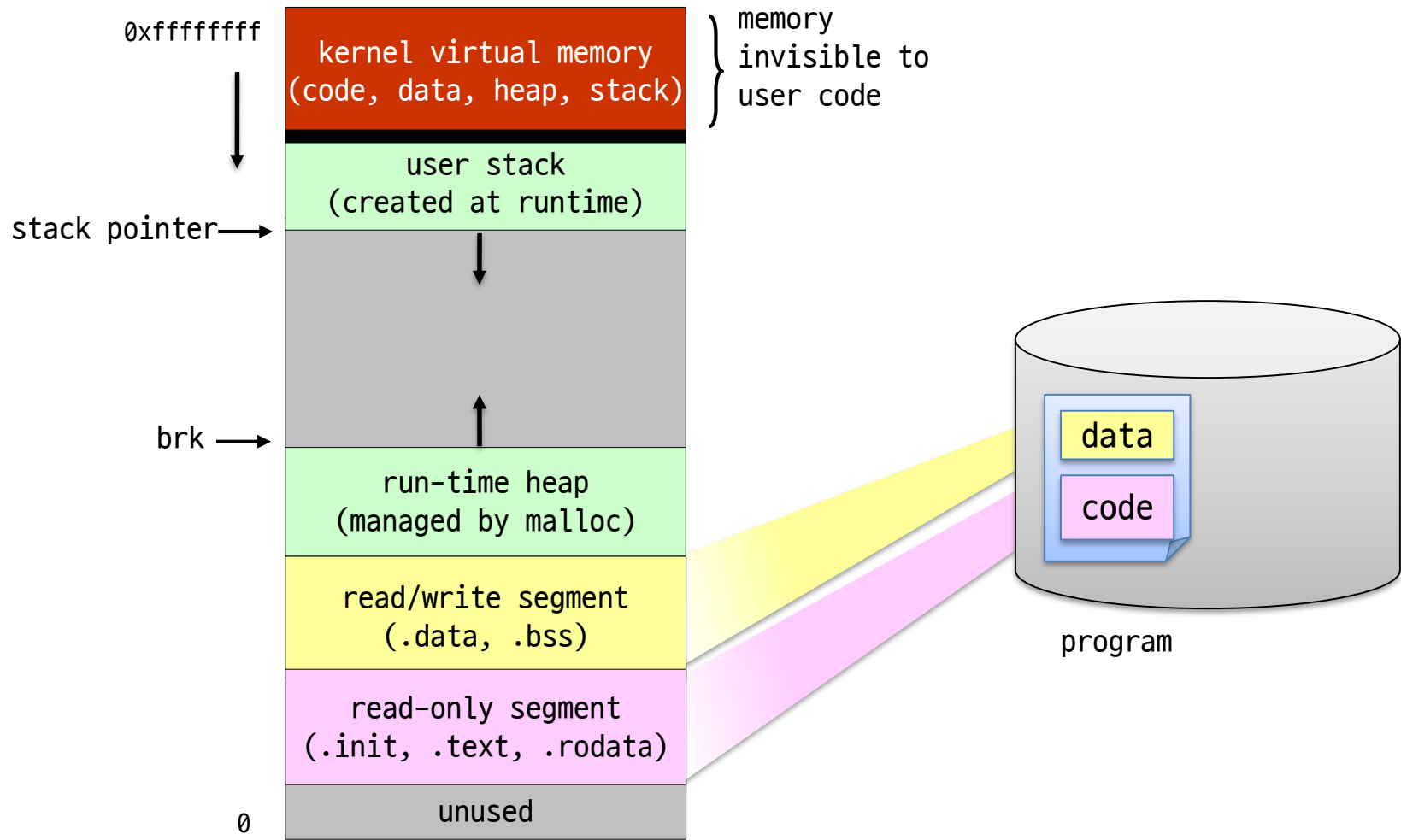
```
#include <stdio.h>

int n = 0;

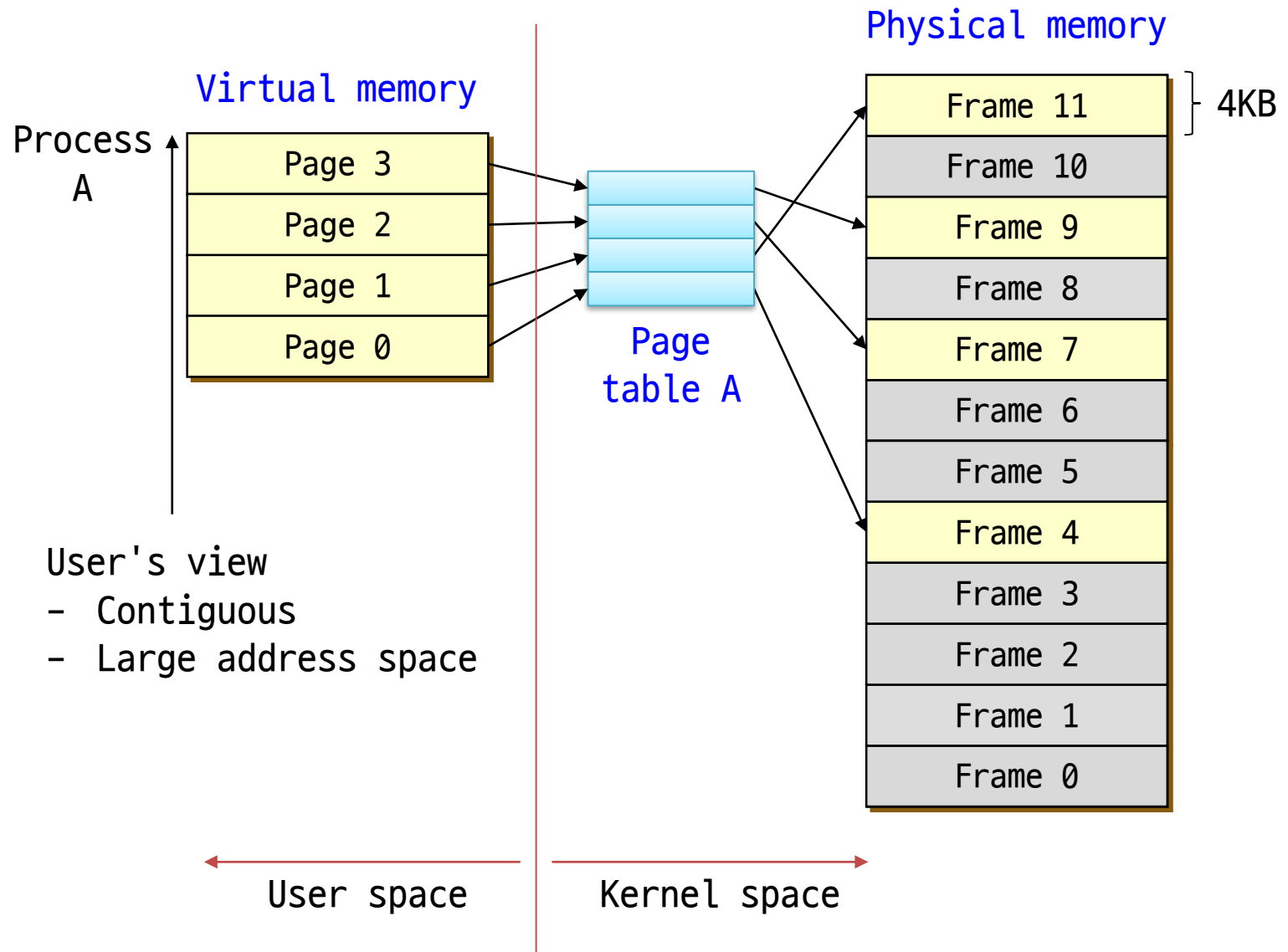
int main ()
{
    printf ("%n = 0x%08x\n", &n);
}

% ./a.out
&n = 0x08049508
```

Process in memory



Paging Introduction

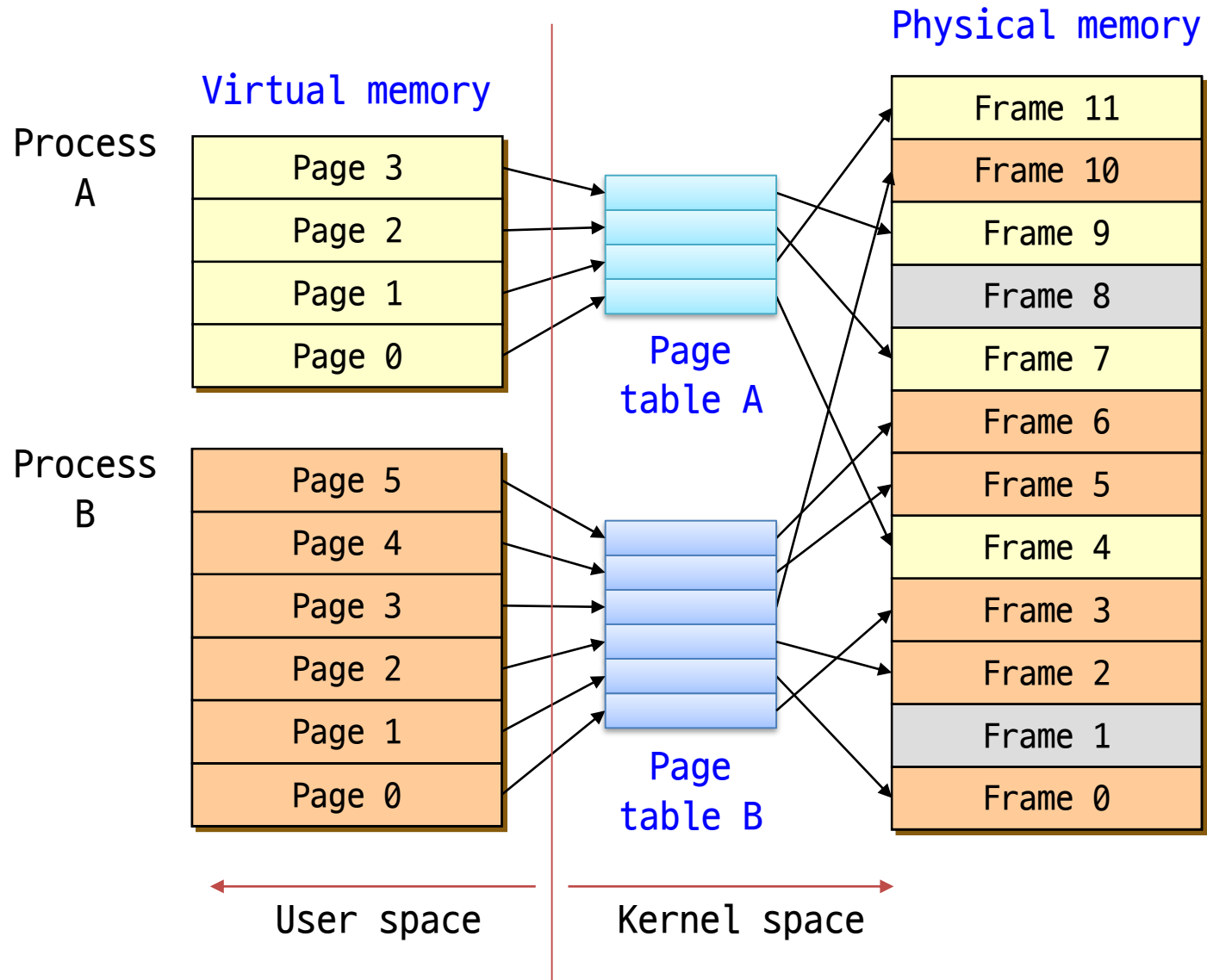


Paging (1)

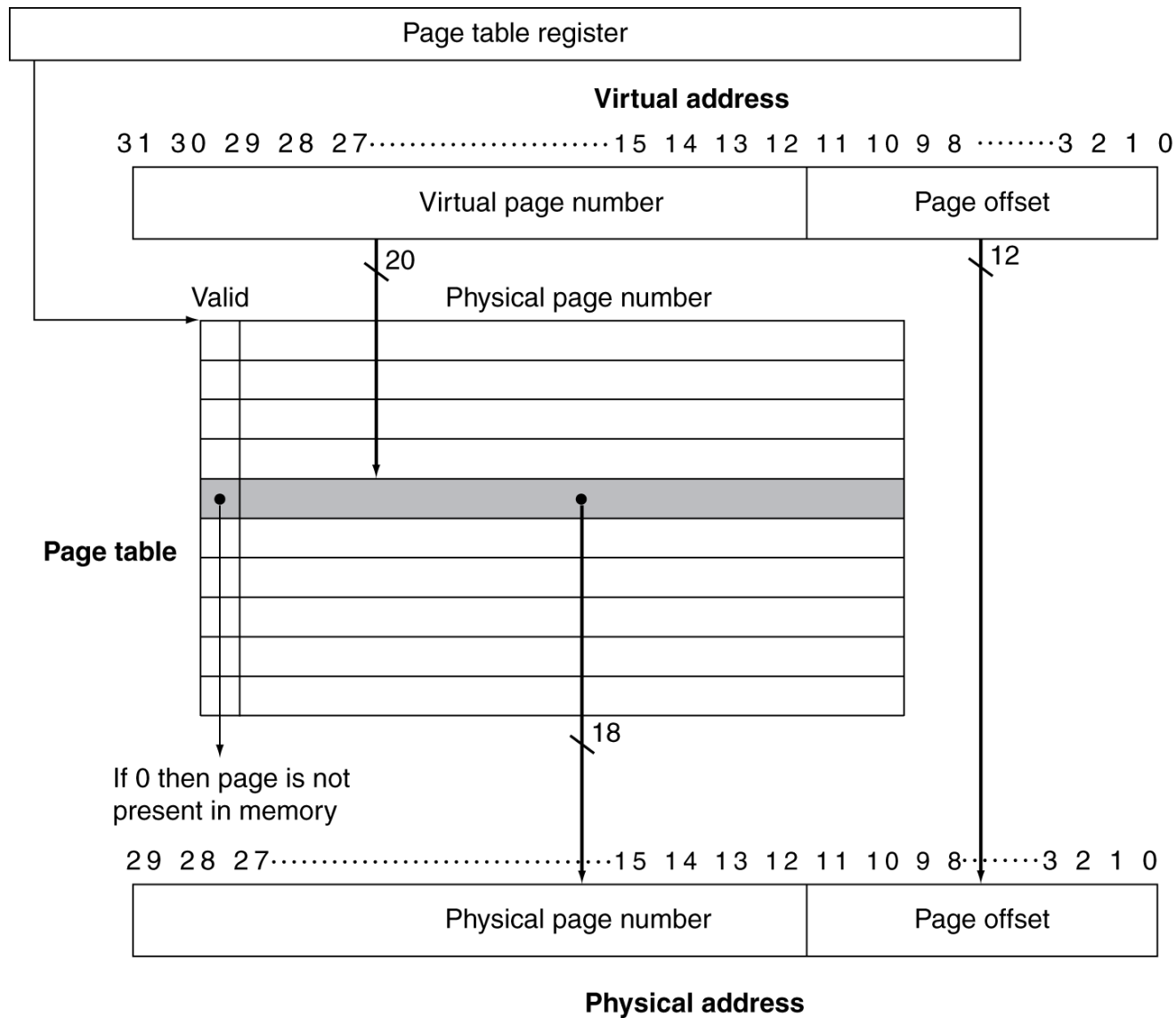
Paging

- Permits the physical address space of a process to be noncontiguous
- Divide **physical memory** into fixed-sized blocks called **frames**
- Divide **logical memory** into blocks of same size called **pages**
 - Page (or frame) size is power of 2 (typically, 512B – 8KB)
 - Mostly use 4K in modern OS
- Set up a **page table** to **translate virtual to physical addresses**

Paging (2)



Translation Using a Page Table



Page Tables

Stores placement information

- Array of page table entries, indexed by virtual page number
- Page table register in CPU points to page table in physical memory

If page is present in memory

- PTE stores the physical page number
- Plus other status bits (referenced, dirty, ...)

If page is not present

- PTE can refer to location in swap space on disk

Mapping Pages to Storage

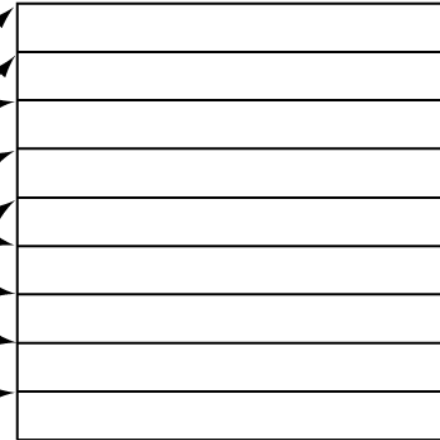
Virtual page
number

Page table

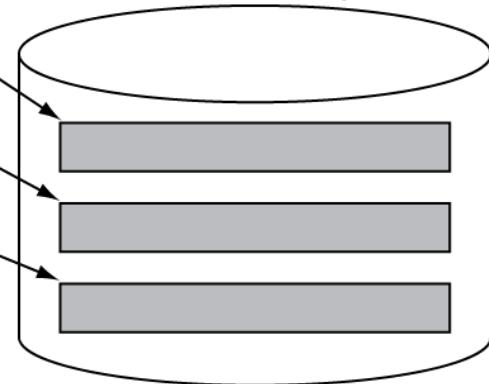
Valid Physical page or
disk address

1	•
1	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•
1	•
0	•
1	•

Physical memory

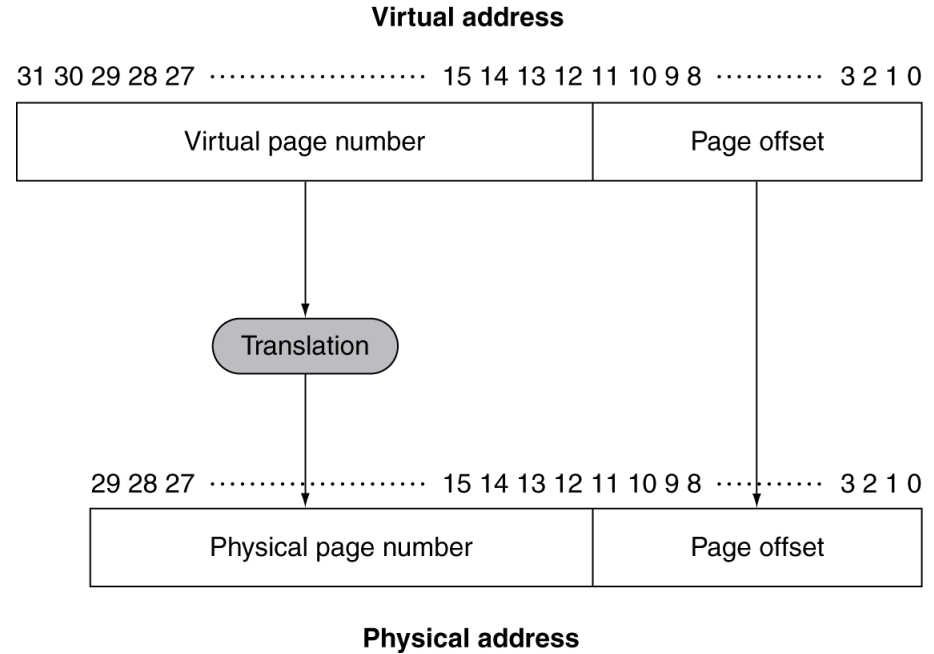
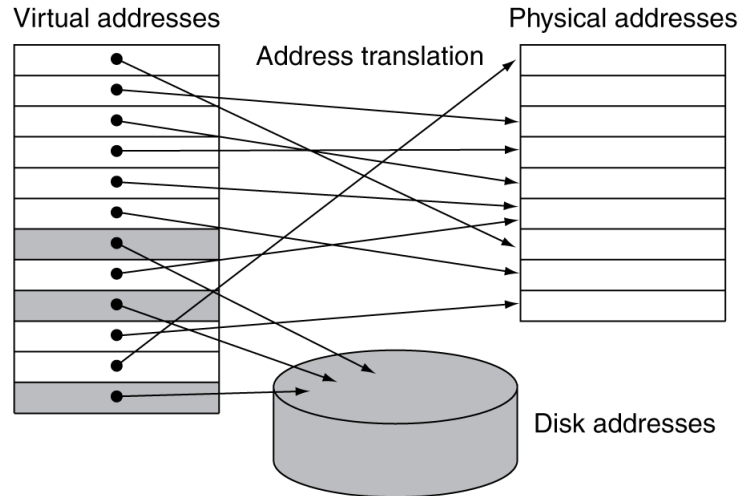


Disk storage



Address Translation

Fixed-size pages (e.g., 4K)



Page Fault Penalty

Page faults

- Referencing a virtual address in an evicted page

On page fault, the page must be fetched from disk

- Takes millions of clock cycles
- Handled by OS code

Try to minimize page fault rate

- Fully associative placement
- Smart replacement algorithms

Page Fault Handler

Use faulting virtual address to find PTE

Locate page on disk

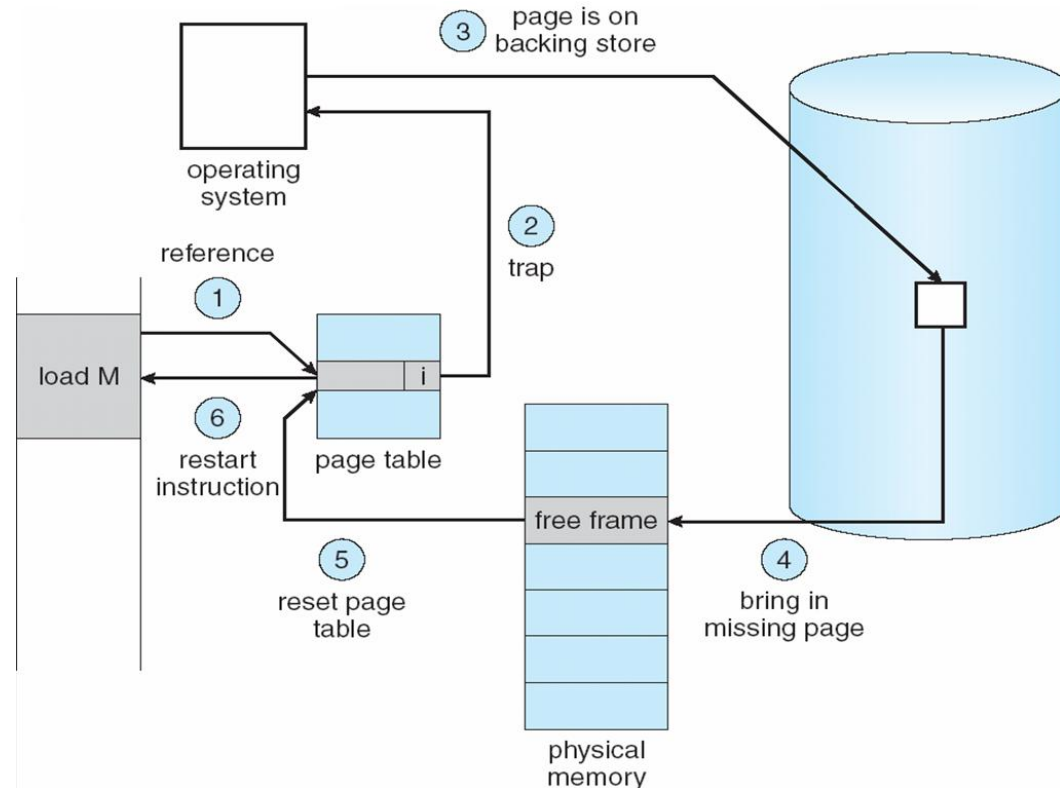
Choose page to replace

- If dirty, write to disk first

Read page into memory and update page table

Make process runnable again

- Restart from faulting instruction



Replacement and Writes

To reduce page fault rate, prefer **least-recently used (LRU)** replacement

- Reference bit (aka use bit) in PTE set to 1 on access to page
- Periodically cleared to 0 by OS
- A page with reference bit = 0 has not been used recently

Disk writes take millions of cycles

- Write through is impractical
- Use write-back

Fast Translation Using a TLB

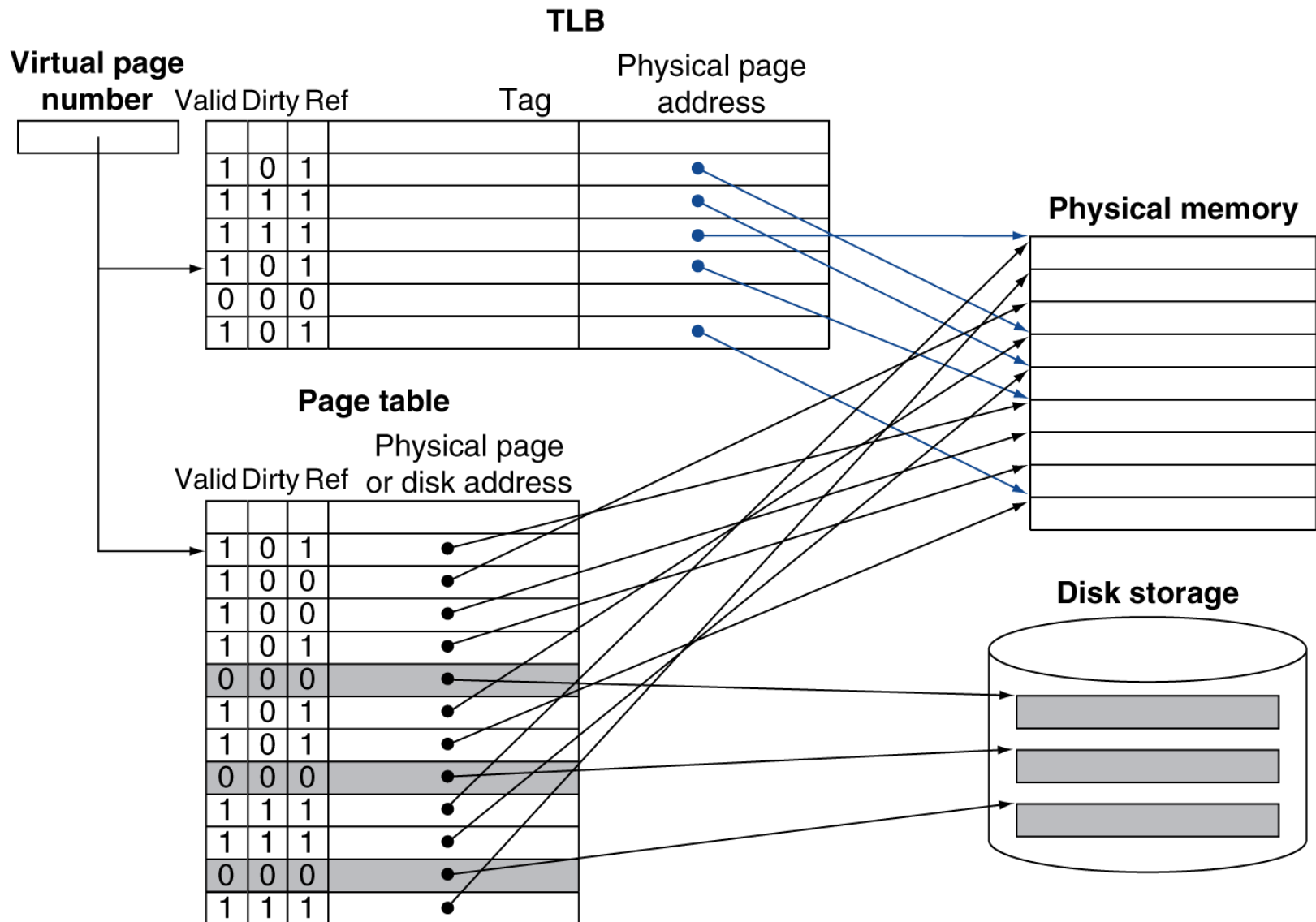
Address translation would appear to require extra memory references

- One to access the PTE
- Then the actual memory access

But access to page tables has good locality

- So use a fast cache of PTEs within the CPU
- Called a Translation Look-aside Buffer (TLB)
- Typical: 16-512 PTEs, 0.5-1 cycle for hit, 10-100 cycles for miss, 0.01%-1% miss rate
- Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB Misses

If page is in memory

- Load the PTE from memory and retry
- Could be handled in hardware
 - Can get complex for more complicated page table structures
- Or in software
 - Raise a special exception, with optimized handler

If page is not in memory (page fault)

- OS handles fetching the page and updating the page table
- Then restart the faulting instruction

TLB Miss Handler

TLB miss indicates

- Page present, but PTE not in TLB
- Page not present

Handler copies PTE from memory to TLB

- Then restarts instruction
- If page not present, page fault will occur

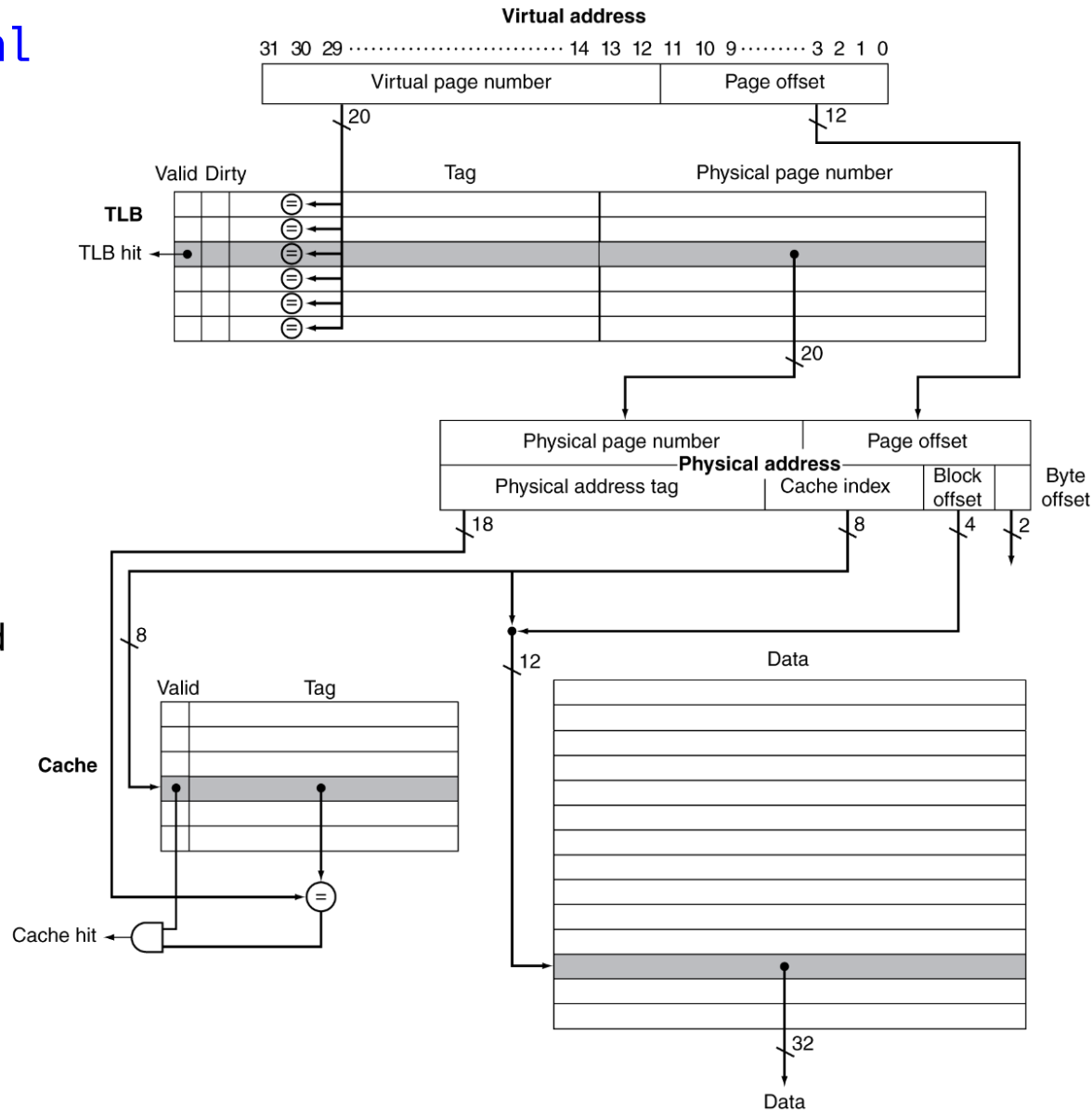
TLB and Cache Interaction

If cache tag uses physical address

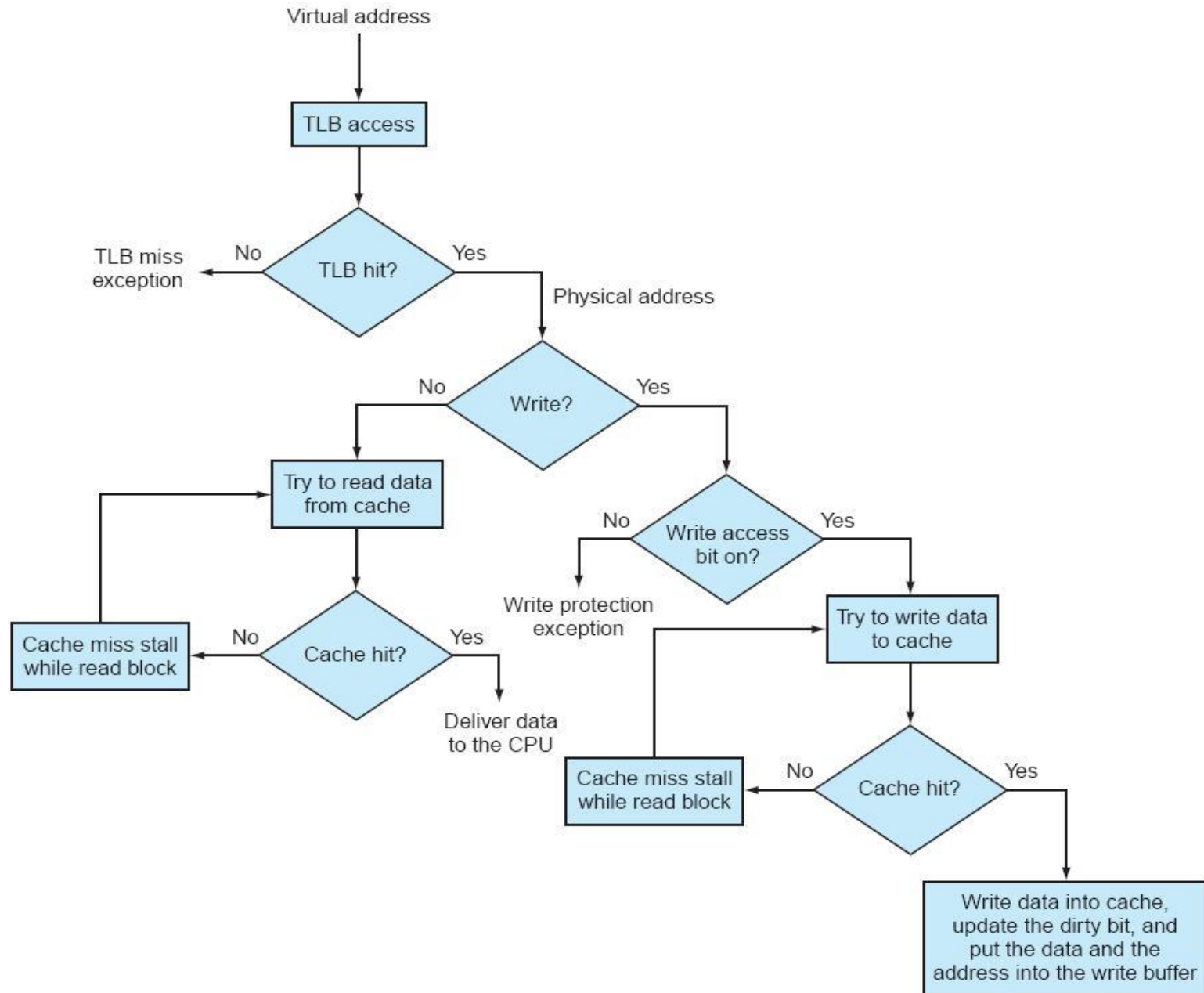
- Need to translate before cache lookup

Alternative: use virtual address tag

- Complications due to aliasing
 - Different virtual addresses for shared physical address



Processing Read/Write



Memory Protection

Different tasks can share parts of their virtual address spaces

- But need to protect against errant access
- Requires OS assistance

Hardware support for OS protection

- Privileged supervisor mode (aka kernel mode)
 - Can perform privileged instructions
- Page tables and other state information only accessible in supervisor (kernel) mode

The Memory Hierarchy

Common principles at all levels of the memory hierarchy

- Based on notions of caching

Considerations at each level in the hierarchy

- Block placement
- Finding a block
- Replacement on a miss
- Write policy

Block Placement

Determined by associativity

- Direct mapped (1-way associative)
 - One choice for placement
- n-way set associative
 - n choices within a set
- Fully associative
 - Any location

Higher associativity reduces miss rate

- Increases complexity, cost

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index Search within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

Hardware caches

- Reduce comparisons to reduce cost

Virtual memory

- Full table lookup makes full associativity feasible
- Benefit in reduced miss rate

Replacement

Choice of entry to replace on a miss

- Least recently used (LRU)
 - Complex and costly hardware for high associativity
- Random
 - Close to LRU, easier to implement

Virtual memory

- LRU approximation with hardware support

Write Policy

Write-through

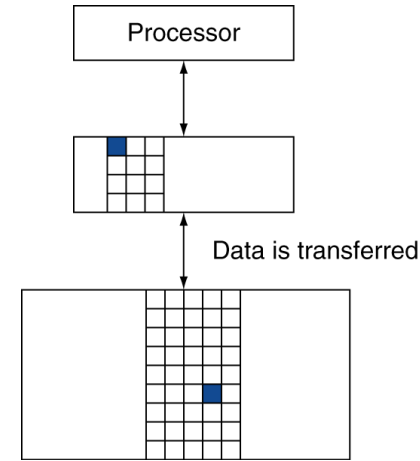
- Update both upper and lower levels
- Simplifies replacement, but may require write buffer

Write-back

- Update upper level only
- Update lower level when block is replaced
- Need to keep more state

Virtual memory

- Only write-back is feasible, given disk write latency



Sources of Misses

Compulsory misses (aka cold start misses)

- First access to a block

Capacity misses

- Due to finite cache size
- A replaced block is later accessed again

Conflict misses (aka collision misses)

- In a non-fully associative cache
- Due to competition for entries in a set
- Would not occur in a fully associative cache of the same total size

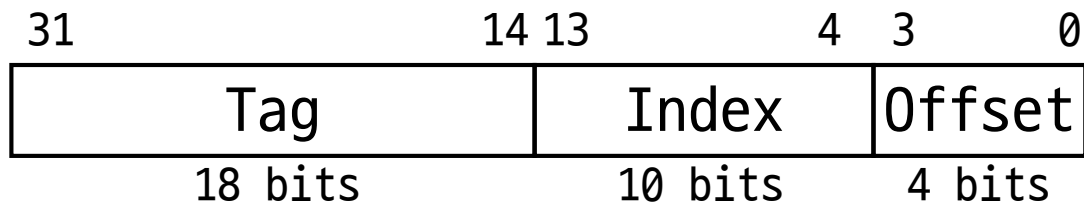
Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

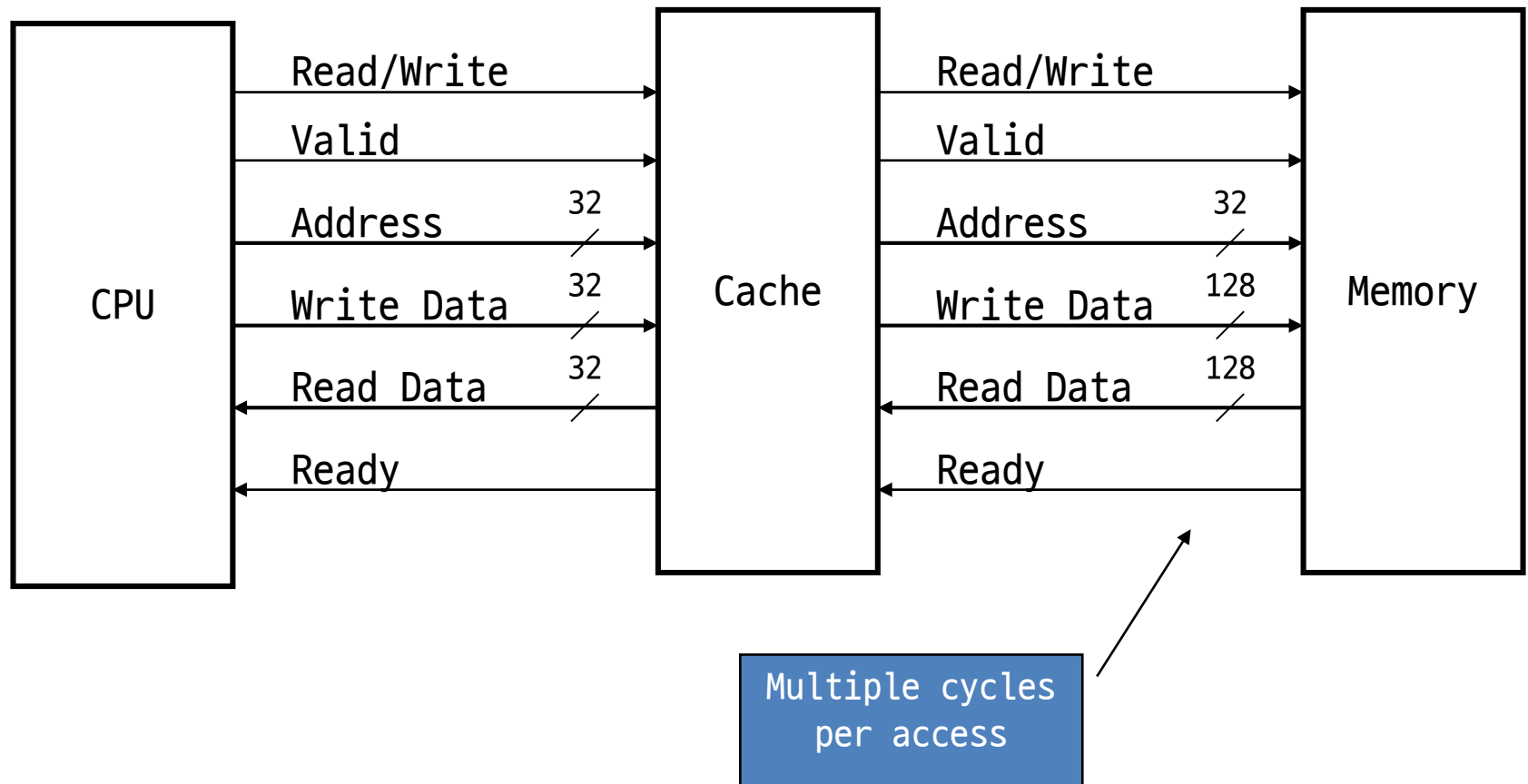
Cache Control

Example cache characteristics

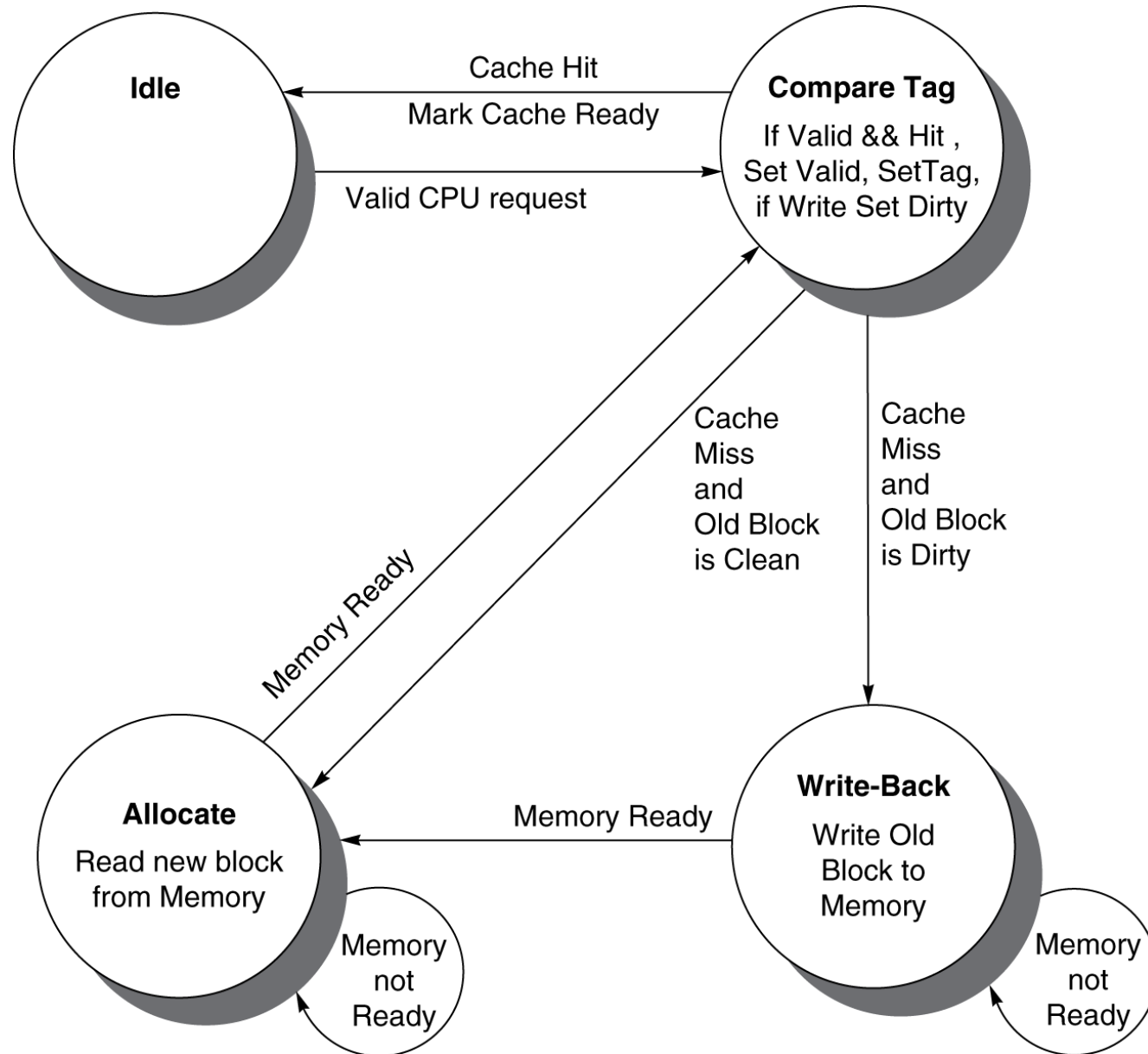
- Direct-mapped, write-back, write allocate
- Block size: 4 words (16 bytes)
- Cache size: 16 KB (1024 blocks)
- 32-bit byte addresses
- Valid bit and dirty bit per block
- Blocking cache
 - CPU waits until access is complete



Interface Signals



Cache Controller FSM



Cache Coherence Problem

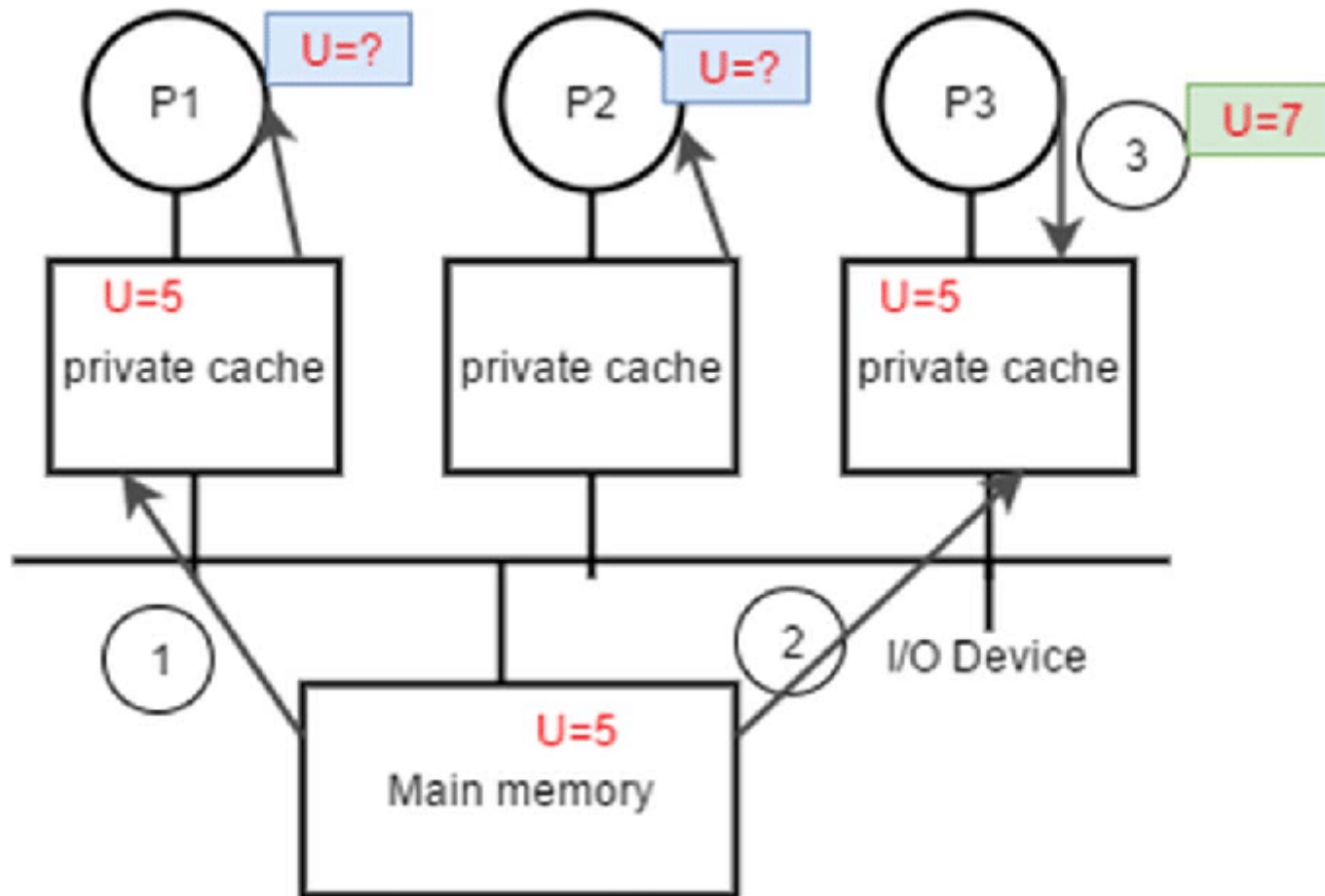
Suppose two CPU cores share a physical address space

- Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

e.g.) Multi-core processor, Multi-core multiprocessor

Cache Coherence Problem



Coherence Defined

Informally: Reads return most recently written value

Formally:

- P_1 writes X ; P_1 reads X (no intervening writes)
⇒ read returns written value
- P_1 writes X ; P_2 reads X (sufficiently later)
⇒ read returns written value
- P_1 writes X ; P_2 writes X
⇒ all processors see writes in the same order
 - End up with the same final value for X

Cache Coherence Protocols

- Snooping protocols
 - Each cache monitors bus reads/writes
- Directory-based protocols
 - Caches and memory record sharing status of blocks in a directory

Snooping Protocols

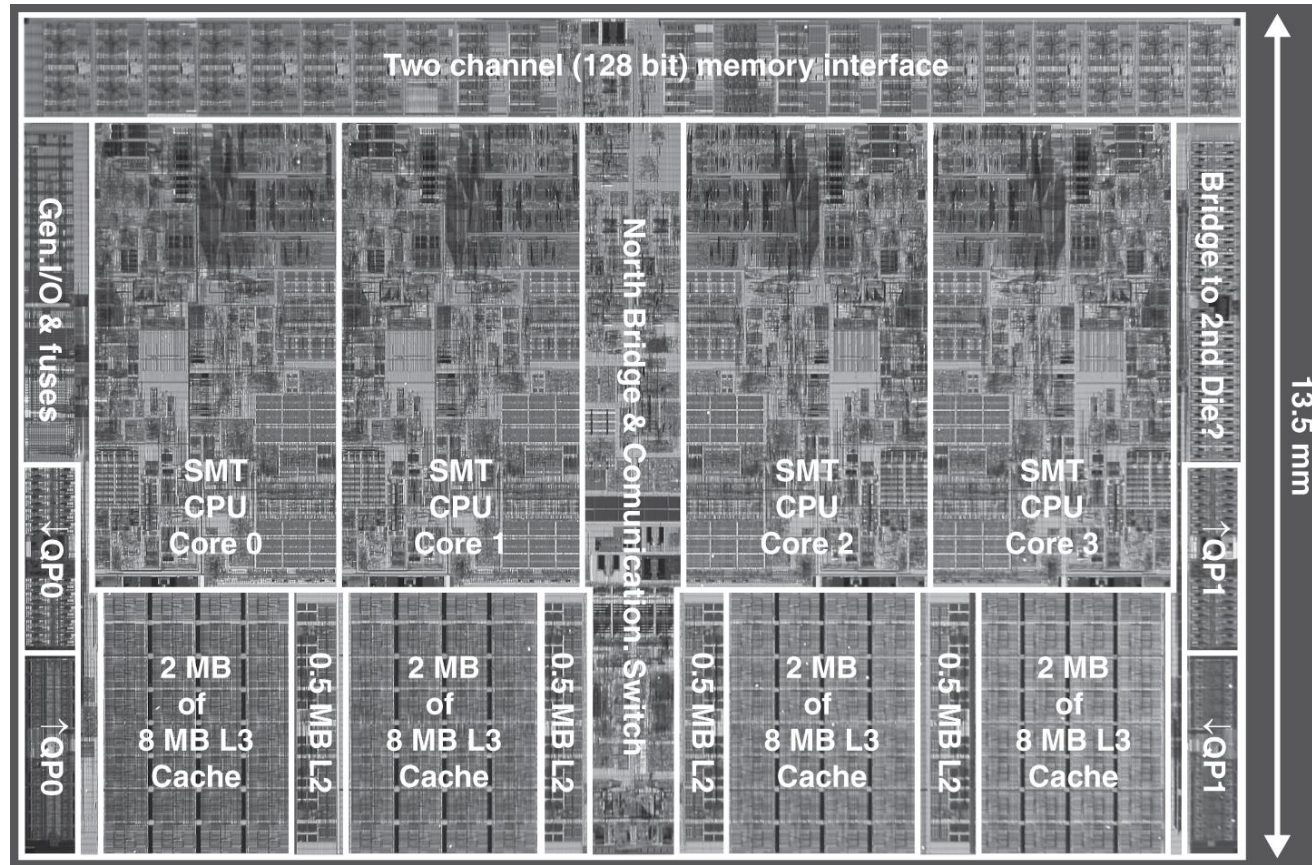
Cache gets exclusive access to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read in another cache misses
 - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		1
CPU B read X	Cache miss for X	1	1	1

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Concluding Remarks

Fast memories are small, large memories are slow

- We really want fast, large memories ☹️
- Caching gives this illusion 😊

Principle of locality

- Programs use a small part of their memory space frequently

Memory hierarchy

- L1 cache ↔ L2 cache ↔ ... ↔ DRAM memory ↔ ↔ disk

Memory system design is critical for multiprocessors