CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Translation Look-Aside Buffer

Need to consider implementation of page table

Each process requires a page table

When context switching, current page table must be stored to memory and next page table loaded

For small systems, the current page table can be stored as set of registers

Lookups are very fast, but context switch times increase because all registers must be reloaded from memory

For larger page tables, we do not have enough registers to do this

Another way is to leave current page table in memory and use **page-table base register** (PTBR) to point to page table

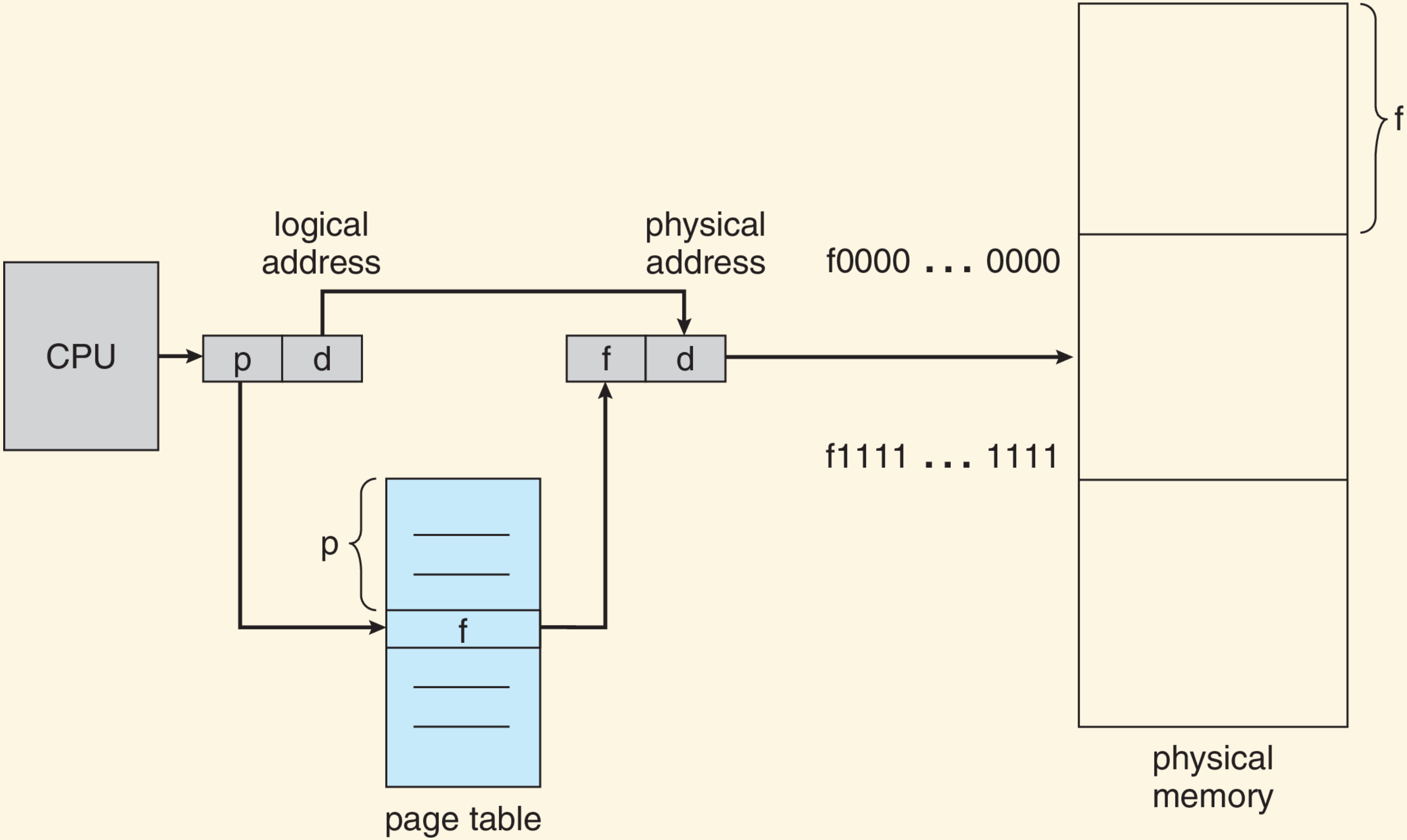
Page table portion of context switches are very efficient -- simply changing one register

However, severely decreases speed of lookups

When page table is in memory, one memory access is required simply to find the frame number

Only then is the physical address known, and a second memory access is performed to get the desired memory

Every memory access by a program now requires two actual memory accesses


logical
address
physical
address
CPU
p
d
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
page table
physical
memory

Page table is used *for every memory access*, so we have effectively cut the speed of memory access in half, which is unacceptable

What do we do when memory access it to slow?

(Hint: you've seen this before)

Caching!

Translation look-aside buffer (TLB) is small, fast-lookup hardware cache specifically for page table entries

TLB stores `(page #, frame #)` pairs

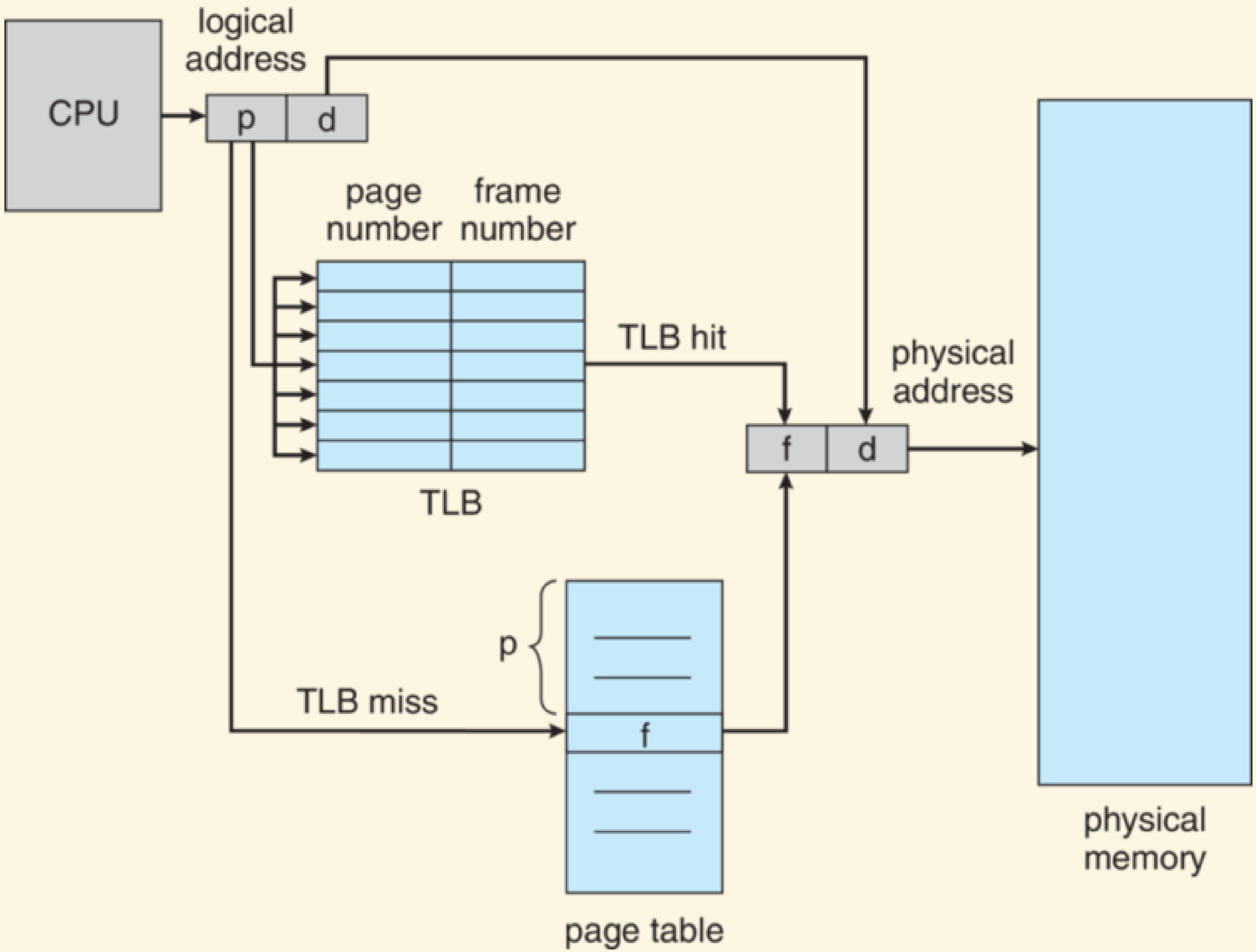
When logical address is requested, if corresponding page number is in TLB, frame number is returned very quickly

TLB is fully associative, so all entries are checked for match at the same time

If matching page number is found in TLB, frame number is returned with essentially no time penalty

Otherwise, access is a **TLB miss** and requires full trip to memory

TLB must be small to be efficient: often just 32 - 1024 entries



logical address
CPU
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

On TLB miss, offending `(page number, frame number)` pair is added to TLB

When TLB is full, adding new pair requires removing old pair

This can be done with any of the replacement policies you learned in 351

TLB, like any cache, is purely for performance -- it does not result of page table lookup or memory access

To measure performance improvement, we will assume memory access costs 100 ns

Without TLB, every memory access generated by program results in two memory accesses (one for page table to generate physical address, followed by memory access at desired address)

Effective memory-access time without TLB is 200 ns

TLB performance depends heavily on **hit ratio** -- proportion of lookups that are satisfied by TLB

With 80% hit ratio, 80% of accesses take just 100 ns (no lookup to page table in memory) and 20% of accesses take 200 ns (one lookup to page table, another to find desired physical address)

Effective memory access time is 0.80 * 100 + 0.20 * 200 = 120 ns

This is already a large improvement over non-TLB system

In reality, TLB will be successful far more than 80% of the time

According to your textbook, 99% hit ratio is much more realistic

Effective access time is $0.99 * 100 + 0.01 * 200 = 101$ ns

This is barely more than if we had used registers instead of memory to store page table

As described, TLB must be emptied (**flushed**) every context switch

If process A and process B both try to access logical address `0x12345678`, they should end up with different physical addresses

However, if TLB is not flushed, process B will always have same frame number returned from TLB as process A

To avoid this, TLB can store additional **address-space identifiers** (ASIDs)

ASID keeps track of which process stored `(page number, frame number)` relationship belongs to

Allows context switches without flushing buffer because TLB can track which relationships go with which processes

Much like caches for main memory, modern TLBs are typically more complicated (e.g., multilevel)

- Each process has a page table
- Page table for current process must be stored somewhere; typically in memory for modern, large page tables
- Doubling access time with in-memory page table is too slow
- TLB is small, fast cache that greatly improves performance, but has no effect on result of page-table lookup