CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Structure of Page Tables

Page tables are stored in memory because we do not have enough registers to store an entire table

Even in memory, storing large page tables can be prohibitively expensive

Most modern computers are 32- or 64-bit address architectures

The logical address space is then (2^32) to (2^64) bits

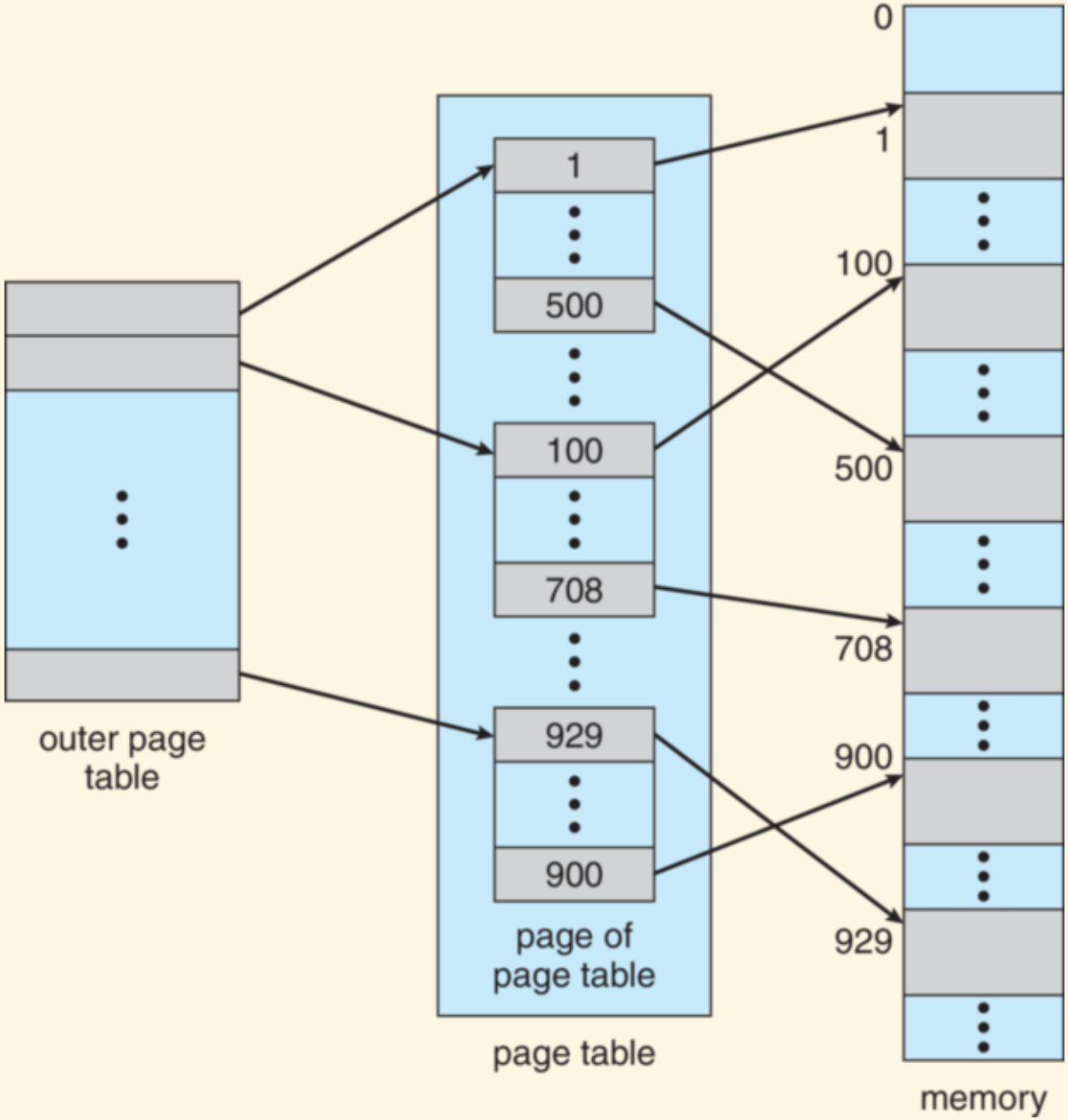
In a 32-bit system with page size of 4 KB (2^12), page table needs (2^32/2^12 = 2^20) entries, which is more than 1 million

Assuming each entry is 4 bytes, the page table *for each process* could require up to 4 MB of physical memory

Note that, until now, the page table had been assumed to be stored contiguously in memory

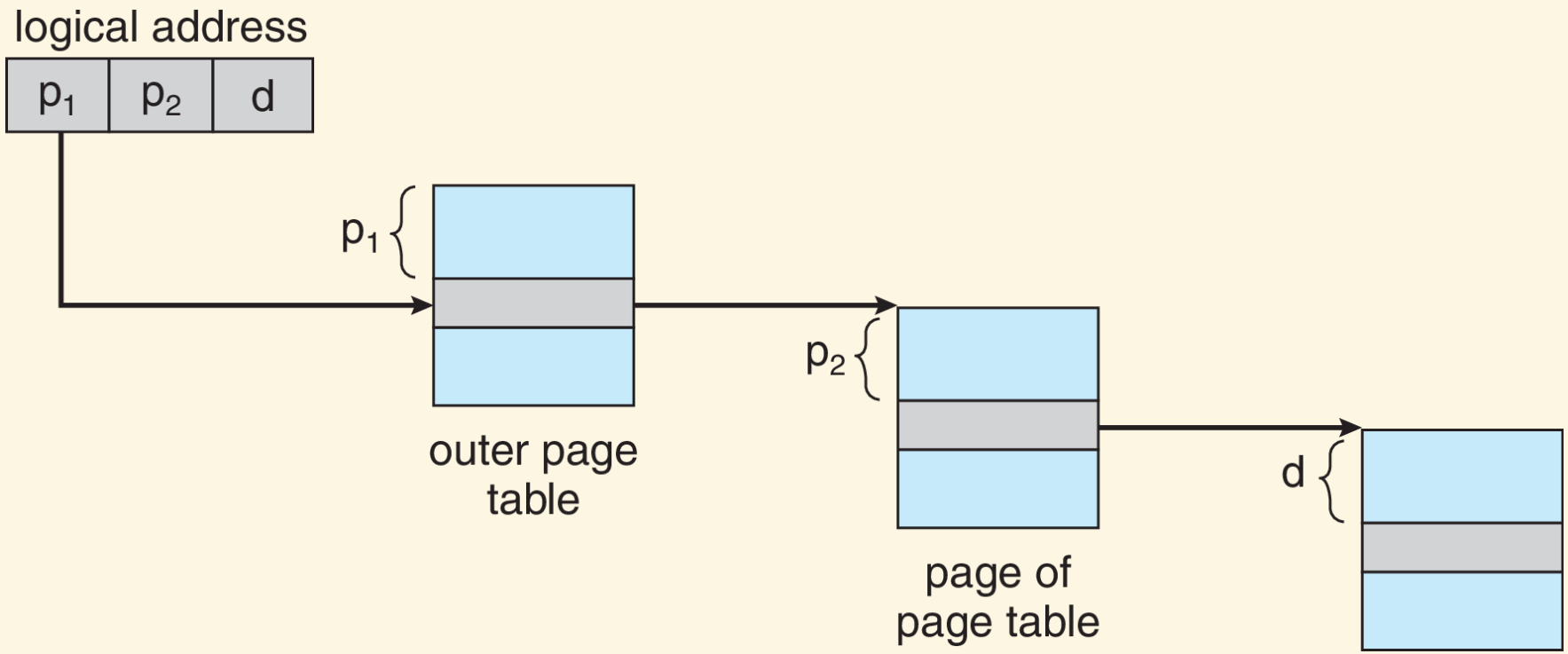
Requiring 4 MB of physical memory is bad enough, but to keep the page tables contiguously in memory would be extremely inconvenient

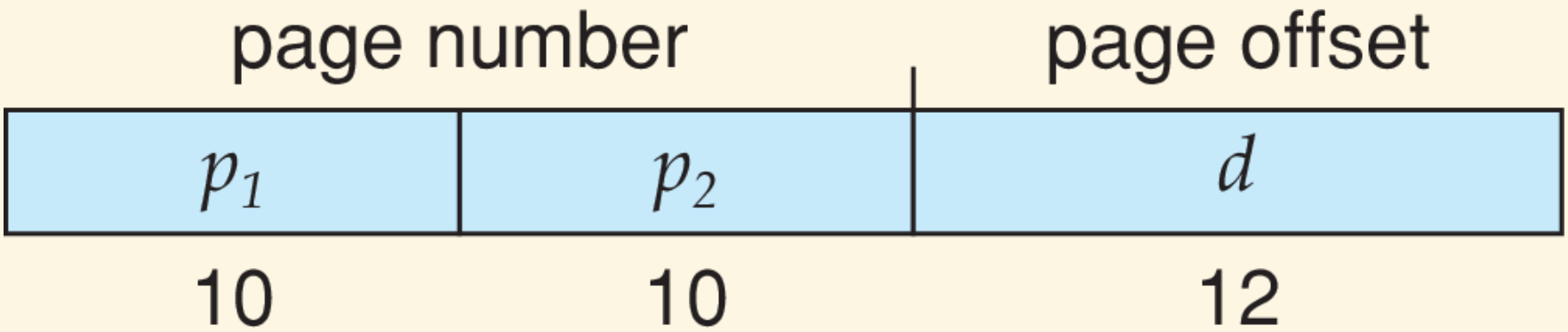
We can instead page the page table


0
1
100
500
708
900
929
outer page table
page of page table
page table
memory

"Page number" portion of address is then split into two parts

First part now represents which "page of page table" is requested


logical address
p1
p2
d
p1
outer page table
p2
page of page table
d

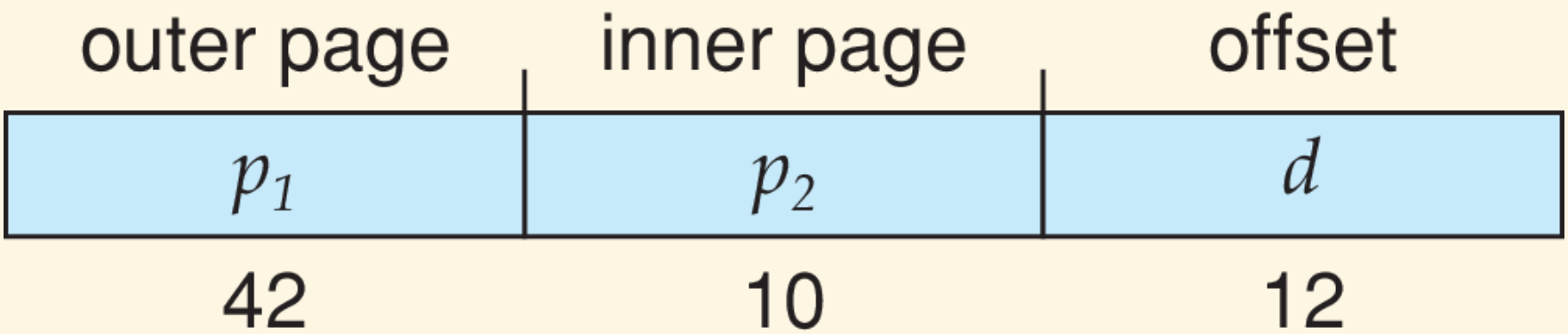

page number
page offset
p_1
p_2
d
10
10
12

This is an example of **hierarchical paging** known as a **forward-mapped page table**

We need only store page table entries for pages that are mapped to physical frames

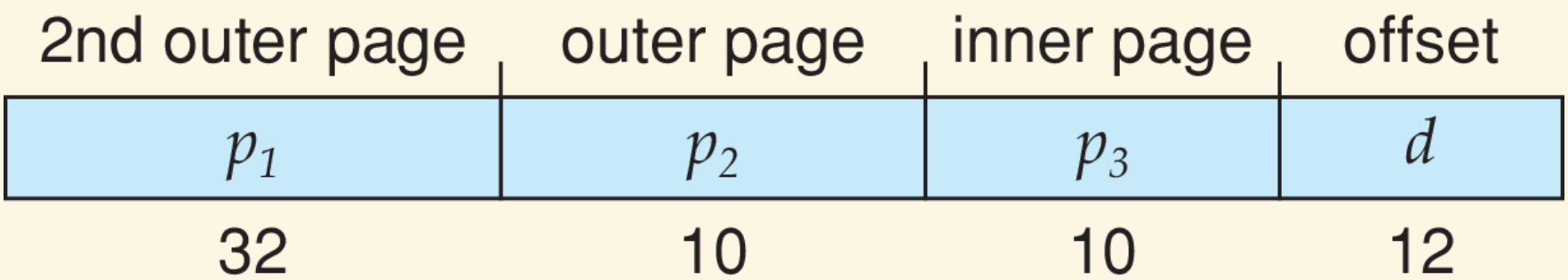
Large portion of virtual address space may go unused, so this could be substantial memory savings

For a 64-bit machine, this will not break the page table into small enough chunks to be manageable


outer page
inner page
offset
p_1
p_2
d
42
10
12

Outer page is much too large

What if we repeat the same idea?


2nd outer page
outer page
inner page
offset
p_1
p_2
p_3
d
32
10
10
12

Even so, outer page table is still 16 GB

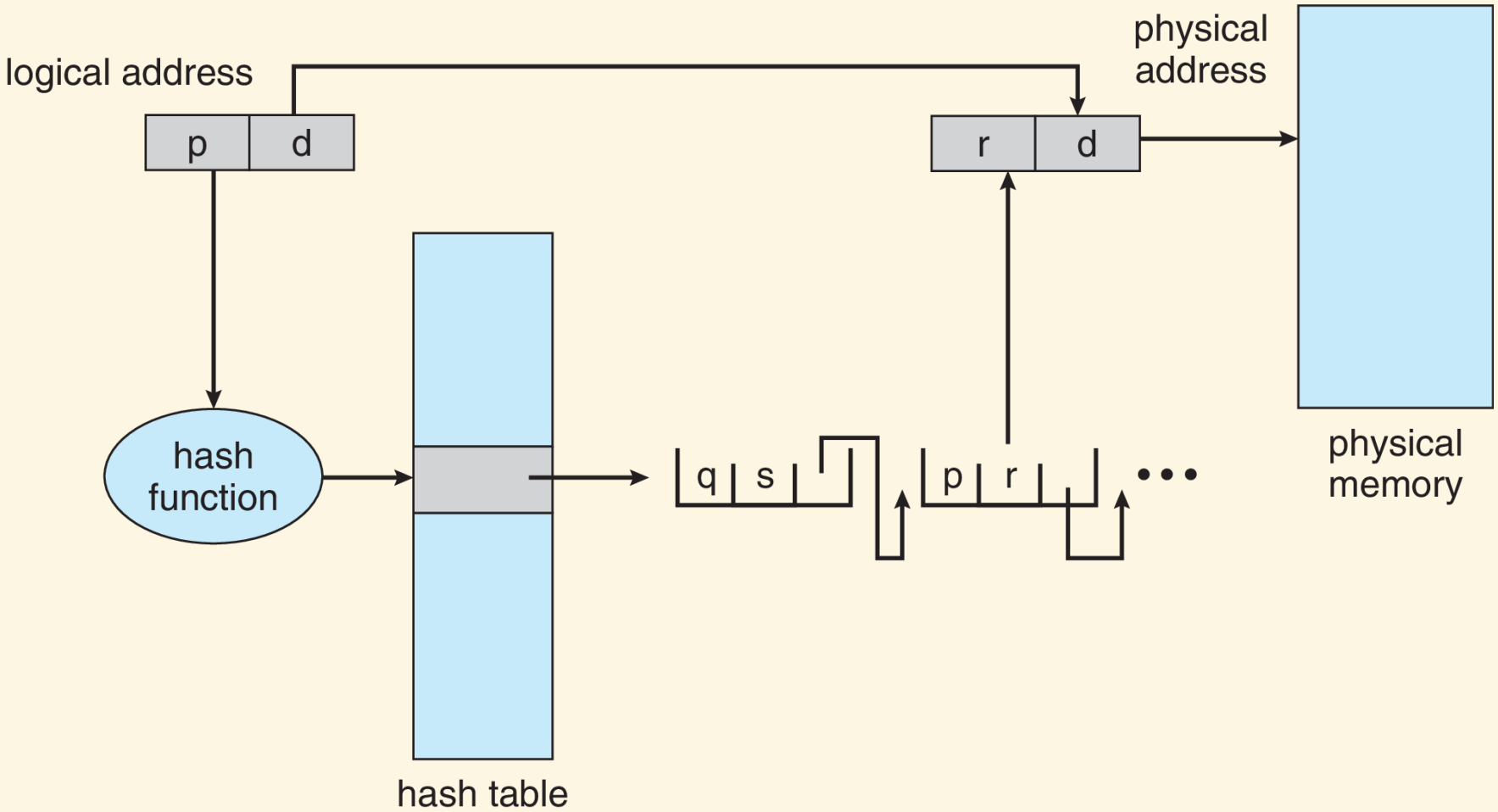
Could keep going, but the number of required memory accesses goes up each time

For larger address spaces, hashed page table is a better choice

Hashed page table is hash table whose key is page number

Values are linked lists of elements, where element contains

- page number
- corresponding frame number
- pointer to next element


logical address
p
d
physical address
r
d
hash function
hash table
q
s
p
r
physical memory

In this setup, only need to store page table entries for pages that are mapped to physical frames

Large portion of virtual address space may go unused, so this could be substantial memory savings

One downside to hashed page tables is that each process still has its own page table

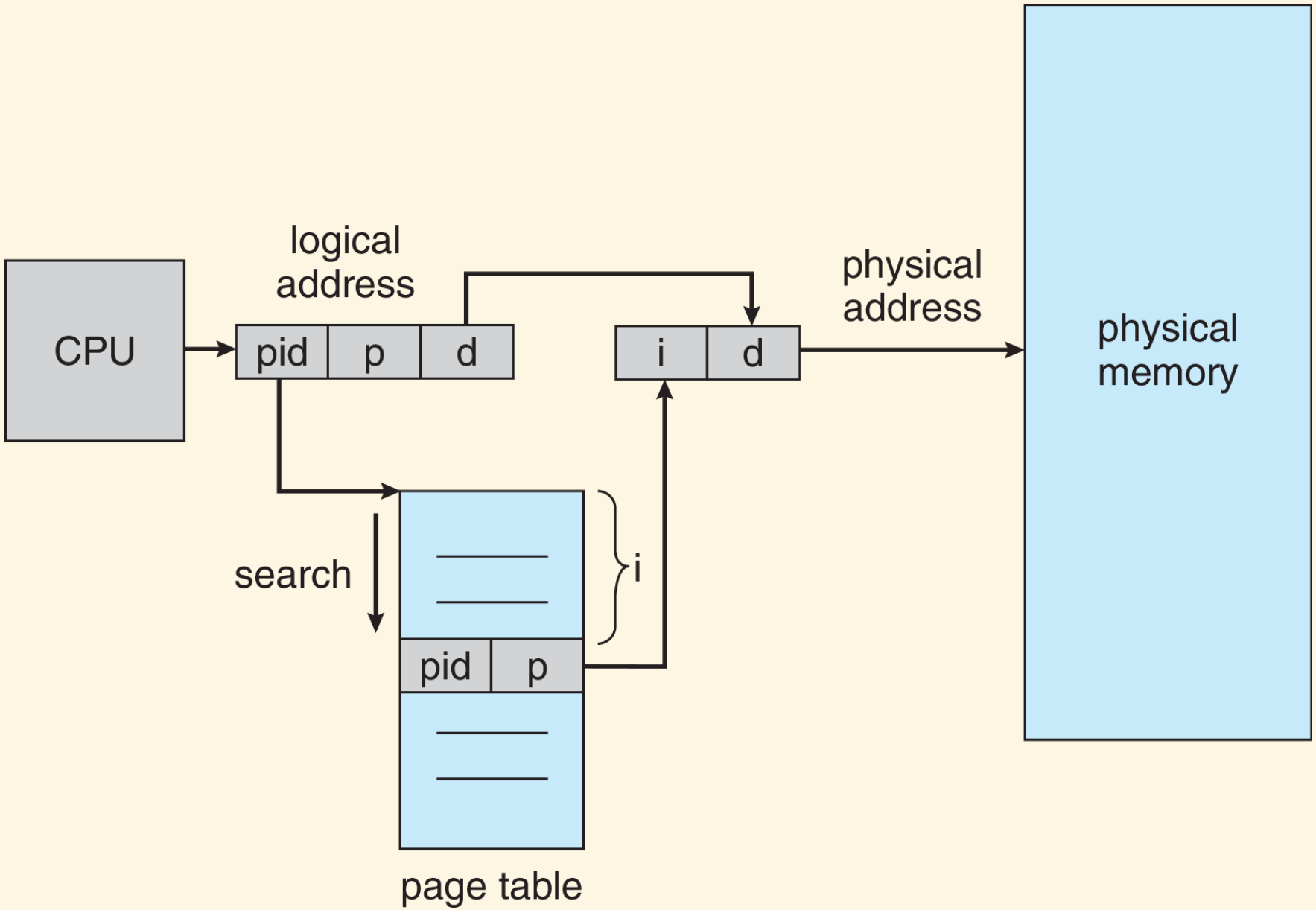
There may be many processes, but there are only so many frames of memory

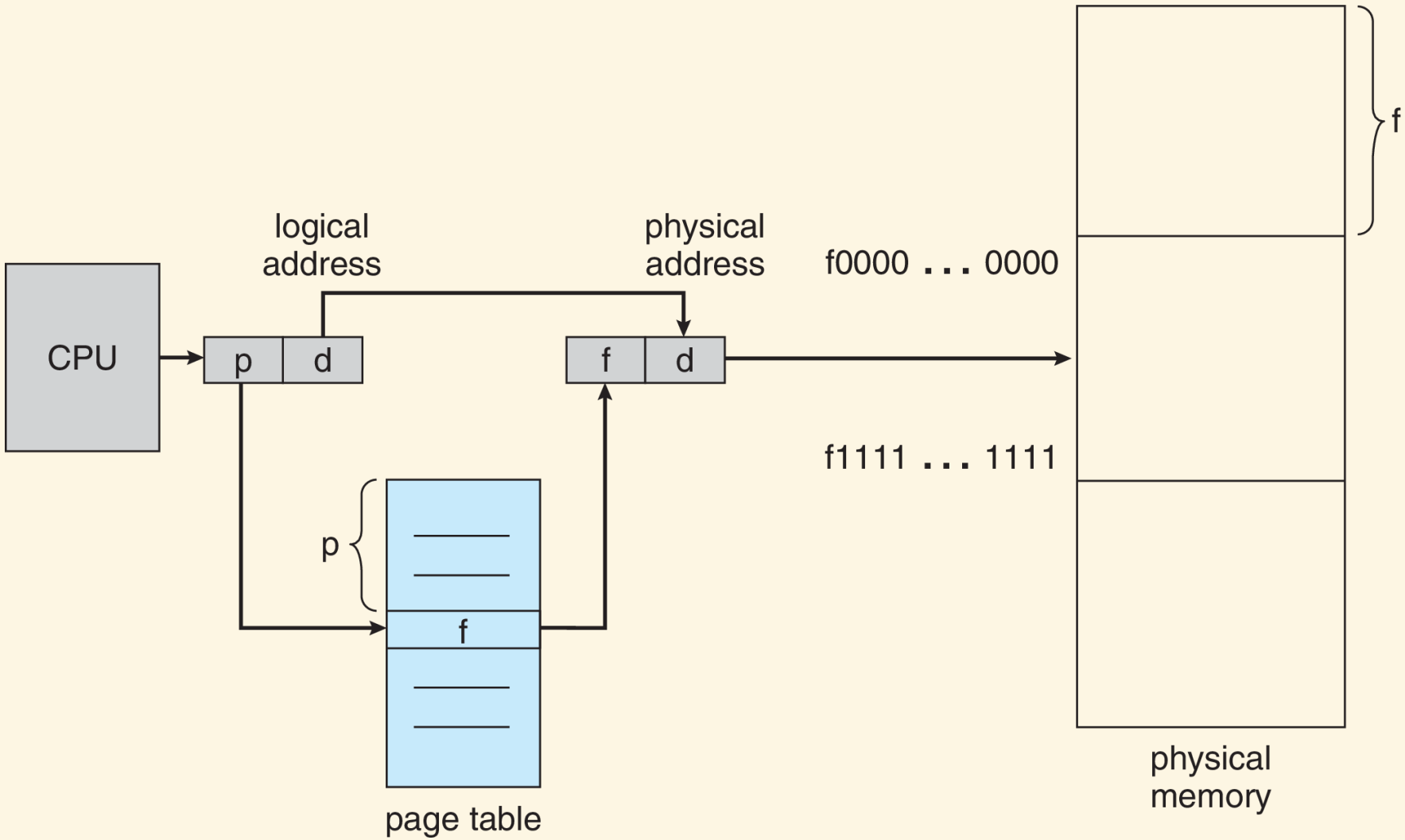
Inverted page table stores just one entry per frame

Information is kept about corresponding virtual address for each frame

Only need one inverted page table system-wide, as opposed to one per process

Because of this, entries of table must also track which process owns each frame


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table


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

This is great for saving memory, but it does not match how addresses are generated

CPU generates logical addresses which must then be mapped to physical addresses

Storing table in order of logical addresses makes lookup easy, but storing by physical address gives us no clue where to look

Searching every table entry for match would not be feasible

Instead, use hash table again

Map logical address to linked list and search list until a match for **both** PID and page number is found

Difference from previous hash setup is that we now store each frame address just once, so there is one table per system, which is why storing PID is also necessary

Small memory footprint and much faster than searching entire table, though still requires more than one memory access

Note that regardless of which scheme is used, a TLB should still be used for efficiency

There are four basic ways to store page tables in memory

- direct storage
- hierarchical page tables
- hashed page tables
- inverted page table

These have tradeoffs between speed and memory footprint