OPERATING SYSTEMS

Memory Management - 6

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 3

UNIT 3: Memory Management

Main Memory: Hardware and control structures, OS support, Address translation, Swapping, Memory Allocation (Partitioning, relocation), Fragmentation, Segmentation, Paging, TLBs context switches .Virtual Memory - Demand Paging, Copy-on-Write, Page replacement policy - LRU (in comparison with FIFO & Optimal), Thrashing, design alternatives - inverted page tables, bigger pages. Case Study: Linux/Windows Memory.

Course Outline - Unit 3

25	8.1	Main Memory: Hardware and control structures, OS support, Address translation,	8	64.2
26	8.2-8.3	Swapping, Memory Allocation (Partitioning, relocation), Fragmentation,	8	
27	8.4	Segmentation	8	
28	8.5	Paging	8	
29	8.5	TLBs context switches	8	
30	8.6	Structure of page tables	8	
31	8.6.3,8.7	design alternatives - Inverted page tables, bigger pages	8	
32	9.1-9.2	Virtual Memory - Demand Paging	9	
33	9.3,9.4.1-9.4.3	Copy-on-Write, Page replacement: Basic page replacement (FIFO page replacement and optimal page replacement)	9	
34	9.4.4, 9.5	LRU Page replacement, Allocation of frames	9	
35	9.6	Thrashing	9	
36	9.10	Case Study: Linux/Windows Memory	9	

Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Page Table Structure

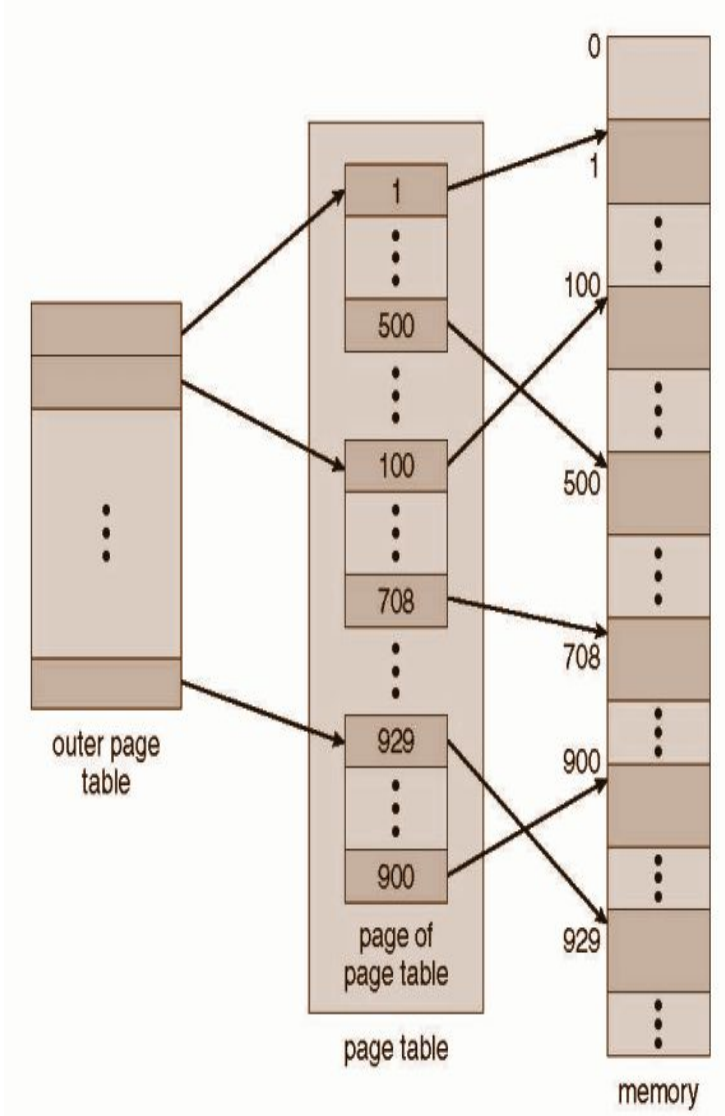
- Memory structures for paging can get huge using regular methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space or memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory

Hierarchical Page Table – Two Level Page table

- Break up the logical address space into multiple page tables

- A simple technique is a two-level page table

- We then page the page table



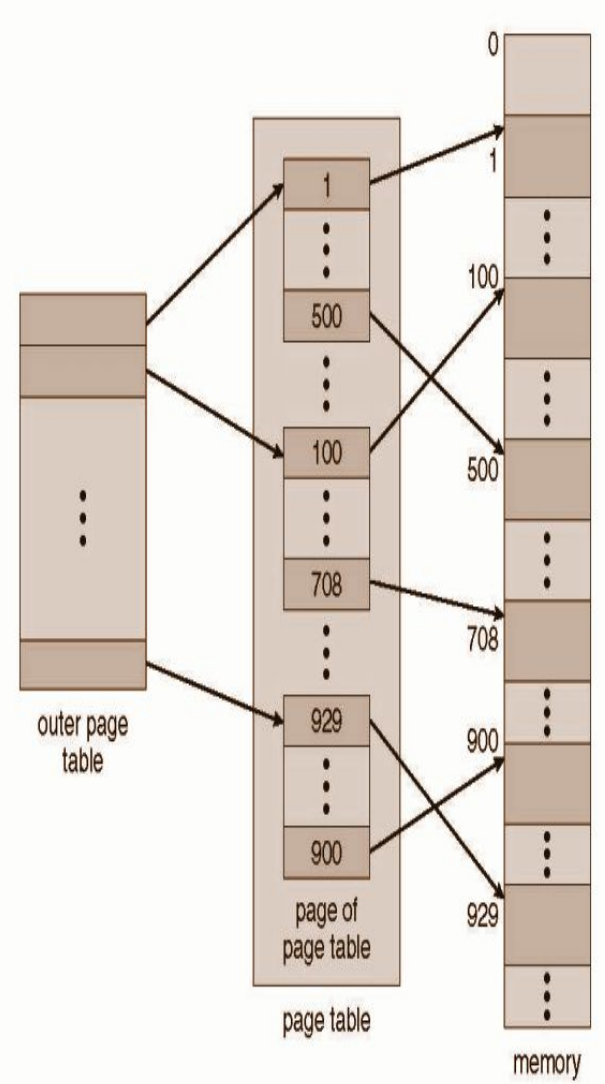
Two Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits

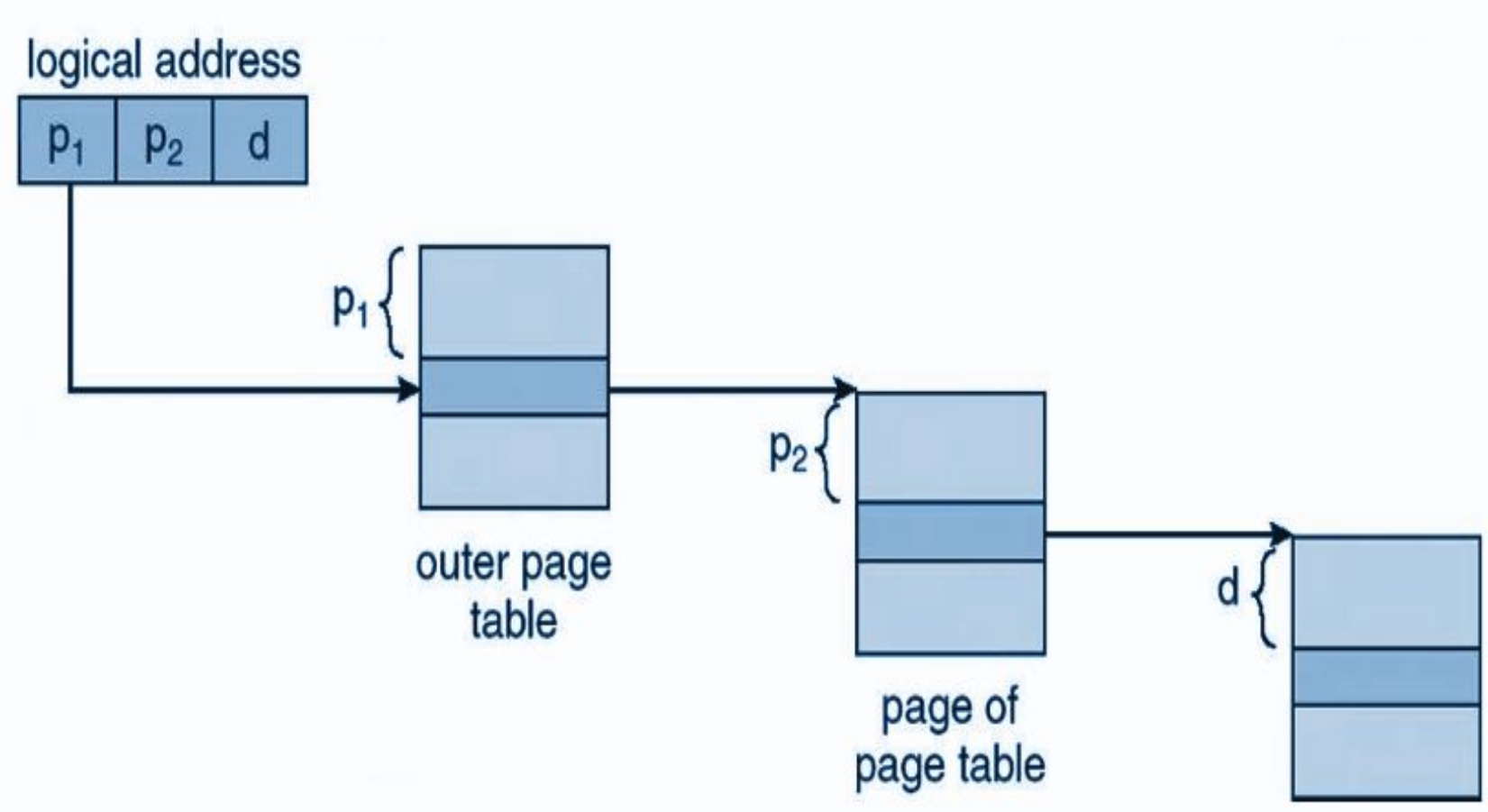
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset

- It is also Known as forward-mapped page table

- where p1 is an index into the outer page table, and p2 is the displacement within the page of the inner page table



Two level Address Translation Scheme



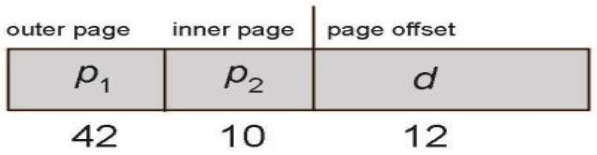
Two level Address Translation Scheme - For a 32 bit Logical address space and page size of 1024 bytes

32 bit Logical address space - 2^{32}	
2^{22} bits for Page Number	2^{10} bits for Page offset

P1=> 2^{12} bits for Outer Page Table	P2=> 2^{10} bits for displacement within the Inner page table	2^{10} bits for Page offset

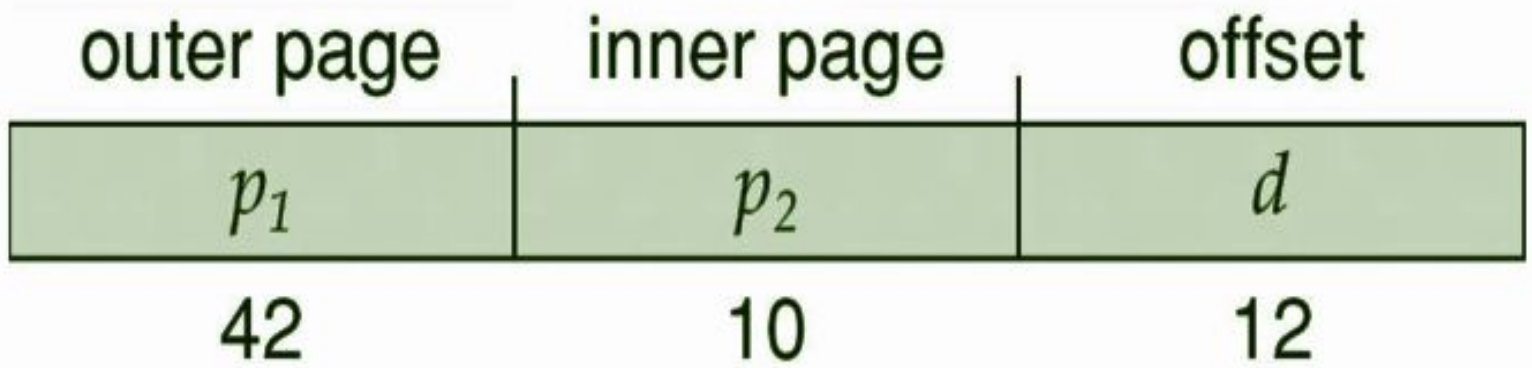
Two Level Paging Example - 64 Bit Logical Address Space

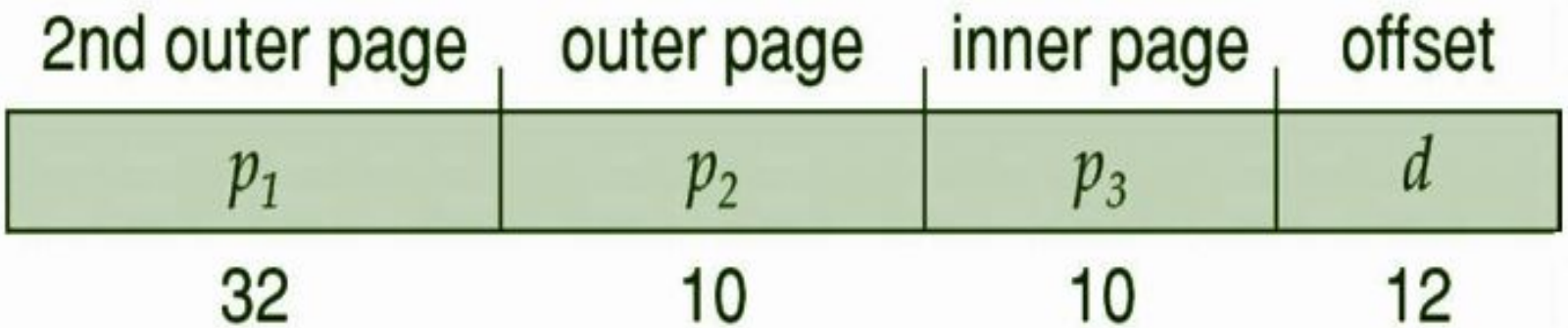
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
- Address would look like



- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
  - And possibly 4 memory access to get to one physical memory location

Three Level Paging Scheme

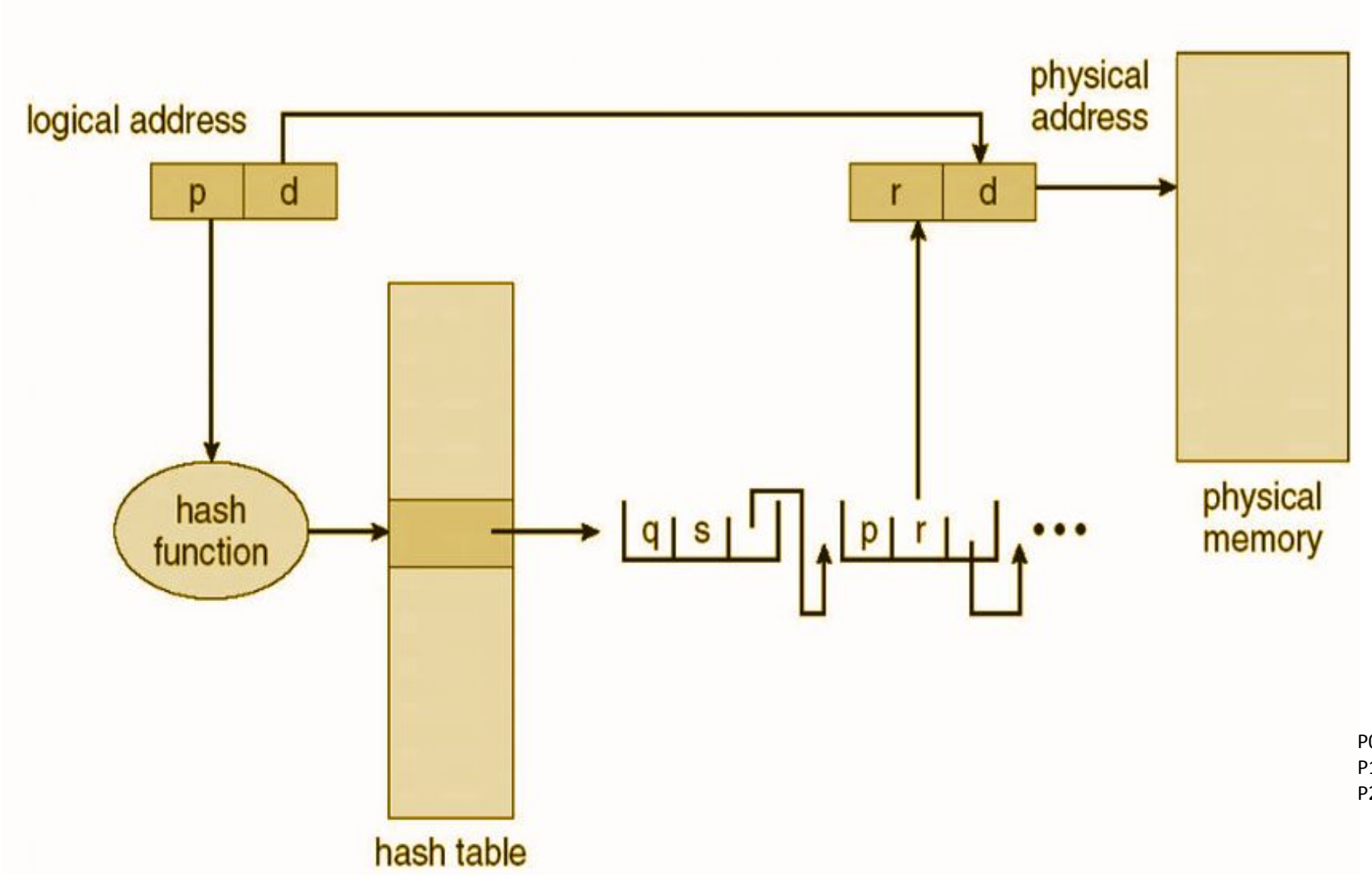




Hashed Page Table

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - (1) the virtual page number
  - (2) the value of the mapped page frame
  - (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is clustered page tables
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table Architecture



P0=10

P1=14

P2=18

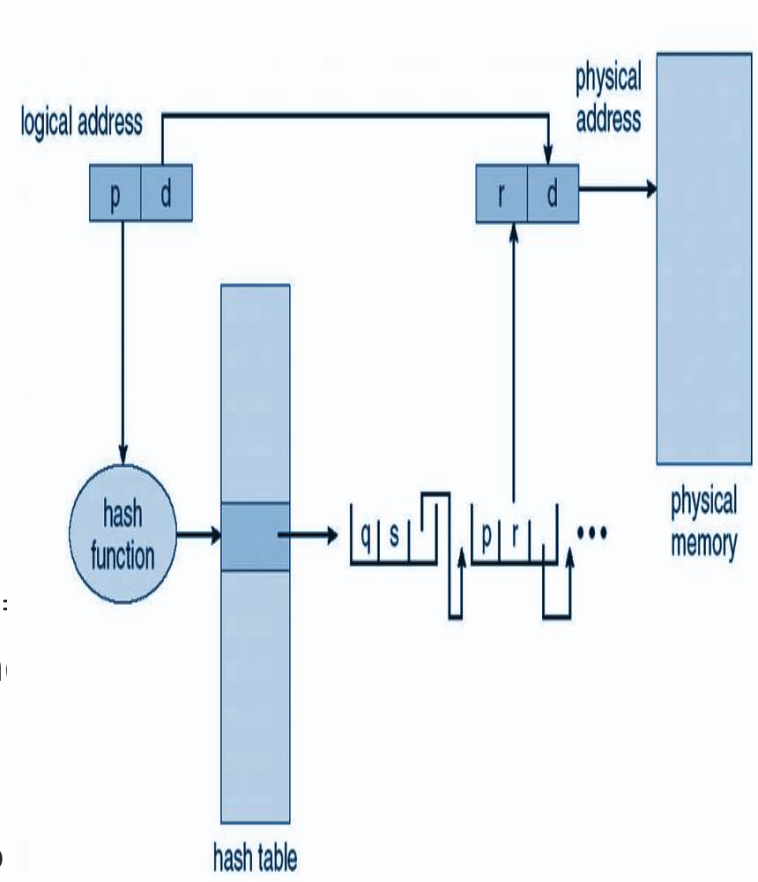
Hash Table =4096

P0 =>I want Page 5
P1=> I also want Page 5
P2 => I also want Page 5

Hashed Page Table - Example

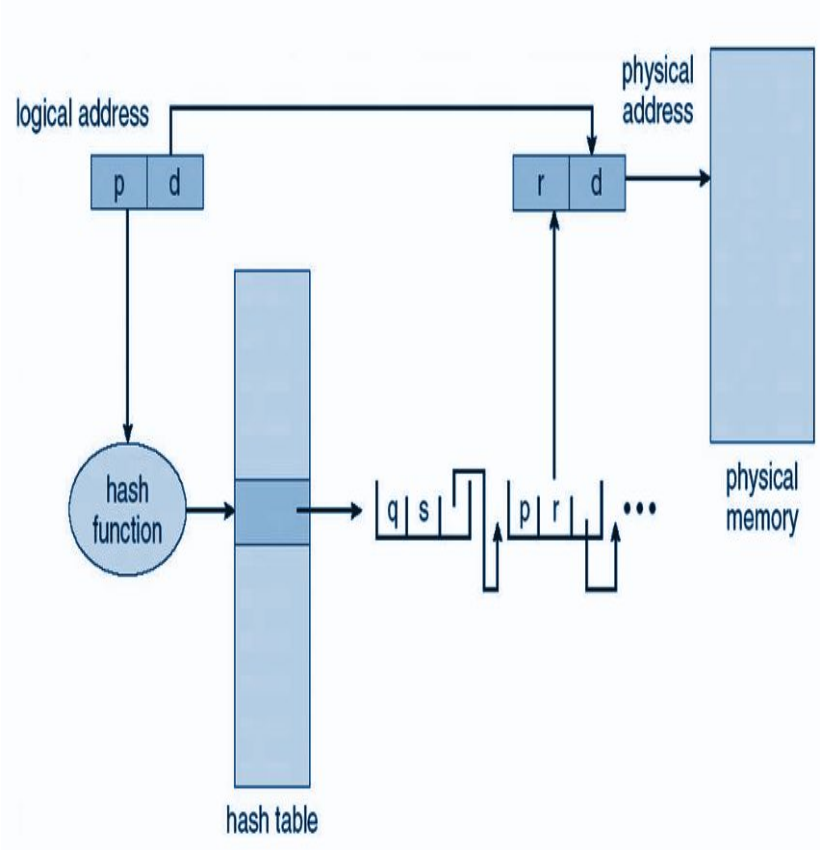
In the diagram, we have the following components:

- Virtual Page Number (VPN): p, q
- Page Frame Number (PFN): r
- Offset: d
- Hash Function: h(x)
- Hashed Page Table with schema (key, VPN, PFN, Pointer to next entry with key) for each entry in the table
- It so happens that h(p) = same_key and h(q) = same_key. There is hash collision. Both p and q are hashed to the same_key.
- This is resolved by chaining the entry with VPN q to the entry with VPN = p. Chaining means to use the Pointer field in the entry with VPN = q to point to the entry with VPN = p.



Hashed Page Table - Example

- Operating system (OS) grabs p from the CPU, and performs h(p) to get same_key.
- OS looks up the first entry in the Hashed Page Table with key = same_key and checks p against the first entry's VPN field. It checks p against q. This is incorrect.
- OS uses the Pointer in the first entry to find the second entry. It knows that the second entry has the same key = same_key, because the Page Table is constructed this way. OS checks p against the second entry's VPN field. It checks p against p. This is correct.
- OS knows that this is the correct entry it is looking for. It grabs PFN from the second entry. It grabs r. r is the correct physical frame number that corresponds to virtual page number p.
- OS uses are to look for the physical frame it wants in physical memory, and looks for the exact word wanted which is offset by d within frame r in physical memory. OS grabs the contents of the word and we're done.



Inverted Page Table

- Inverted Page Table is the global page table which is maintained by the Operating System for all the processes.
- In inverted page table, the number of entries is equal to the number of frames in the main memory.
- It can be used to overcome the drawbacks of page table.
- There is always a space reserved for the page regardless of the fact that whether it is present in the main memory or not. However, this is simply the wastage of the memory if the page is not present.
- We can save this wastage by just inverting the page table.
- We can save the details only for the pages which are present in the main memory.
- Frames are the indices and the information saved inside the block will be Process ID and page number.

0				
1				
2	P2	10	P5	11
3	P3	15		
4	P15	4		
5				
6				
7				
8				
9				
10				

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
- TLB can accelerate access
- But how to implement shared memory ?
- One mapping of a virtual address to the shared physical address

Inverted Page Table

- Still, since some programs may have very large virtual address spaces, resulting in (relatively) large page tables.
- A different approach, currently adopted on the RT/RIOS, is an inverted page table.
- The basic function that we want to accomplish for paging is to map a virtual address into a physical address, by mapping a page number into a frame number.
- The page table stores the frame number into an array indexed by page number: given the page number, the frame number is achieved by a simple indexed load.
- An inverted page table stores the page number in an array indexed by frame number. Given the page number, you search for the entry with that page number: the entry number specifies the frame.
- An inverted page table has one major advantage: it need never be larger than the size of physical memory. Assuming that a page table entry is 8 bytes, and pages are 4K bytes, then an inverted page table always take 8 bytes per frame, or 8 bytes for every 4K bytes of physical memory -- about 0.5 per cent.

Inverted Page Table

- By contrast, the size of a page table is determined by the amount of virtual memory used.

- If only small programs are run, the page table sizes will be small; if large programs (in terms of their virtual memory size) are run, the page tables will be large (by about the same factor: a page table entry is probably about 4 or 8 bytes, and each 4K page requires one page table entry).

- Since each process requires its own page tables (the page tables are swapped as part of a context switch), if several large virtual memory programs are run, the total amount of page table space needed is the sum of the page table space for each process.

- An inverted page table has one major disadvantage: it is more complex and expensive to use (in terms of time) for mapping the virtual address to a physical address.

Inverted Page Table

- For a small inverted page table, it would be possible to keep the entire thing in a set of associative registers, but for a larger inverted page table (corresponding to a larger physical memory), it will need to be kept in memory and searched. For example, a 4M memory with 4K pages, requires an inverted page table of 1K entries (8K bytes at 8 bytes per entry).

- The search of the inverted page table can be improved by using a hashed search.

- The virtual address is hashed to get an index into a hash table.

- The hash table indicates the first address of a chain of values which with the same hash value; the chain is searched to determine the frame associated with the given virtual address.

- With a large enough hash table and a good hash function, the length of each hash chain would be small, and the search would be quick. The current guess is that, for the RT, the hash chain is normally only 1 or 2 entries long.

Inverted Page Table

- This approach uses memory for the hash table and the hash chain pointers in each table entry, to allow a faster search of the inverted page table.
- In addition, if the size of the hash table is fixed, adding (and using) more physical memory would tend to lengthen the hash chains and slow the search.
- Thus the size of the hash table for the RIOS (Riverside-Irvine Operating System) is set as a function of the size of physical memory, varying from 4K to 1M entries.
- Speed problems can be statistically improved by the careful use of caching. For example, a Translation Lookaside Buffer (TLB) is commonly used to remember recent virtual-to-physical translations in an associative memory.
- This can be particularly useful for inverted page tables, since it avoids both the cost of the hash function and the search down the hash chain.

Inverted Page Table

- RT/RIOS, concern about the size of page tables led to the use of an inverted page table rather than a page table.

- This sacrifices the time to translate a virtual address to a physical address - a hashed search of variable length for an inverted page table versus exactly one indexed access for a page table, for the more predictable memory requirements - the memory required for the inverted page table is more predictable than those for page table.

- The use of TLB's - which can be used for either addressing scheme, was felt to minimize the performance impact.

Inverted Page Table - Sharing - Additional Input

- For more complex systems, however, other problems arise with the inverted page table.
- The primary property of the inverted page table is that it records, for each frame, the specific virtual address page that is in that frame.
- This then implies that the virtual address associated with each page must be unique.
- This restriction is not true for a page table: several page table entries can map to the same frame, allowing several different virtual addresses to map to the same memory. This is known as aliasing. Aliasing is easily supported for page tables; it is more difficult to support with an inverted page table.

Inverted Page Table - Sharing - Additional Input

- If the three computations have their local data at the same virtual addresses, an inverted page table cannot store all three at the same time.

- If there are three different frames in memory, each at virtual address 0xffaf000, for each of three different processes, and the inverted page table then has three different entries with the same content, 0xffaf000, how can we search the inverted page table to find the 'right' frame ?

- The interpretation of 'right' varies for each process.

- There are two solutions to this problem. One is to change the inverted page table when the cpu is switched from one process to another; thus only one frame has the aliased virtual address at a time.

Inverted Page Table - Sharing - Additional Input

- The other approach is to modify the virtual address so that the addresses are unique, adding a value which is unique to the process.
- On the RT/RIOS for example, this situation is handled by selecting the values of the segment registers so that the virtual addresses for the different processes are different.
- The 4-bit segment field maps to a 24-bit segment number. 24 bits allows a large number of segment numbers, so that the allocation of segment numbers for RT/RIOS allows the virtual addresses (which include the segment id) to be unique to a process.
- Alternatively, if the same segment id is used, different processes can share memory. Memory sharing is limited in RT/RIOS (because the sharing is based on segments) to being on segment boundaries (the granularity of sharing is 256M), and to no more than 16 shared segments between processes (since there are only 16 segment registers).

Page Table - Existing View

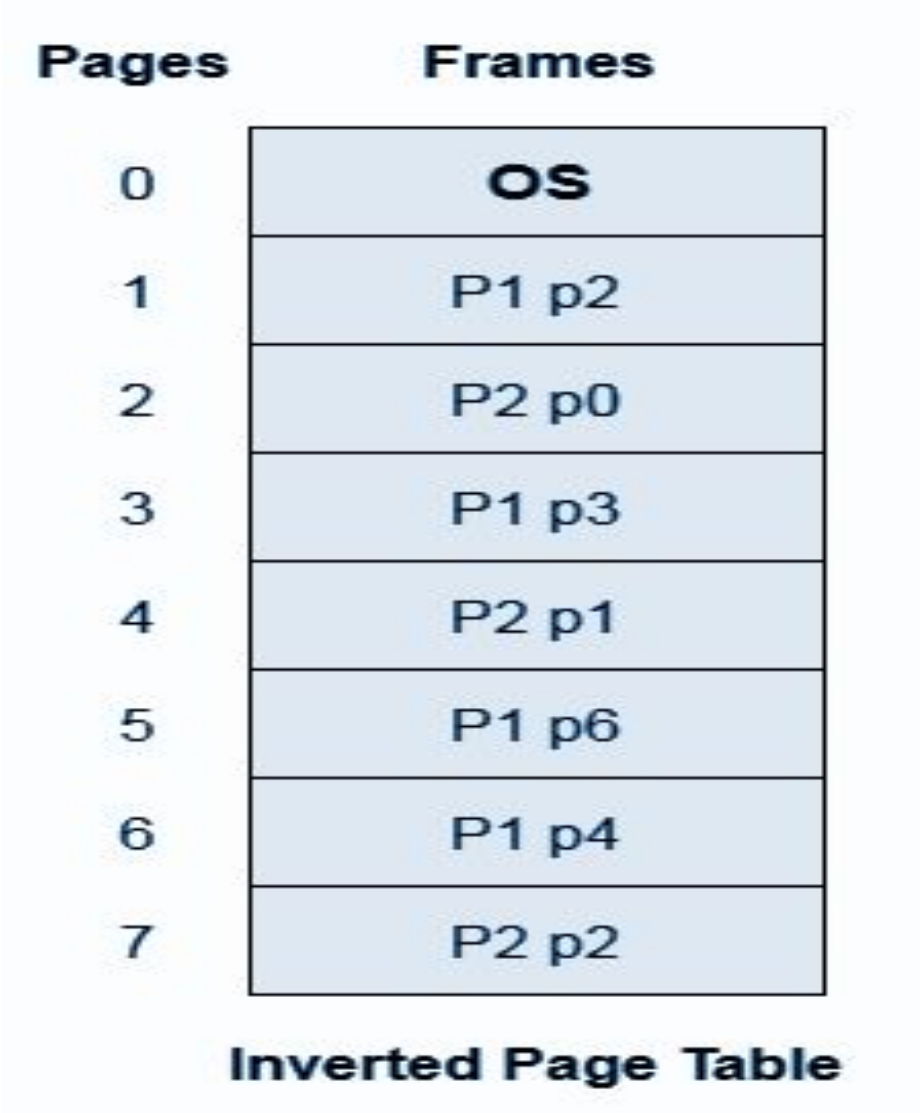
Pages	Frames
0	X
1	X
2	F1
3	F3
4	F6
5	X
6	F5

Page Table of P1

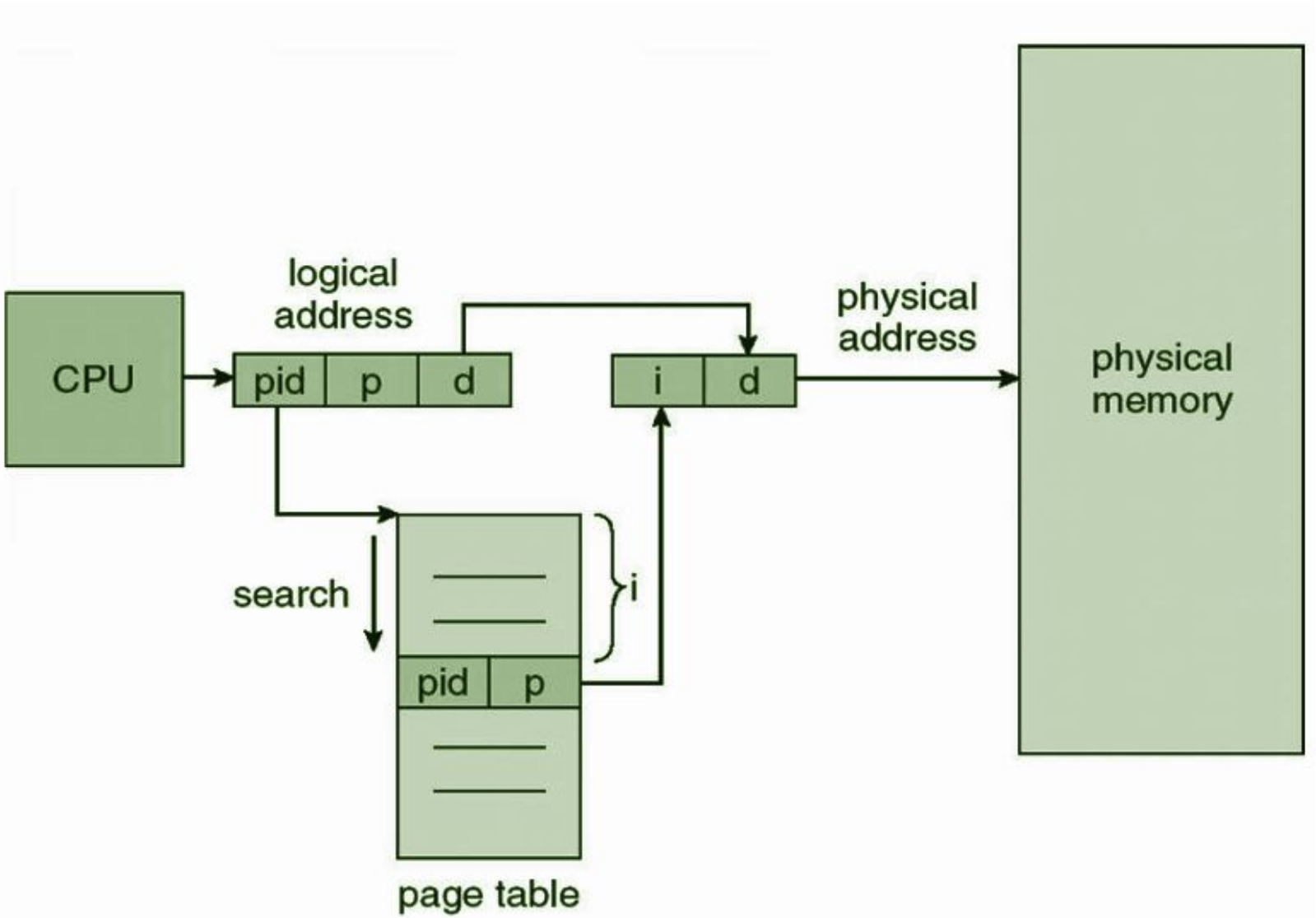
Pages	Frames
0	F2
1	F4
2	F7
3	X
4	X
5	X
6	F0

Page Table of P2

Inverted Page Table - New View



Inverted Page Table Architecture



Bigger Pages

Effect of page size on performance

- The number of frames is equal to the size of memory divided by the page-size. So an increase in page size means a decrease in the number of available frames.
- Having a fewer frames will increase the number of page faults because of the lower freedom in replacement choice.
- Large pages would also waste space by Internal Fragmentation.
- On the other hand, a larger page-size would draw in more memory per fault; so the number of fault may decrease if there is limited contention.
- Larger pages also reduce the number of TLB misses.

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com