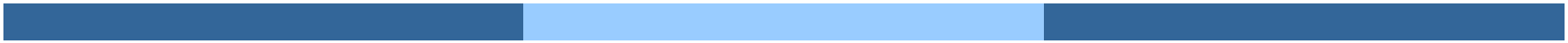
Main Memory and Address Translation

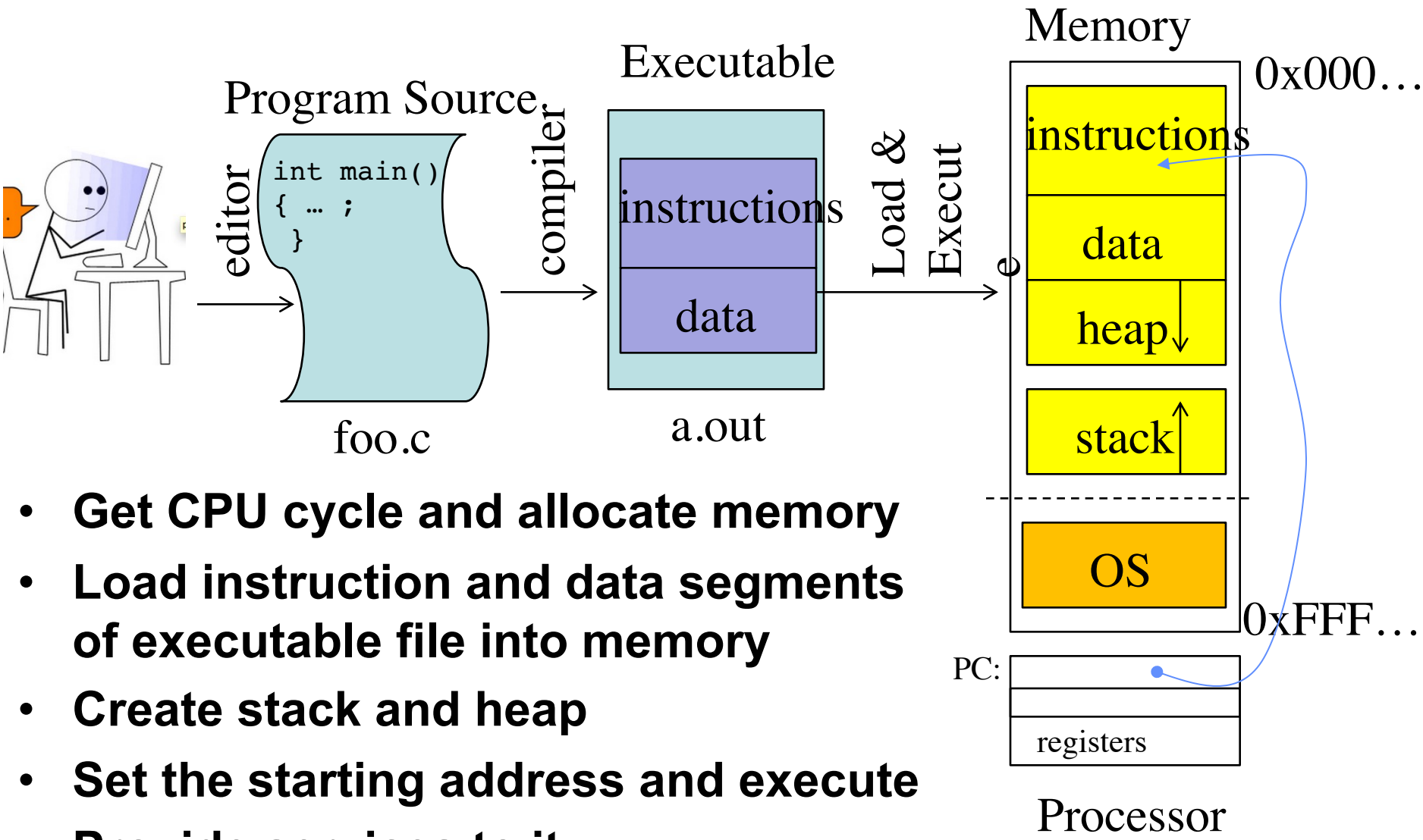


CS 170
Tao Yang

What to Learn

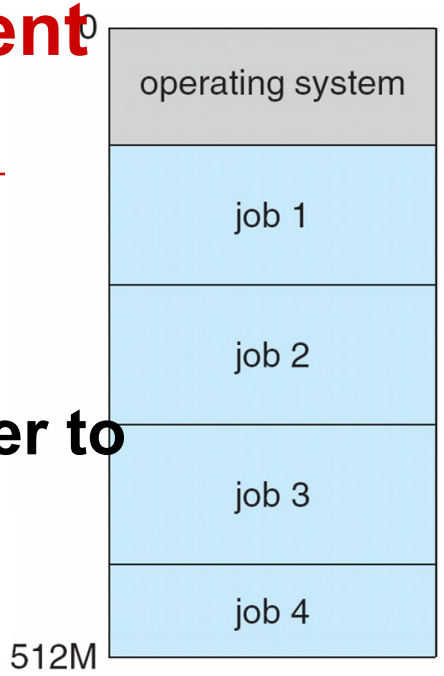
- **Program execution and address space**
- **Memory management**
  - Contiguous Memory Allocation
  - Segmentation
  - Address Translation with Paging
  - TLB support and performance impact

Steps for Running a Program



- **Get CPU cycle and allocate memory**
- **Load instruction and data segments of executable file into memory**
- **Create stack and heap**
- **Set the starting address and execute**
- **Provide services to it**

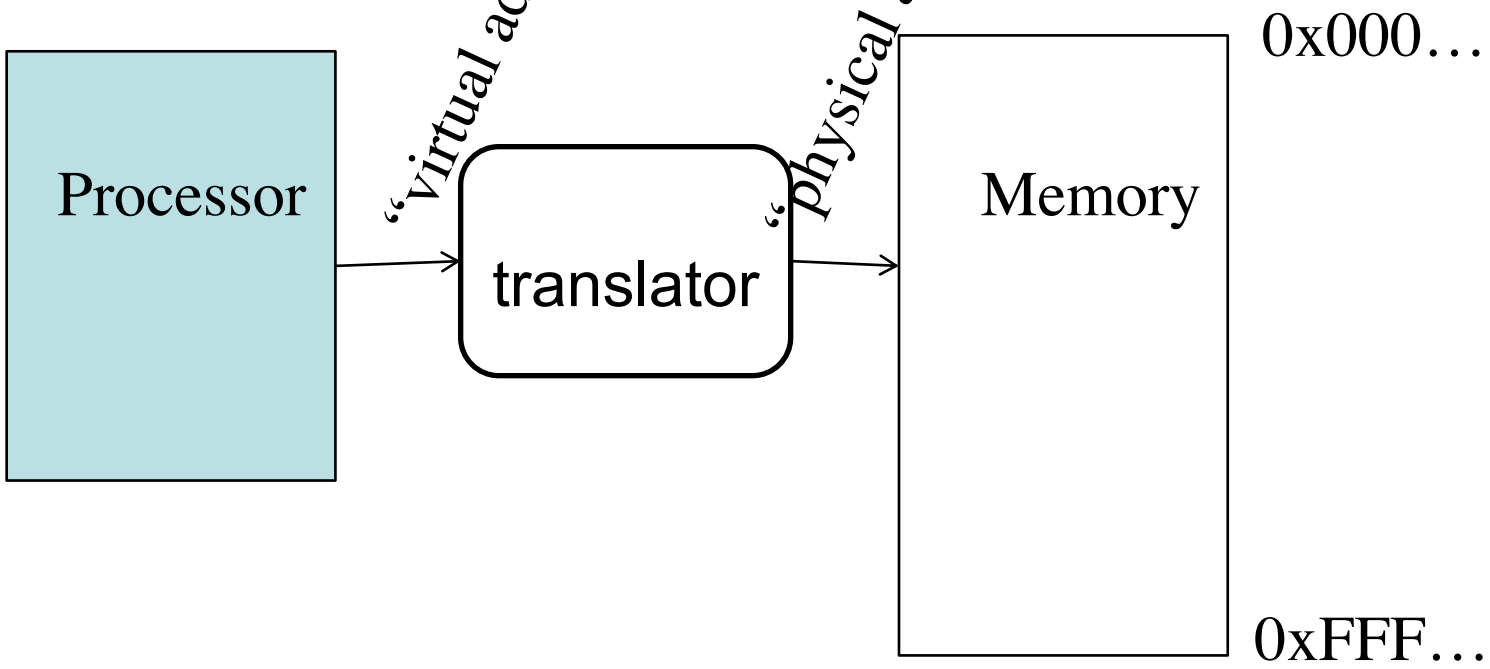
Role of Memory Management



- **Manage data/instructions in memory in order to execute**
- **Memory management activities**
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

Address Space Translation

- **Address Space:**
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- **Program operates in an address space that is distinct from the physical memory space of the machine**



Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **E.g. A program accesses logical address 0**
- **Physical address** – address seen by the memory unit
  - E.g. Address 0 of this program is placed in physical memory 100
- When their binding (translation) happens?

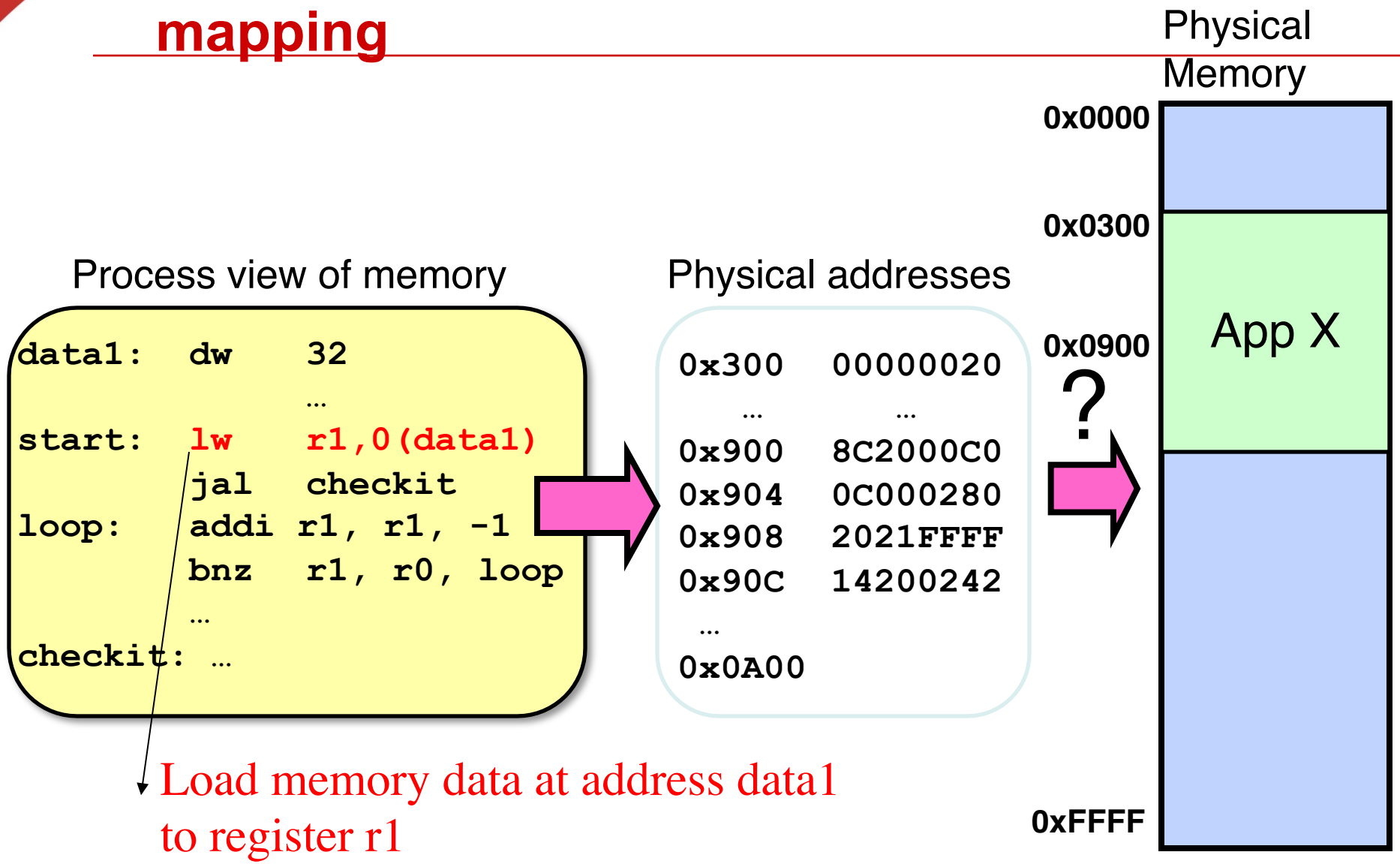
Compile time: If memory location known a priori, **absolute code** can be generated.

Load time: Must generate **relocatable code** if memory location is not known at compile time

Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. (e.g. Dynamic library)

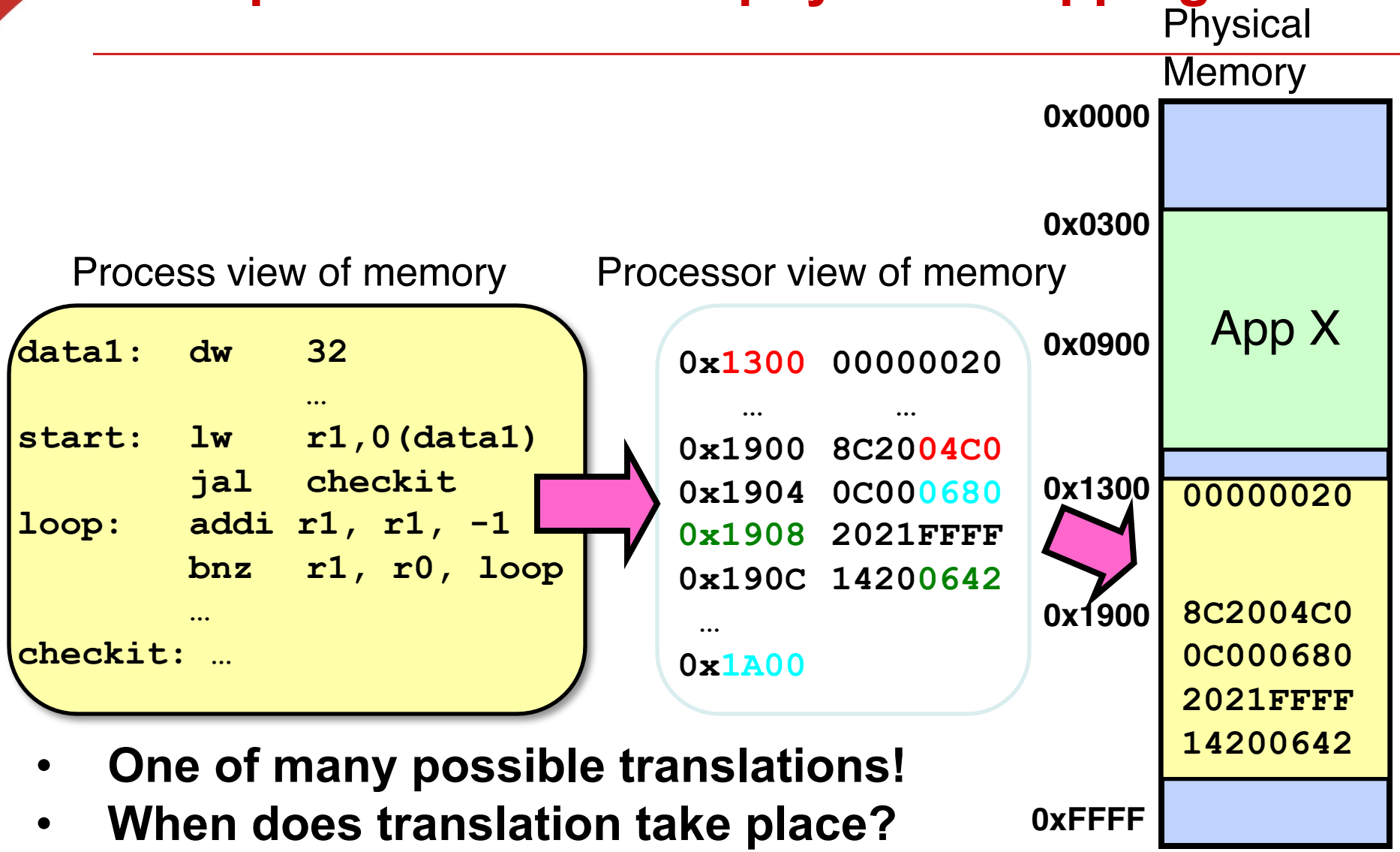
- Memory management unit(MMU) provides address translation

Example of virtual to physical address mapping



Need address translation!

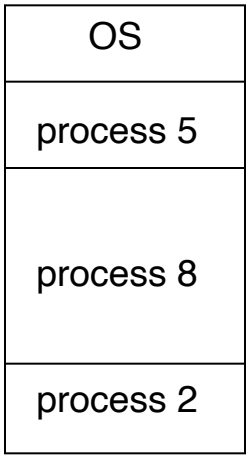
Example from virtual to physical mapping



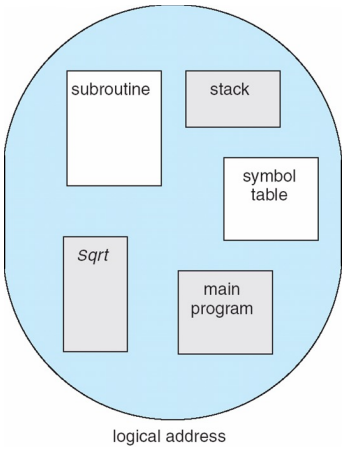
- **One of many possible translations!**
- **When does translation take place?**
  - **Compile time, Link/Load time, or Execution time?**

Method of Memory Allocation

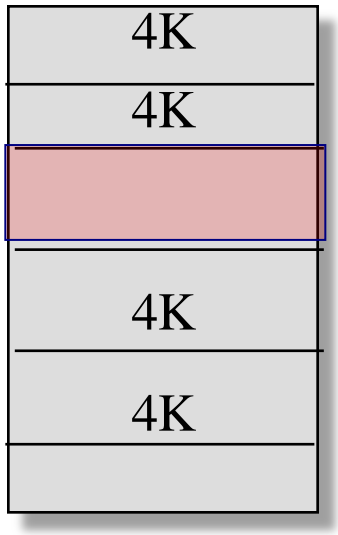
1. Contiguous allocation



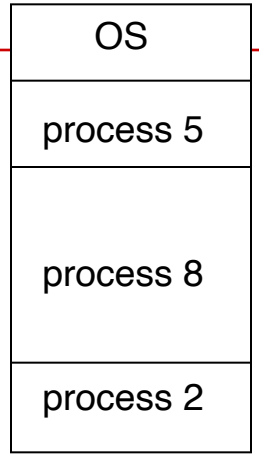
2. Segments



3. Paging



Method 1: Contiguous Memory Allocation

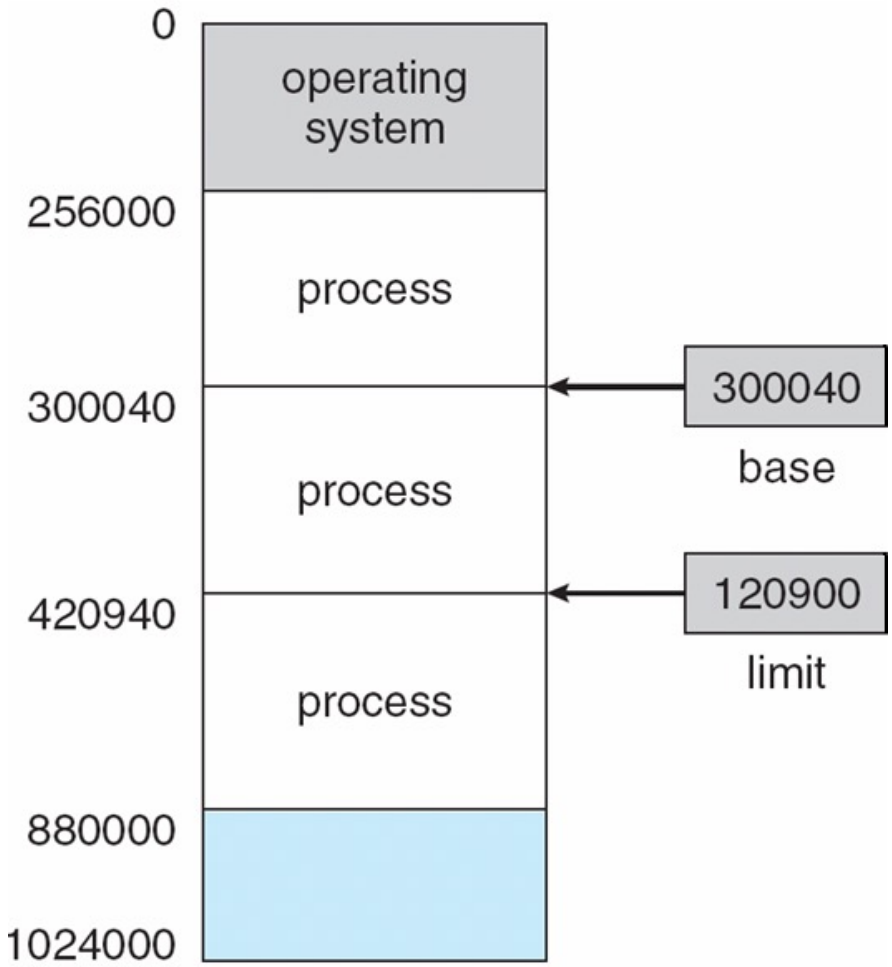


- **Main memory has two partitions:**
  - Resident operating system,

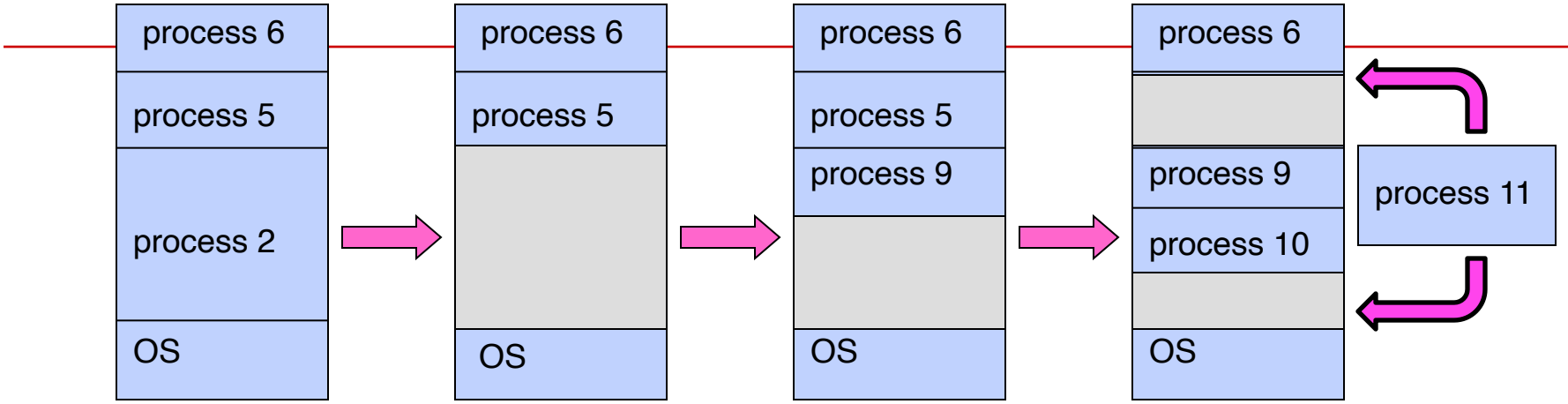
    usually held in low memory
  - User processes then held in high memory
- **Hardware support and protection**
  - **Base register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Example of Memory Protection for Multiprogramming

- **During switch, kernel loads** new base/limit from PCB (Process Control Block)
  - User not allowed to change base/limit registers



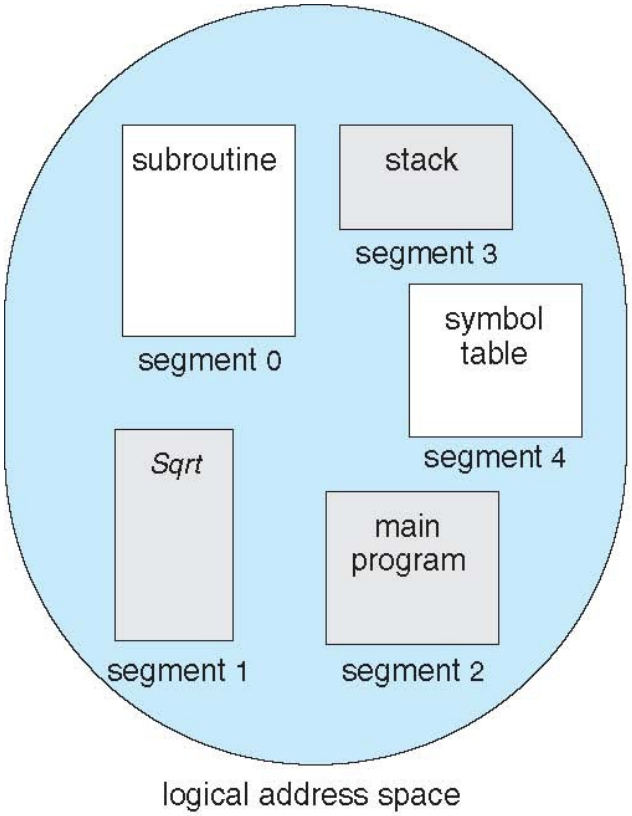
Issues with Contiguous Memory Allocation



- **External Fragmentation problem (free gaps between used slots)**
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- **Missing support for sparse address space**
  - Would like to have multiple chunks/program
  - E.g.: Code, Data, Stack
- **Hard to do inter-process sharing**
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

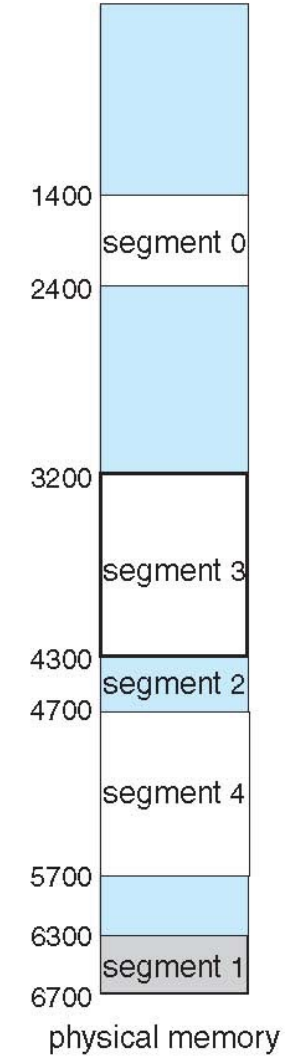
Allocation Method 2: Segmentation

- **A program is a collection of segments**
  - Each is a logical unit with some semantic view such as: Code, data, stack.
- **Each segment is a region of contiguous memory**
  - Has a base and limit
  - Can reside anywhere in memory

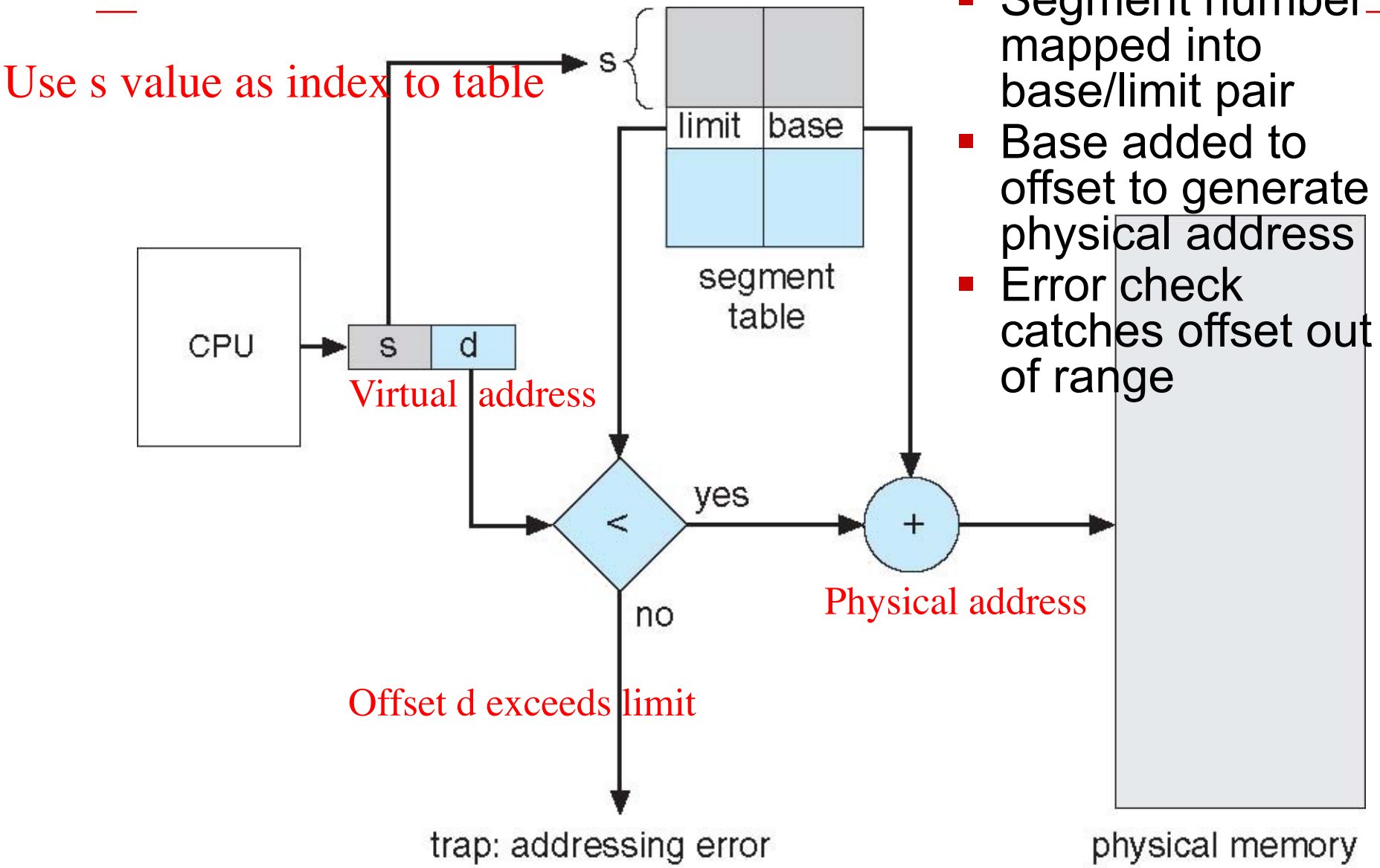


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Segment-based Address Translation



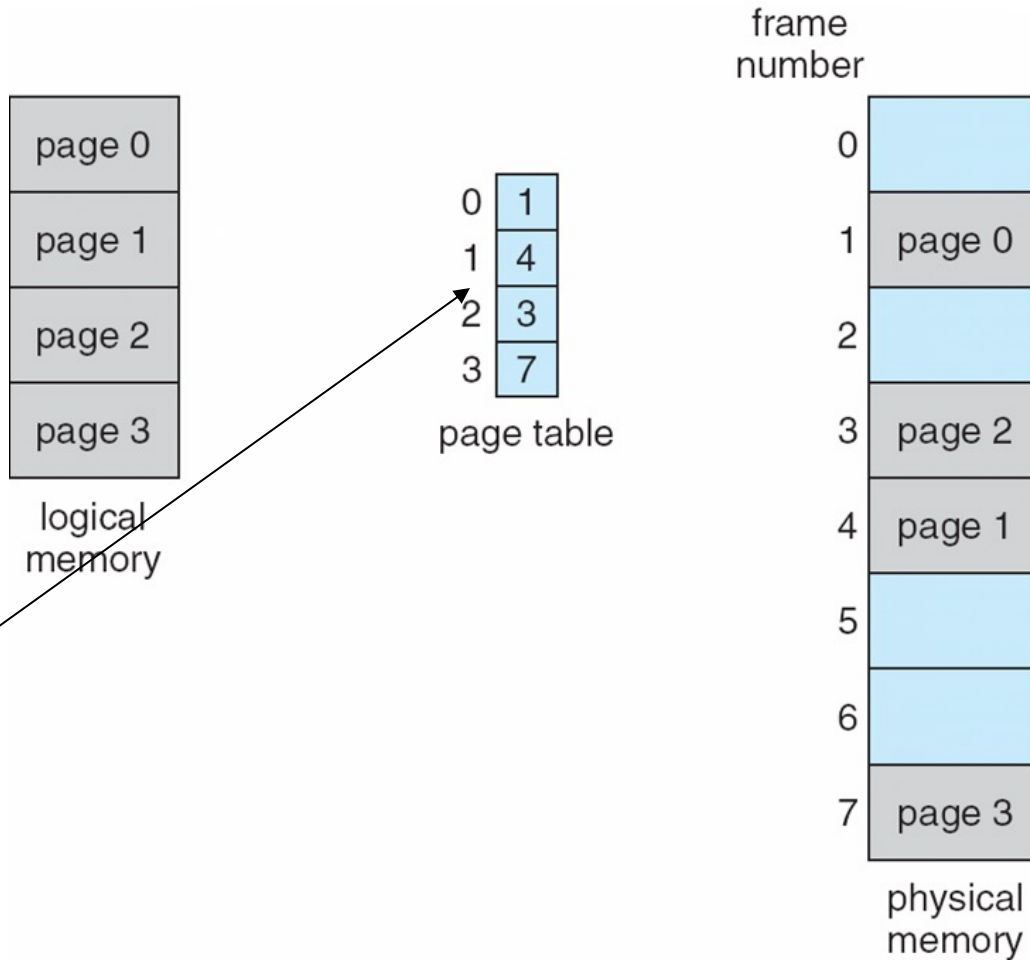
- Segment number mapped into base/limit pair
- Base added to offset to generate physical address
- Error check catches offset out of range

Problems with Segmentation

- **High overhead of manging a large number of variable size and dynmaically growing memory segments**
  - Memory is divided into many used/unused regions over the time
  - Not easy to find space to fit for a new segment
- **Fragmentation: wasted space. Two types**
  - External fragmentation: free gaps between allocated chunks
  - Internal fragmentation: don't need all memory within allocated chunks

Memory Allocation Method 3: Paging

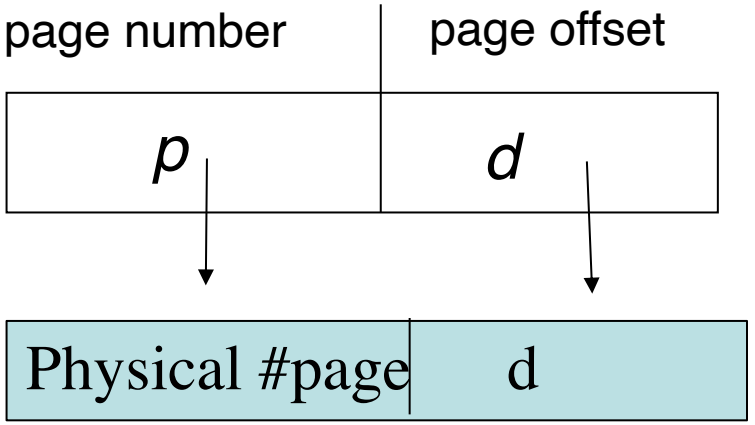
- **Divide physical memory into fixed-sized blocks called frames or pages**
  - Size is power of 2, e.g. 4K bytes.
- **Divide logical memory into blocks of same size called pages or logical pages**
- **Use a page table per process for address translation**
  - Index 1 entry is the physical page (frame) number of virtual page 1



Address Translation Scheme

- **Logical (virtual) address → Physical address**
- **Address generated by CPU is divided into:**
  - **Page number (*p*)**
  - **Page offset (d)**
- **Simple mapping:**
  - **Logical page no →**

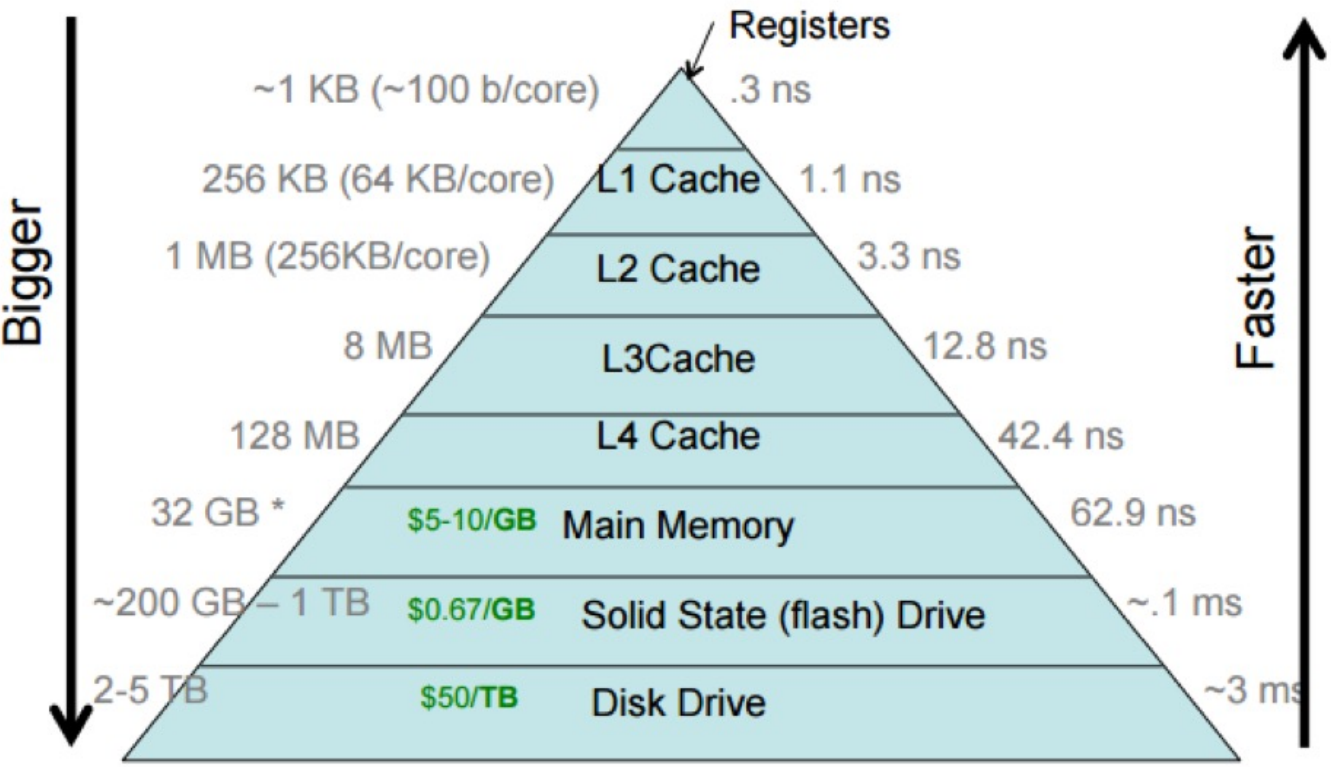
  Physical page no



  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, which can hold 4 million entries

$2^{10} = 1024 = 1K.$ $2^{20} = 1M = 1024K.$ $2^{30} = 1G = 1024M$

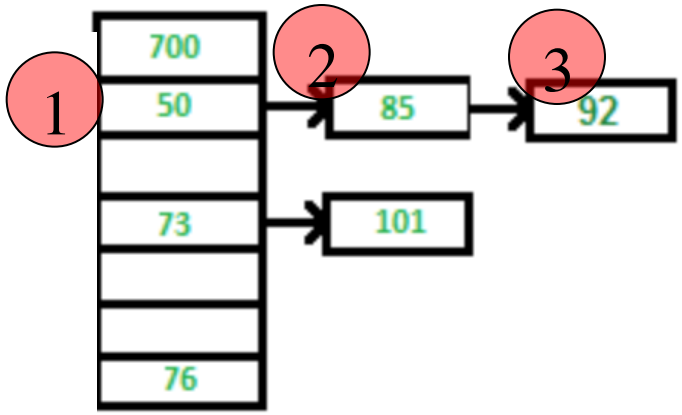
Performance of Memory Hierarchy Affects Design Choices



- RAM is about two orders of magnitude slower than a register
- Design considerations
  - Minimize # of slow memory accesses
  - Use faster cache

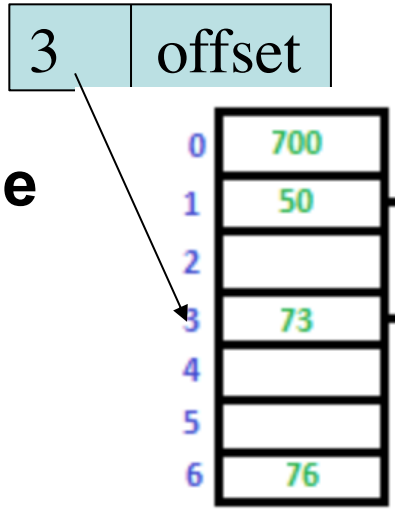
Design options to map logical #page to physical #page

- **Option 1: Use a hash table**
- **Cost for translation is dominated by total # of memory accesses**
  - 1 access to locate the table entry
  - + #access to visit the daisy chain.

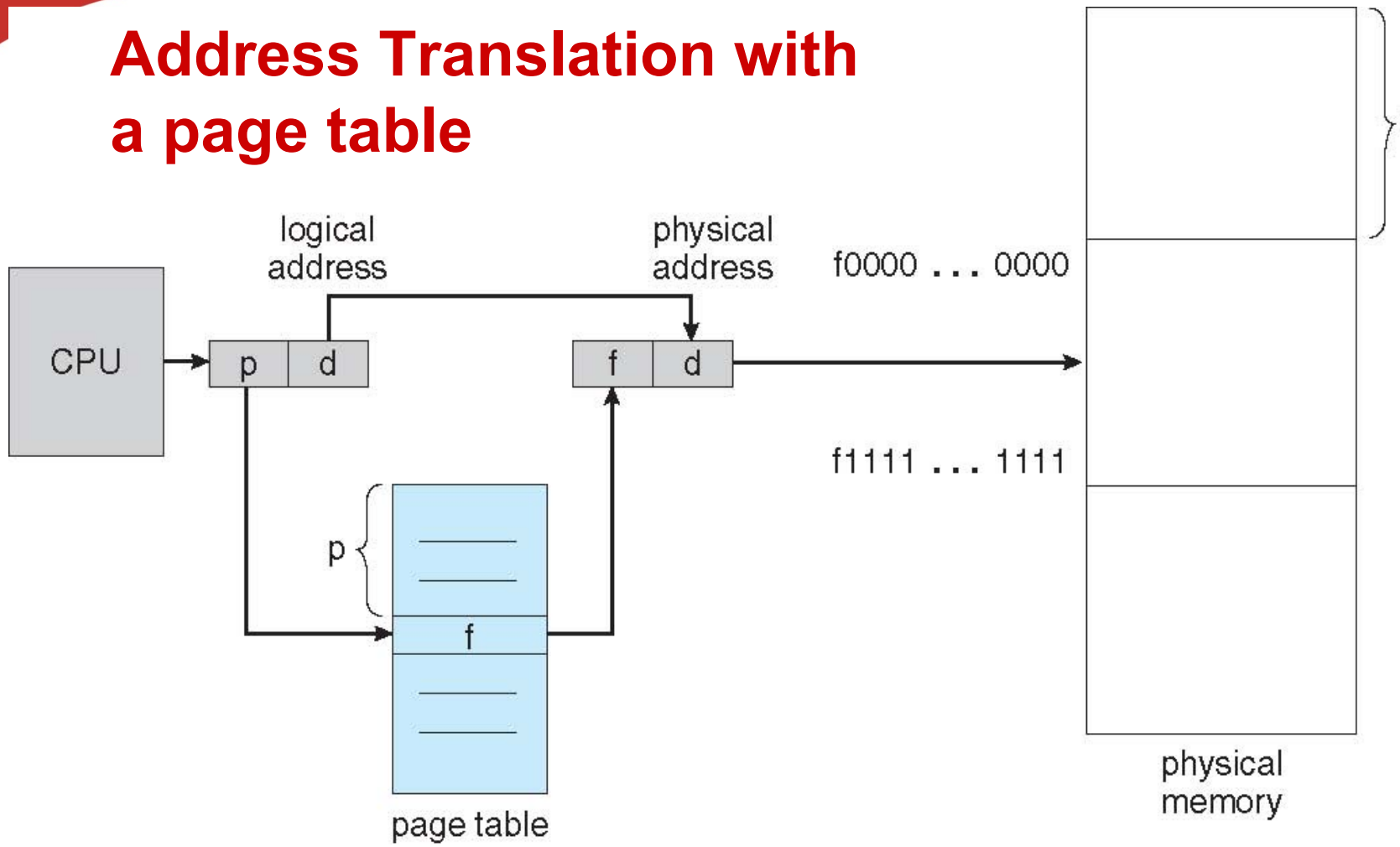


Each memory access is slow. Bad when a long daisy chain.

- **Option 2: Direct mapping with a page table**
  - Physical page p →
    value of p-th entry in table
  - Cost of translation: 1 memory access



Address Translation with a page table



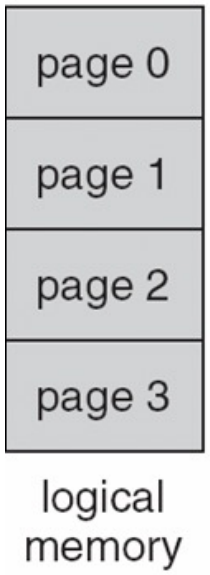
Virtual address = virtual page number + page offset

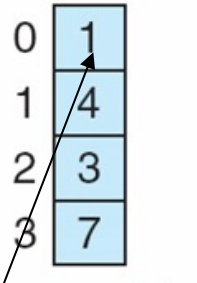
Role of page table: virtual page number → physical page number

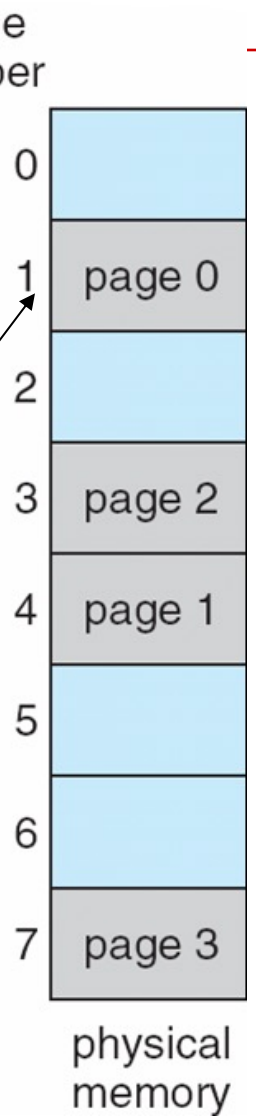
Physical address = physical page number + page offset

Example of Page-based Address Translation

- Given memory page of size 16 bytes
- Code: x =x +1
- 8-bit virtual address of x is: 4
- What is physical addr of x?







Virtual page offset

Virtual addr: 0000 0100

Virtual page number to access: 4 >>4= 0.

Offset 4 within Virual page 0

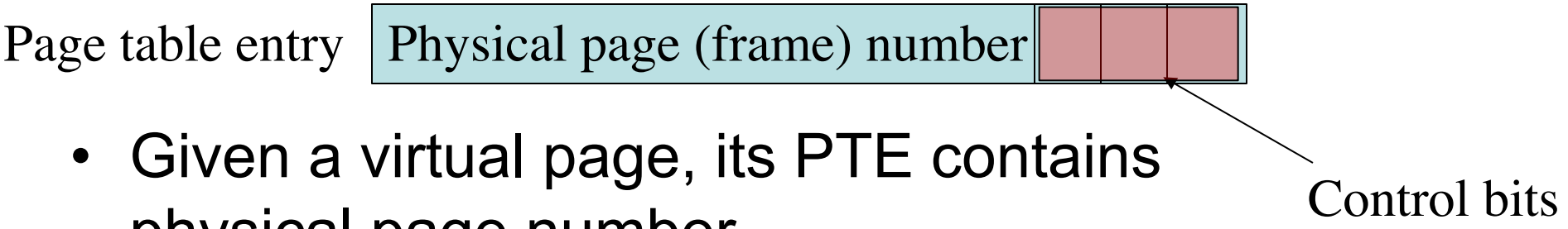
Phyiscal page number is memory fame 1.

Physical address is (1 <<4) +4

Phyiscal Page offset

0001 0100

Page Table Entry (PTE) and Memory Protection

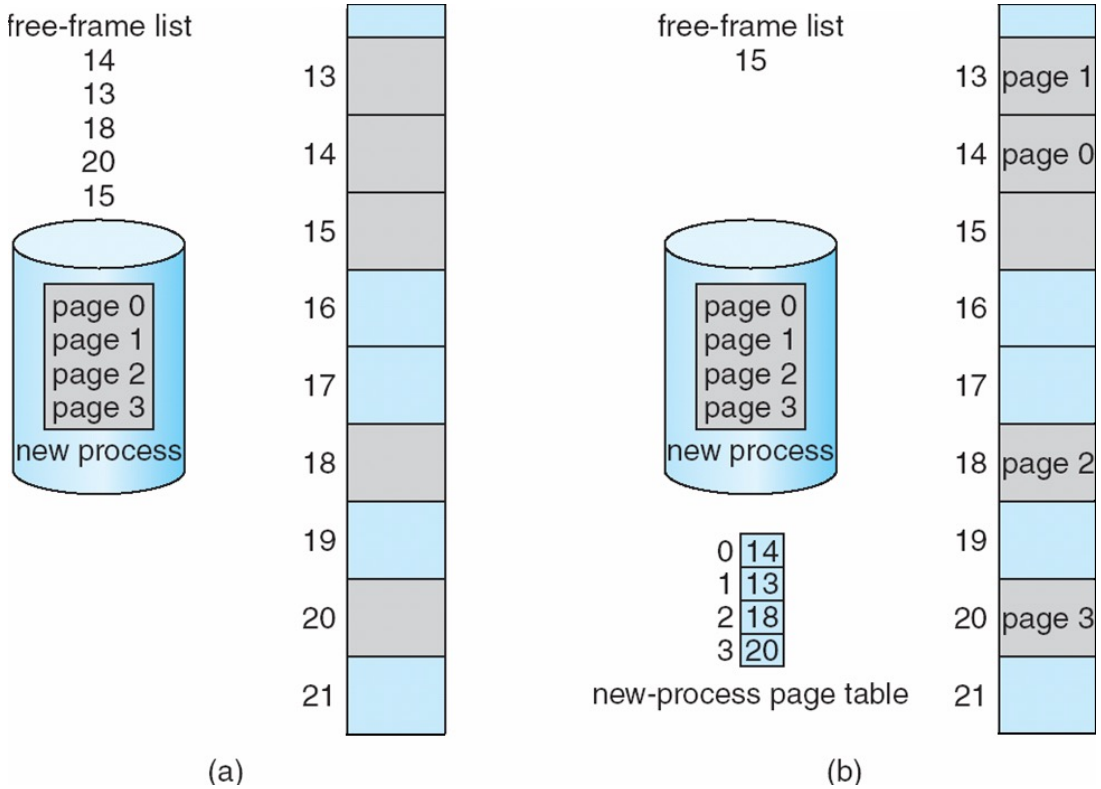


- Given a virtual page, its PTE contains physical page number.
- Other bits in PTE can be allocated for control such as memory protection and permission bits.
  - Read/write permission
  - "valid" bit indicates that the associated page is in memory
- Total number of bits in PTE can be selected to support a large physical space + control. E.g. 32 or 64.

Management of Free Pages: Use a bitmap

Use vector of bits to represent availability of each page

00110001110001101 … 110010 1⇒allocated, 0⇒free

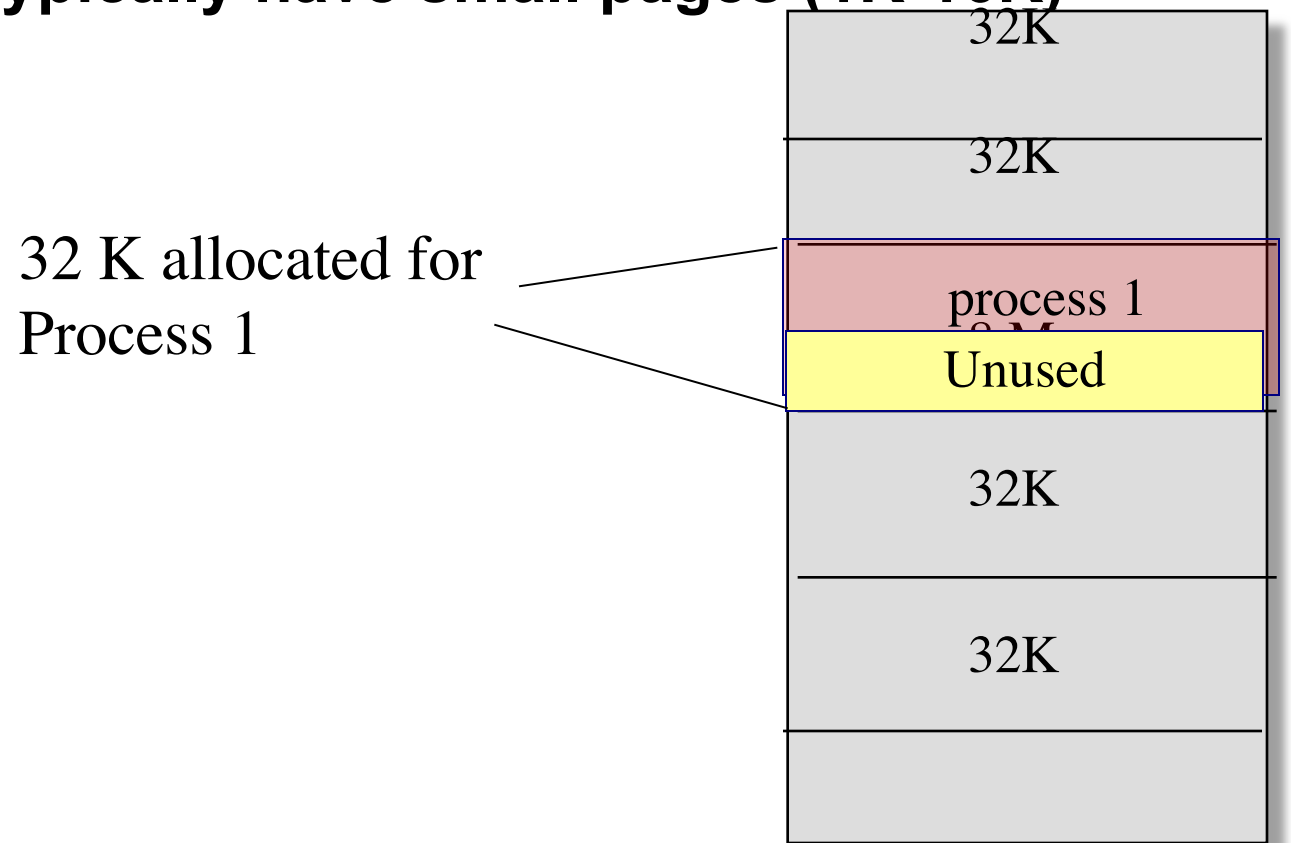


Before allocation

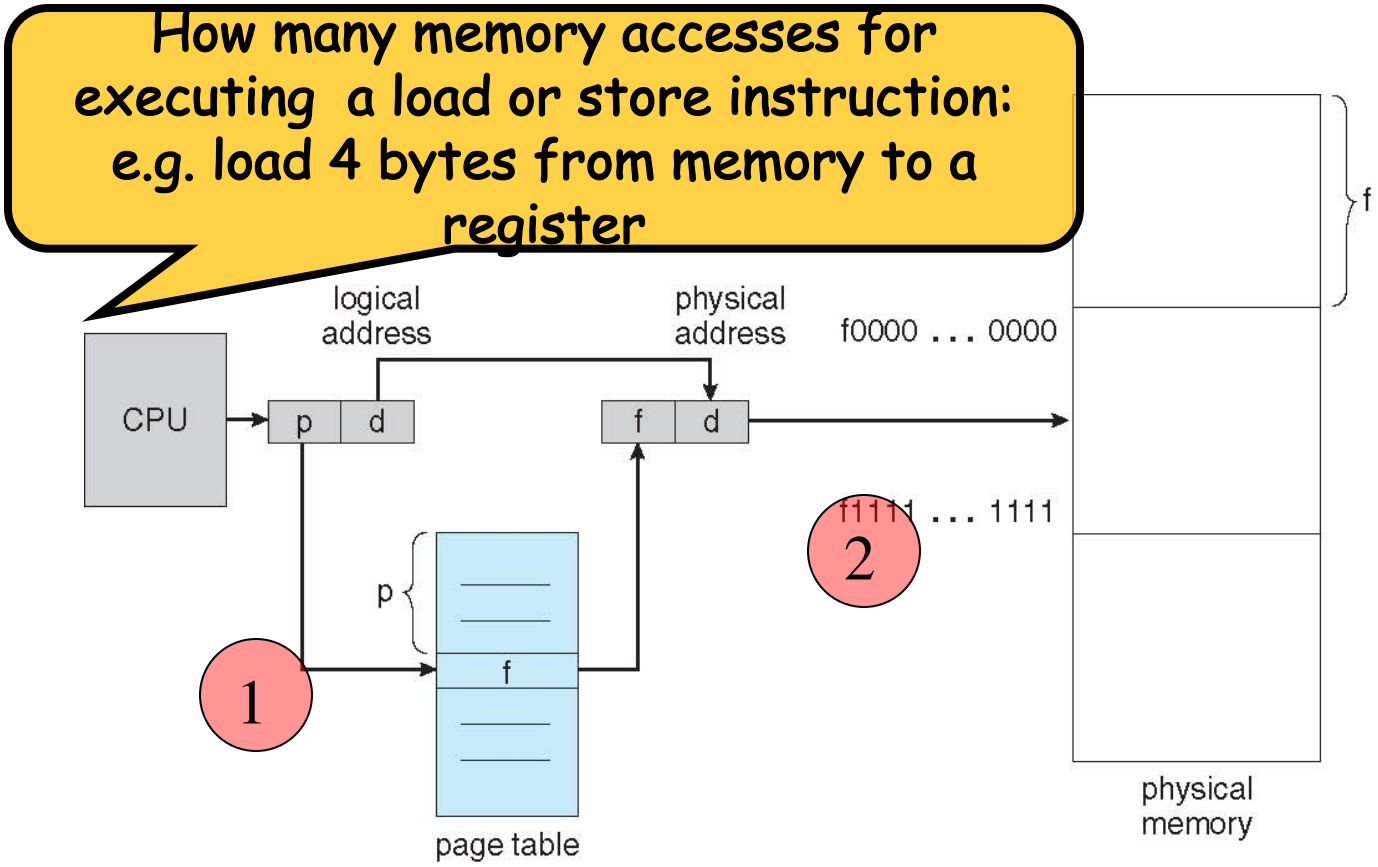
After allocation

Should pages be as big as our previous segments?

- **Remove external fragmentation, but lead to Internal Fragmentation**
  - **allocated memory is larger than requested memory;**
- **Typically have small pages (1K-16K)**



Time Performance of Memory Data Fetch with Address Translation



1 memory access for page table

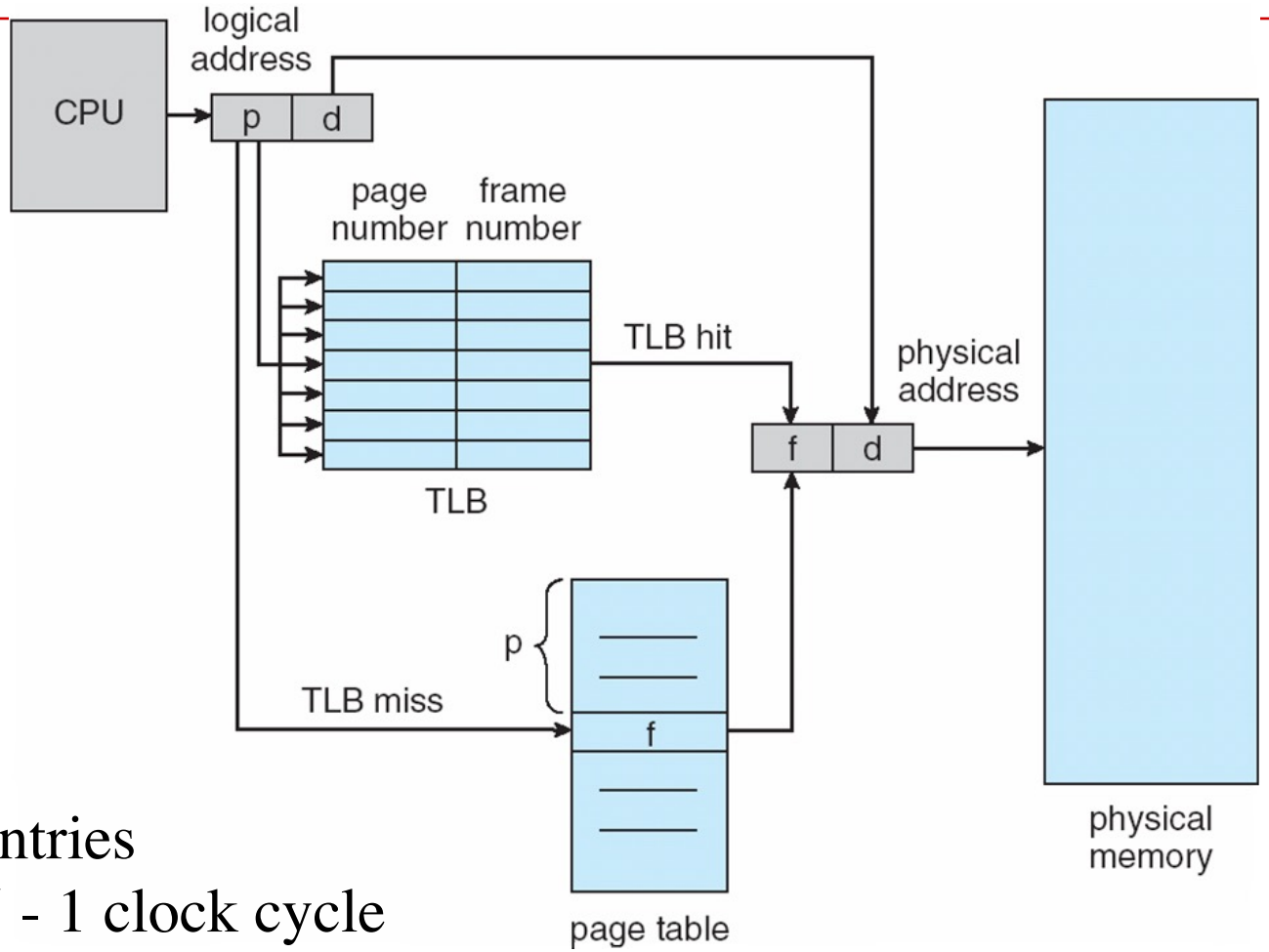
1 memory access to obtain the final data

TLB for Faster Address Translation

- **Every data/instruction fetch requires 1 memory access at least in address translation.**
  - Slow
  - Address can be improved by using a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- **Address translation for logical page p**
  - If p is in TLB, get physical page # out
  - Otherwise get frame # from page table in memory

Logical #	Physical#
p	d

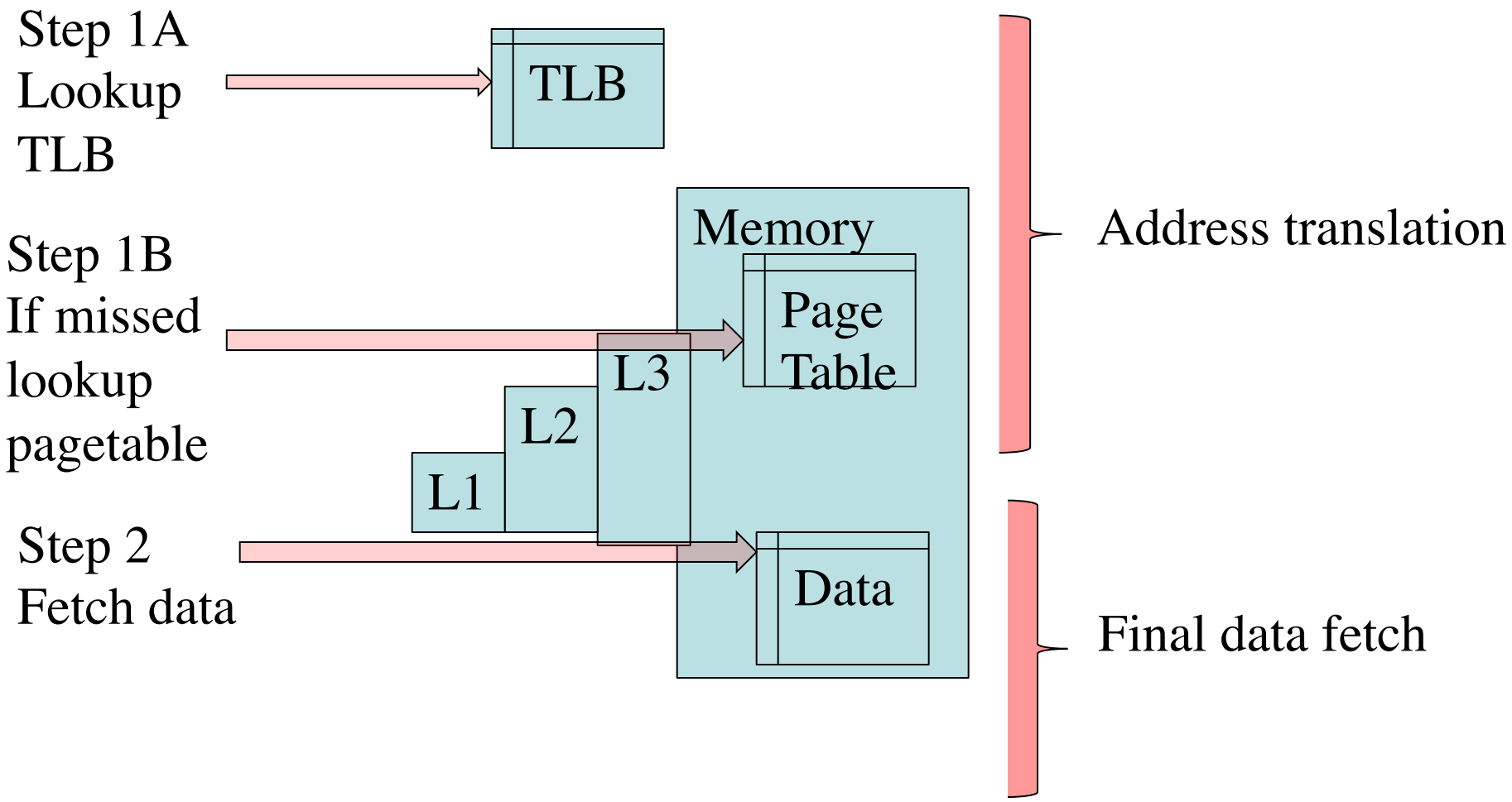
Paging Hardware With TLB



Typical TLB

- Size: 8 - 4,096 entries
- Access time: 0.5 - 1 clock cycle
- Miss penalty: 10 - 100 clock cycles
- Miss rate: 0.01 - 10%

Steps in executing a load/store instruction with TLB, L1, L2, & L3



Example: Performance Characteristics of TLB

- **Effective (average) address translation cost**

Cost(TLB lookup) + Cost(Full Translation)* TLB Miss Rate.

- **Example:** TLB hit takes 1 clock cycle, a miss takes 30 clock cycles to access a memory page table, and the miss rate is 1%.
  - **Average cost for translation**:

    1 + 30X0.01=1.30

  - **Total cost for a load/store instruction**

  = address translation+ final memory data access

  = 1.3+ 30 = 31.30 clock cycles

- 1GHz CPU → 1 ns/cycle

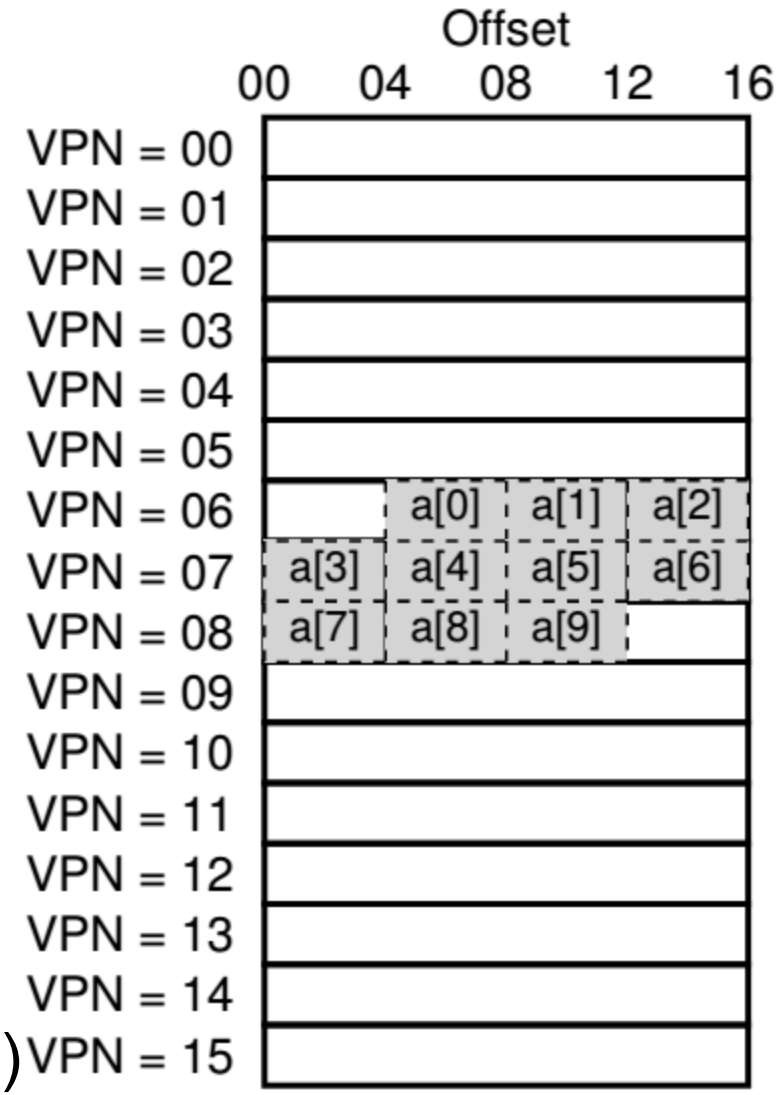
Example: Performance Characteristics of TLB

Each page has 16 byes, hosting 4 integers. Initially TLB is empty.

What is TLB miss rate of this Code with respect to a[]?

```
int sum=0;
For(i=3; i<10; i++)
   sum += a[i]
```

2/7



- How about this?

```
int sum=0;
For(i=0; i<100000; i++)
   sum += a[i]
```

25%

Example: Performance Characteristics of TLB

int b[1024][1024]

What is TLB miss rate of this Code with respect to array b?

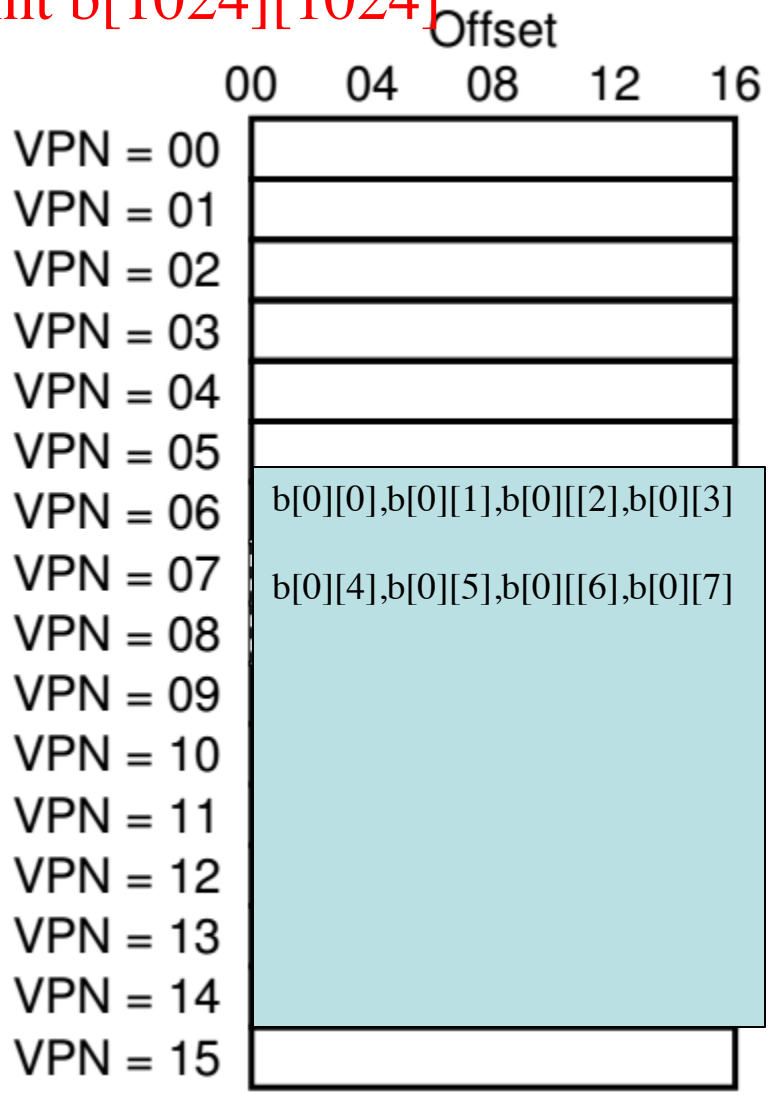
```
int b[1024][1024];
For(i=0; i<1024; i++)
  for(j=0; j<1024;j++)
     b[i][j] =b[i][j]+1;
```

1/8

- How about this?

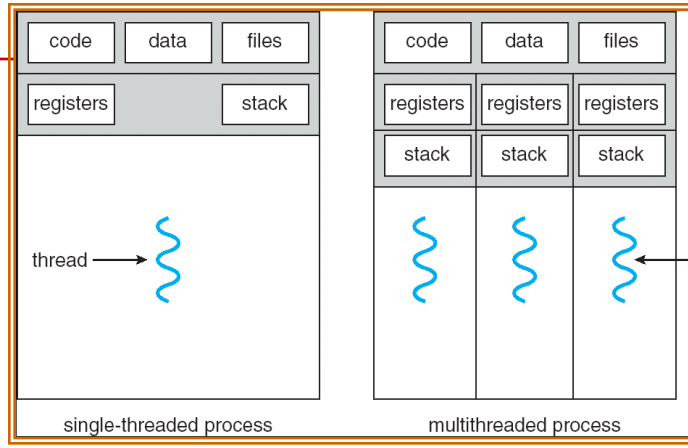
```
For(j=0; j<1024; j++)
  for(i=0; i<1024;i++)
     b[i][j] =b[i][j]+1;
```

50%



Where to store memory control info of a process or a thread?

- **Thread control block (TCB)**
  - Stack info
  - Registers and machine states
- **Process Control Block (PCB)**
  - Process identification data
  - Process state data
  - Process control data
    - memory space management
    - Pointer to a page table



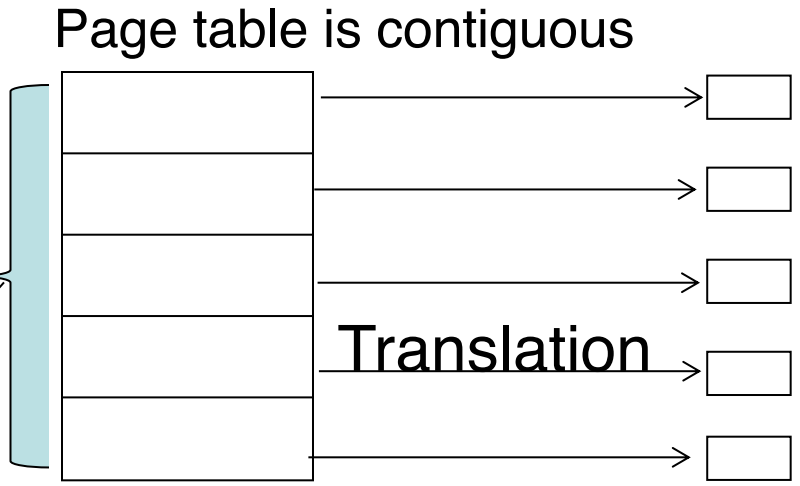
Where does page table reside:
Kernel vs user space?

Kernel.
Page table contains sensitive translation and permission flag

One page table per process
Page table size = # of pages in virtual space!

How big of address space can a page table support for translation?

- **The maximum size of one-level page table**



Constraint:Each page table needs to fit into a physical memory page!

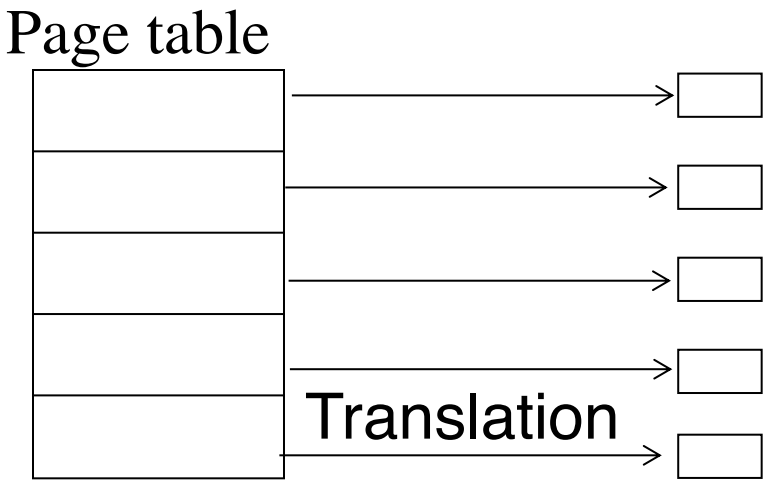
Why? A page table needs consecutive space.

- Memory allocated to a process is a set of nonconsecutive pages.

Maximum size = # entry * page size

One-level page table cannot handle large space

- **Example:**
  - 32 -bit address space with 4KB per page.
  - Page table would contain $2^{32}/ 2^{12}$= 1 million entries.
    - 4 bytes per entry
    - 1 million *4= 4MB page table (contiguous space)
      - Often not available



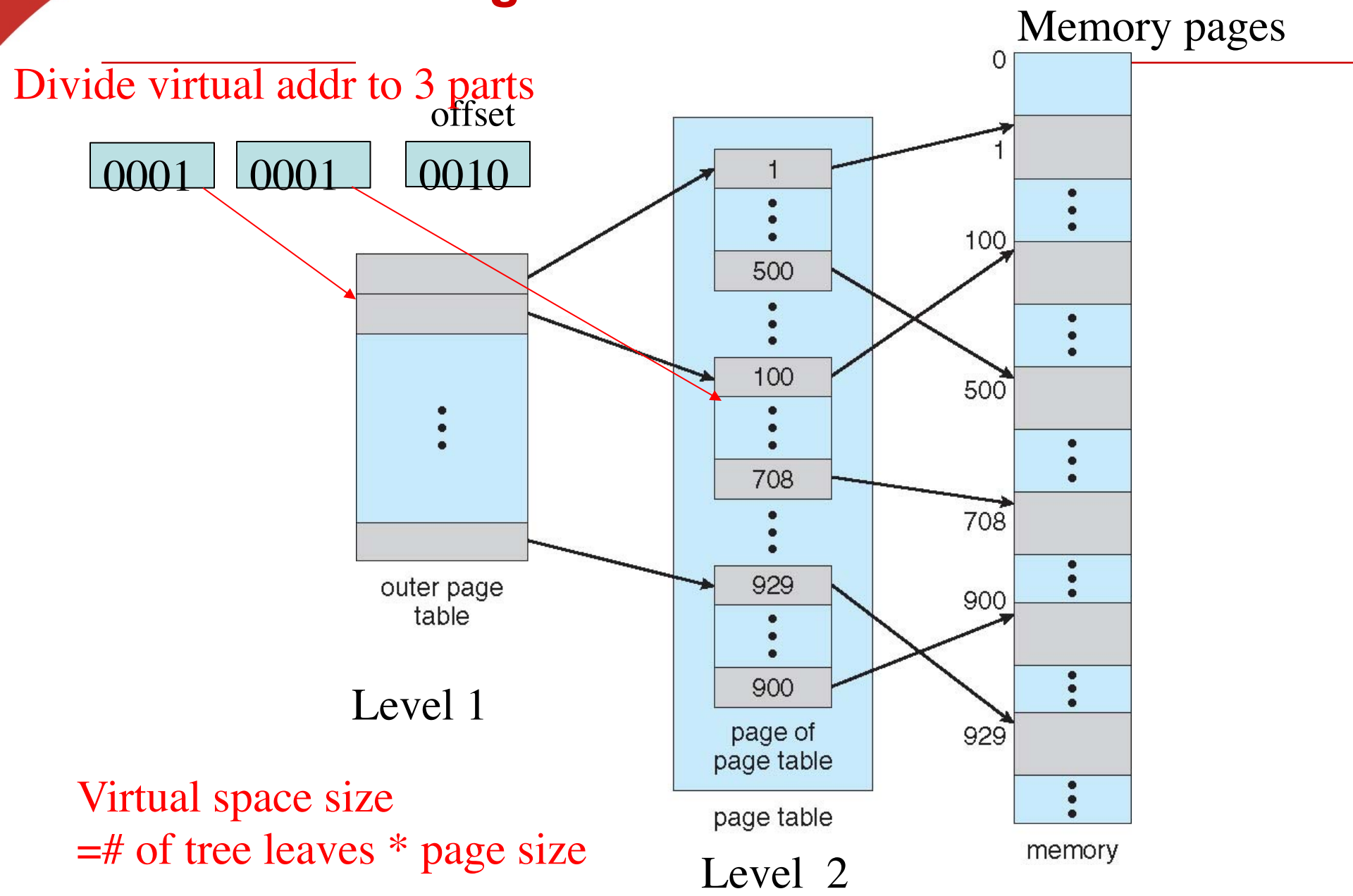
Solution for a larger space is to use a multi-level page table

Maximum size with 4KB per page

#entry= 4KB/4B = 1K.

Maximum logical space=1K*4KB= 4MB.

Two-Level Page Table for Address Translation

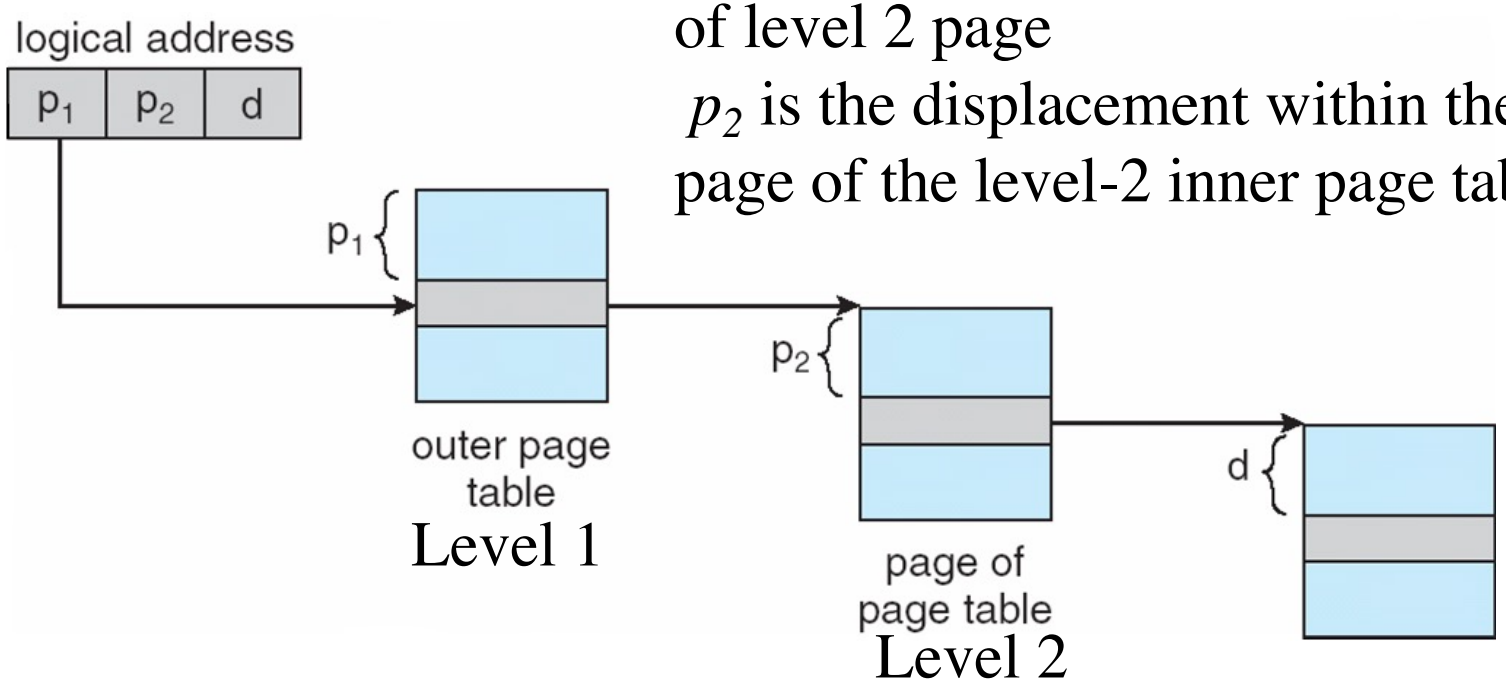


Virtual space size
=# of tree leaves * page size

Address-Translation Scheme

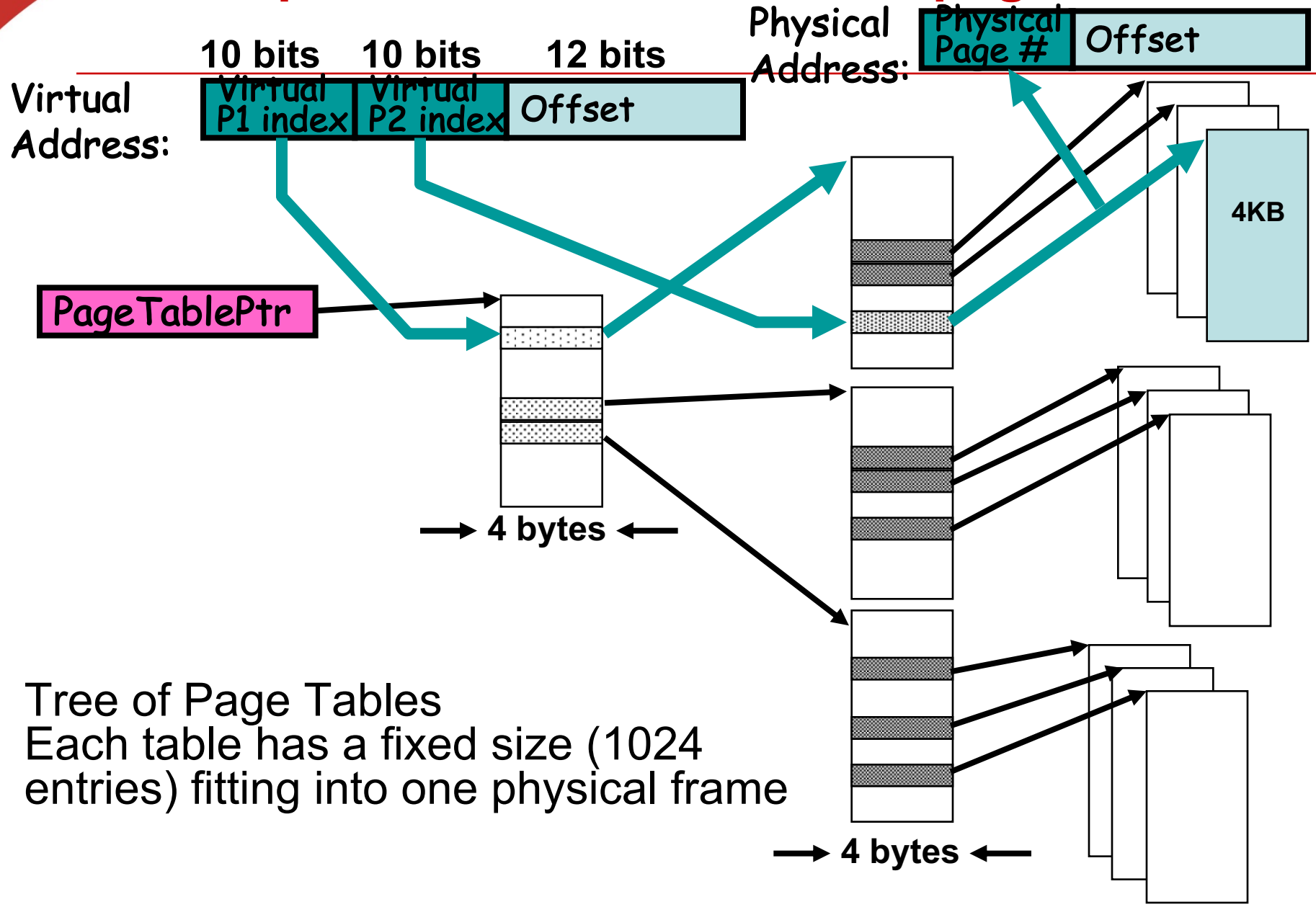
p_1 is an index into the outer level-1 page table. Entry content is location of level 2 page

p_2 is the displacement within the page of the level-2 inner page table



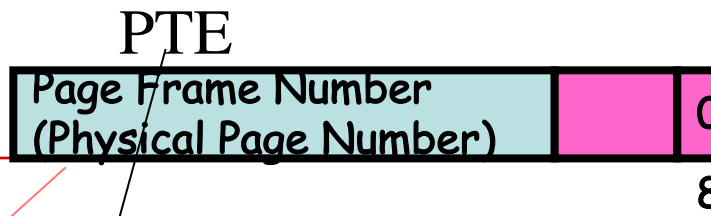
Entry content at Level-2 page table gives the final physical page number

Example: Flow of the two-level page table

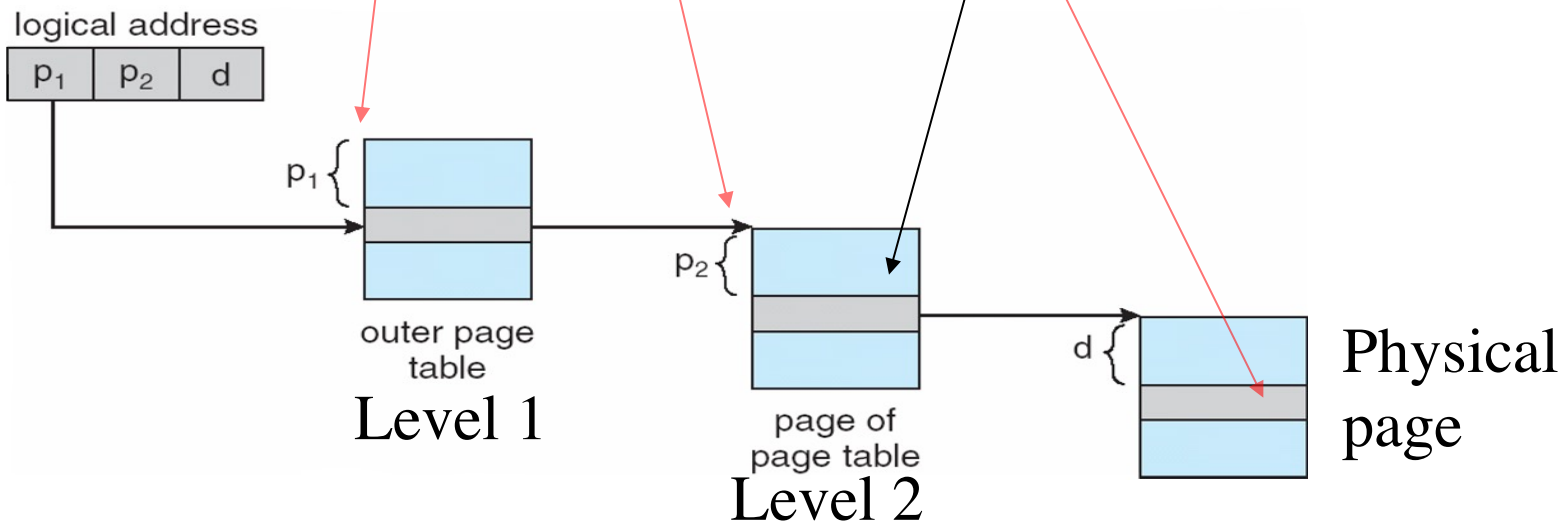


- Tree of Page Tables
- Each table has a fixed size (1024 entries) fitting into one physical frame

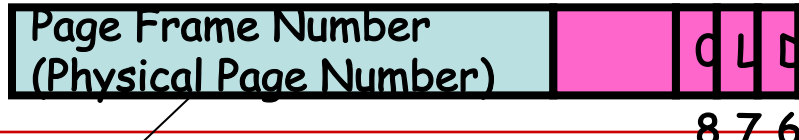
Properties of 2-Level Paging



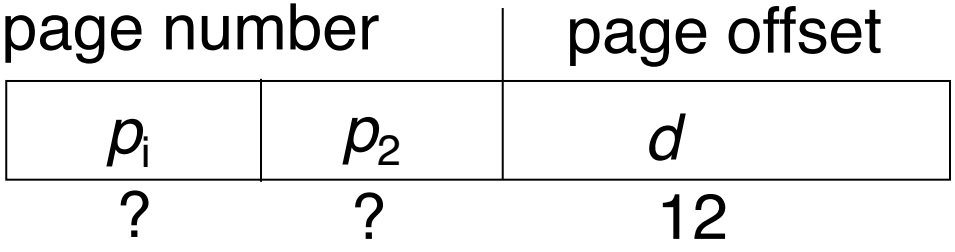
- #Bits for d = log (page size)
- #Bits for p_1 >= log (# entries in level-1 table)
- #Bits for p_2 = log (# entries in level-2 table)
- #Physical pages = 2^entry bits allocated in level-2 table.
  - Physical space size =#physical pages*page size
- Virtual space size = # entry in level-1 table * # entry in level 2 table * page size
  - p_1 may have some unused bits.



Two-Level Paging Example

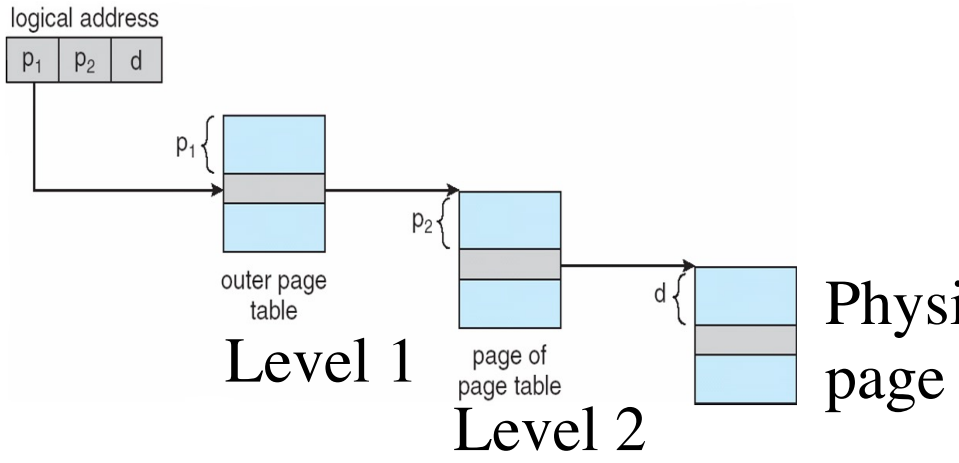


- **A virtual address (on 32-bit machine with 4K page size) is divided into:**
  - a page number area with 20 bits
  - a page offset consisting of 12 bits
- **Each page table entry uses 4 bytes.**
  - 20 bits for physical page number
- **How to build a two-level paging scheme?**
  - How many page table entries can a memory page hold?
  - What are p_1, p_2?



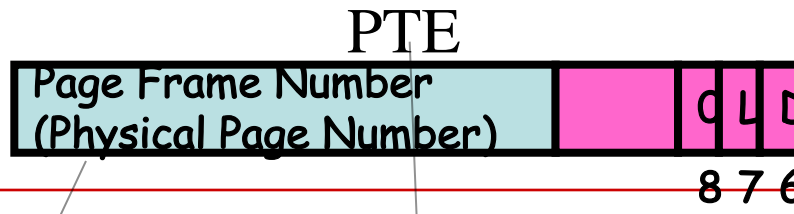
Analysis of a Two-Level Paging Example

- **A memory page with 4KB holds 1K entries and each uses 4B.**
- **1K entries require 10 bits for P_1 and P_2 offset**
  - Bits of p_1 >= log (# entries in level-1 table)
  - Bits of p_2 = log (# entries in level-2 table)
- **The page number is further divided into:**
  - a 10-bit level-1 index
  - a 10-bit level-2 index



page number		page offset
p_i	p_2	d
10	10	12

Maximum virtual & physcal space size



- **Maximum virtual space size**

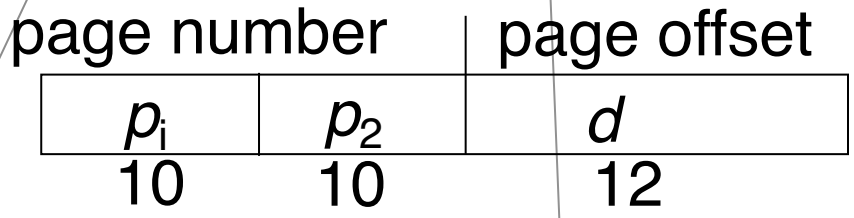
=#leaves * page size

=# entry in level-1 page table * # entry in level-2 page table * page size

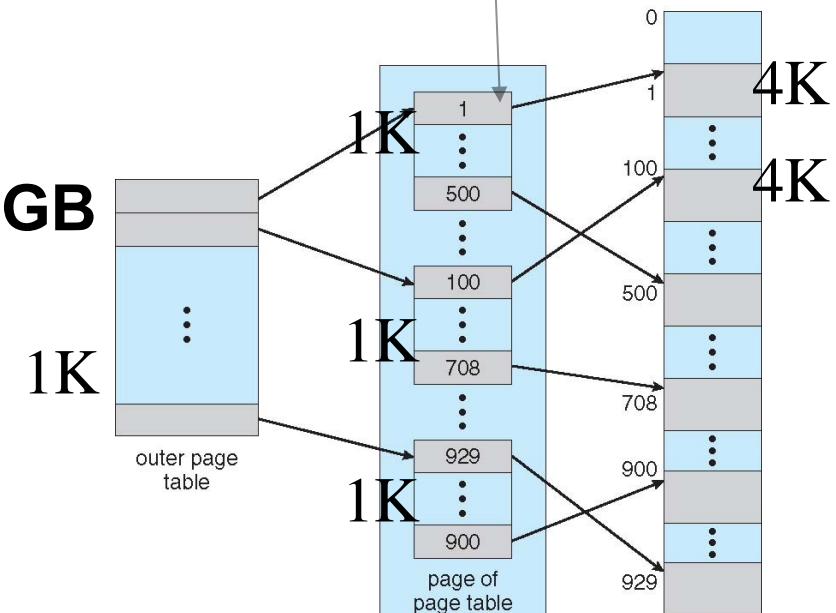
= 1K * 1K * 4KB

= 2^{32} bytes

= 4GB



- **Max. Physical space =2^{20}*4K=4GB**



What if we use 2 bytes for each table entry?

- Increased virtual space size?
- Increased physical space size?

What if we use 2 bytes for each table entry?

- **Maximum logical space size**

entry in level-1 page table * # entry in level-2 page table * page size

= 0.5K * 2K * 4KB

= 2^{32} bytes

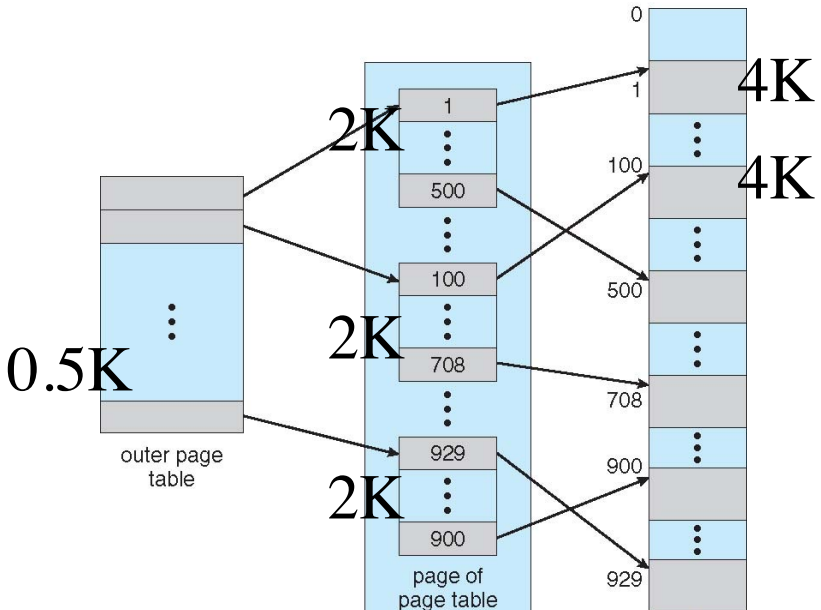
= 4GB

page number		page offset
p_i	p_2	d
9	11	12

Physical space $\leq 2^{16} * 4K = 256MB$

What if we use 2 bytes for each table entry?

- Increased virtual space size? Same
- Increased physical space size? Decrease

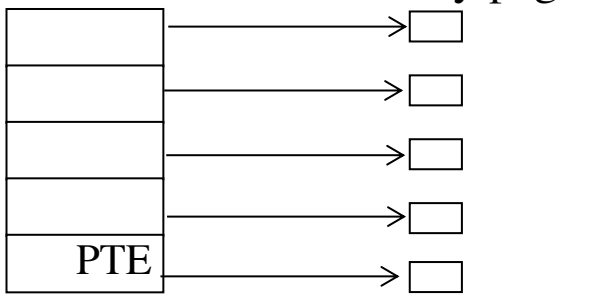


Maximum virtual space size for 1 or multi-level page-table based address translation

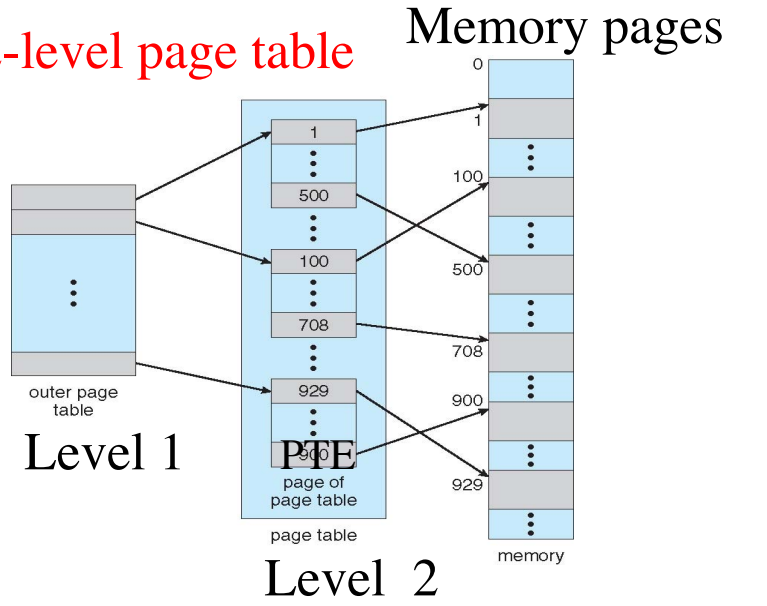
Vitual space size =# of tree leaves * page size

1-level page table

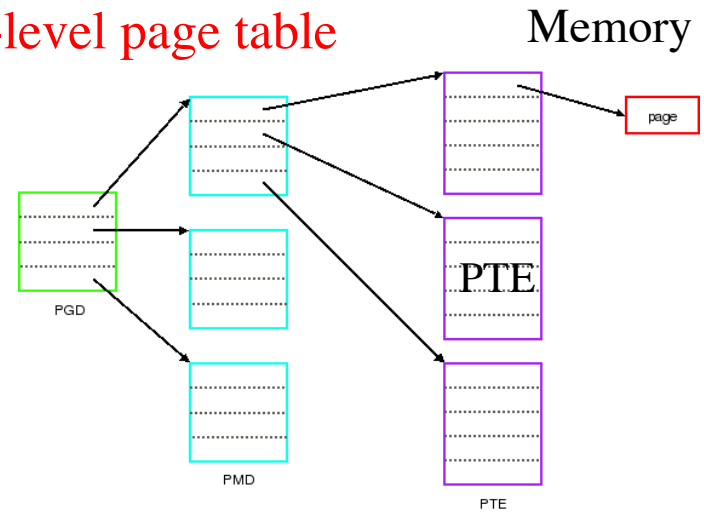
#of leaves = #page table entries per page



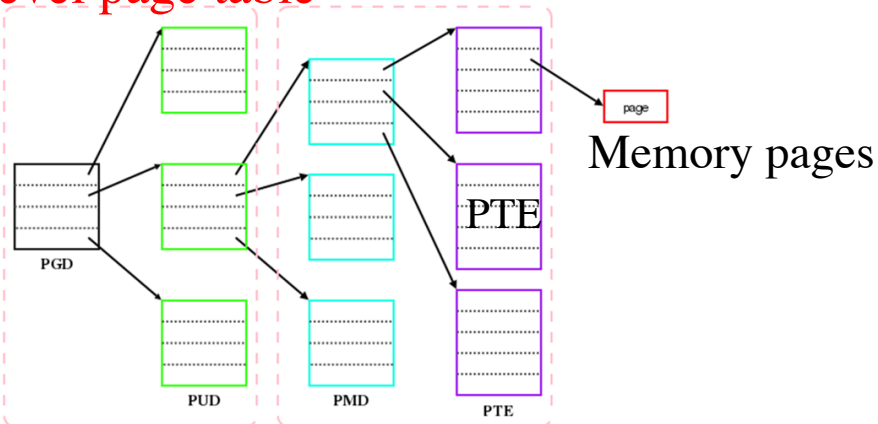
2-level page table



3-level page table

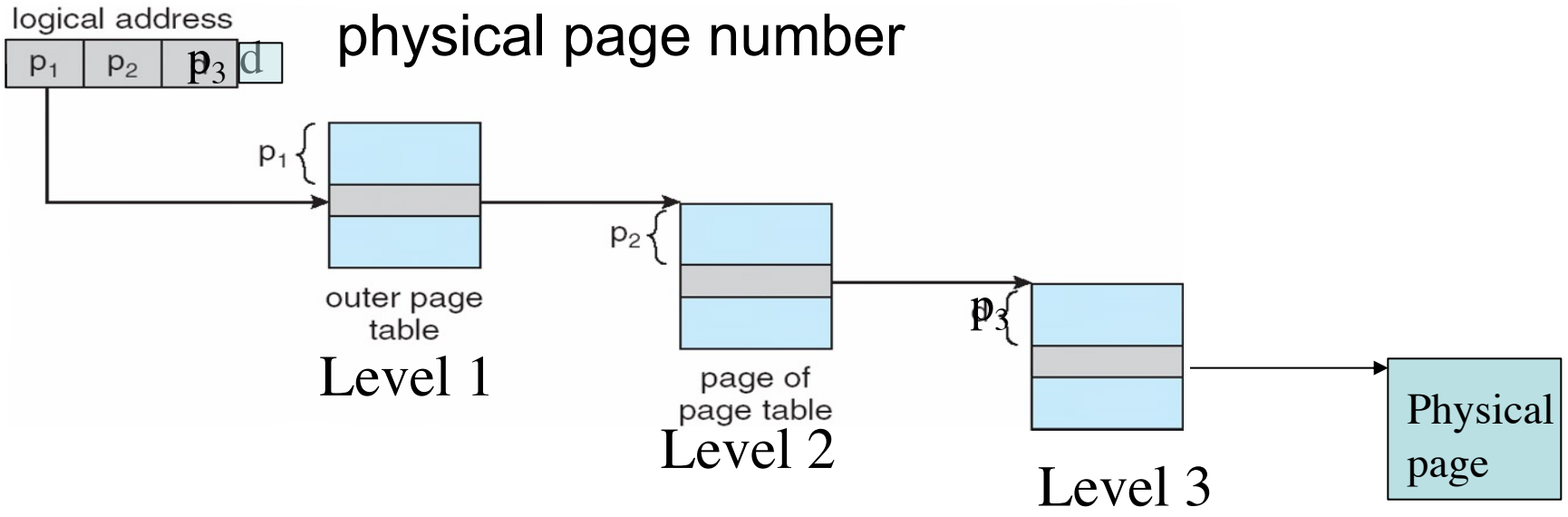


4-level page table



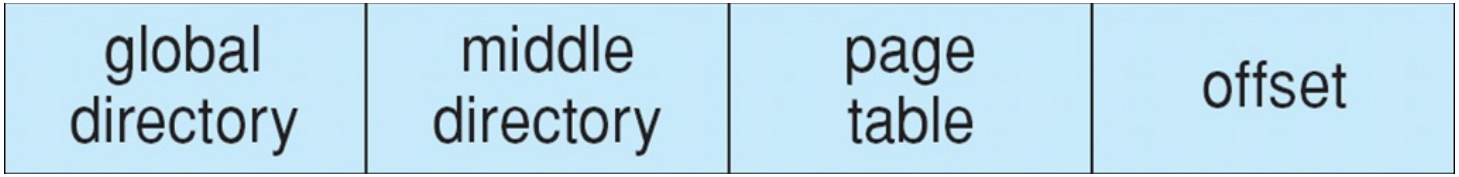
Address translation with 3-level paging

- **Virtual Address is decomposed as 4 parts**
  - p_1, p_2, p_3, d
  - Use p_1 to access an entry at level 1 table. Content is location of level 2 table
  - Use p_2 to access level 2 table. Entry content is location of level 2 table
  - Use p_3 to access level 3 table . Entry content is physical page number



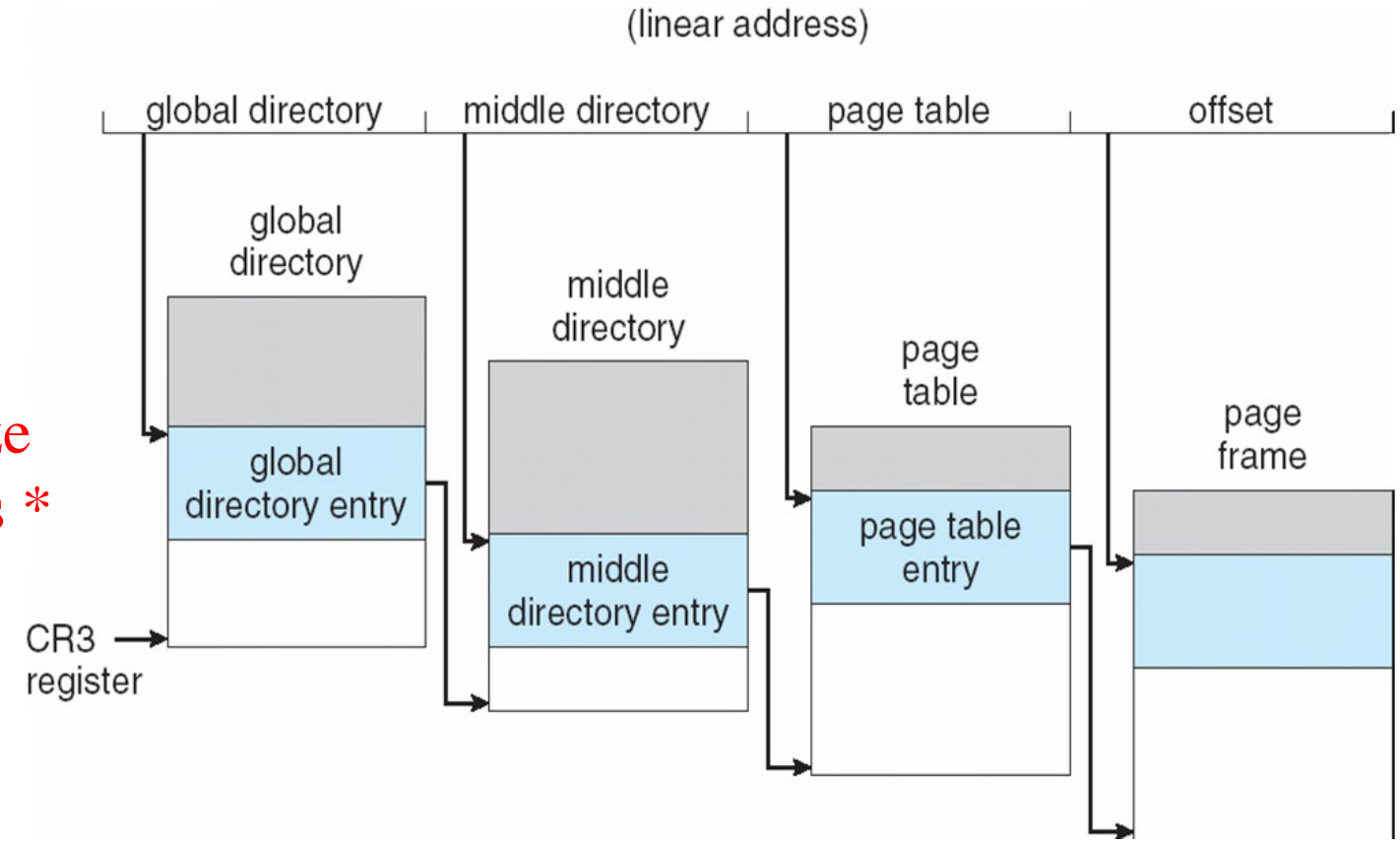
Three-level Paging in Linux

Address is divided into 4 parts



Why 3-level? Larger memory space

Virtual space size =# of tree leaves * page size



Summary

- **Memory is a resource that must be multiplexed**
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- **Simple protection through segmentation**
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well
- **Page tables**
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address
  - Page sharing among processes is easy
  - 1-level vs multi-level page table
- **TLB**
  - Essential for speeding up address translation