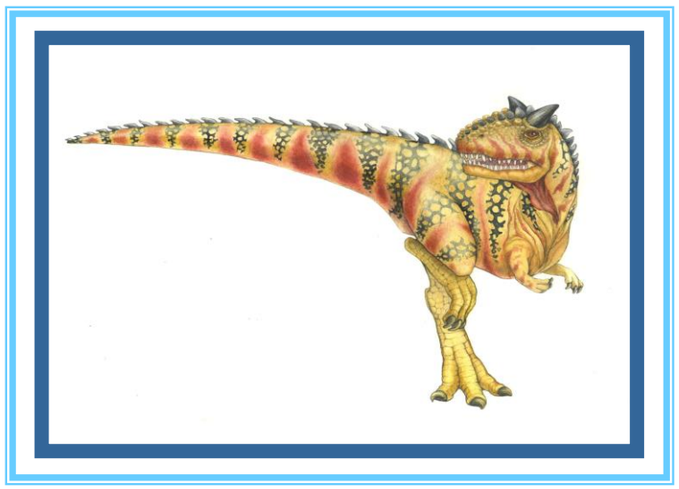
Chapter 8: Memory Management Strategies



Introduction

- “I cannot guarantee that I carry all the facts in my mind. Intense mental concentration has a curious way of blotting out what has passed. Each of my cases displaces the last”.

  - “The Hounds of Baskervilles” by Sir Arthur Conan Doyle.

Introduction

- We have seen in earlier chapters that the CPU can be shared by a set of processes.
- This is called “multiprogramming” and is a key OS feature.
- What do we gain from this?
  - We can improve the utilization of the CPU.
  - We can improve the speed of a computer’s response to its users.
- To do this, however, the price we pay is that: we keep several processes in memory.
- In other words, several processes share memory.
- Clearly, memory management becomes more complex.
- This is clearly more overhead for the operating system.
- In this chapter, we will discuss various techniques to manage memory.
- These techniques require appropriate hardware support.

Chapter 8: Memory Management

- Chapter outline:
  - Background/introduction.
  - Swapping.
  - Contiguous Memory Allocation.
  - Paging.
  - Structure of the Page Table.
  - Segmentation.
  - Example: The Intel Pentium and ARM.

Objectives

- To provide a detailed description of various ways of organizing memory hardware.
- To discuss various memory-management techniques: paging and segmentation.
- Case study: the Intel Pentium processor, which supports both pure segmentation and segmentation with paging.

Background

- Memory consists of a large array of words/bytes, each with its own address.
- The CPU fetches instructions from the memory according to the program counter value.
- These instructions may cause additional loading from and storing to specific memory addresses.
- **An instruction execution cycle.**
  - First, fetch an instruction from memory.
  - Instruction is then decoded and may cause operands to be fetched from memory.
  - After the instruction has been executed on the operands, results may be stored back in memory.
- We have a sequence of memory addresses in which a process is stored and executed. What is this called?
- Address space.

Basic Hardware

- CPU can directly access only: the main memory and the registers.
  - Note that it cannot directly access the disk memory.
- Hence, programs in disk need to be brought into main memory before they can be executed.
- Any instruction to be executed, and any data needed by the instructions, must be in one of the above direct access storage devices.
- Most CPUs can execute simple instructions stored in the registers at the rate of ≥ 1 per clock cycle.
- Completing a memory access, however, can take many cycles (roughly 100 cycles).
  - This results in the “stalling” of the CPU.
- The remedy is to add fast memory between the CPU and main memory.
  - This memory buffer is called a cache.
  - More on this later.

Base and Limit Registers

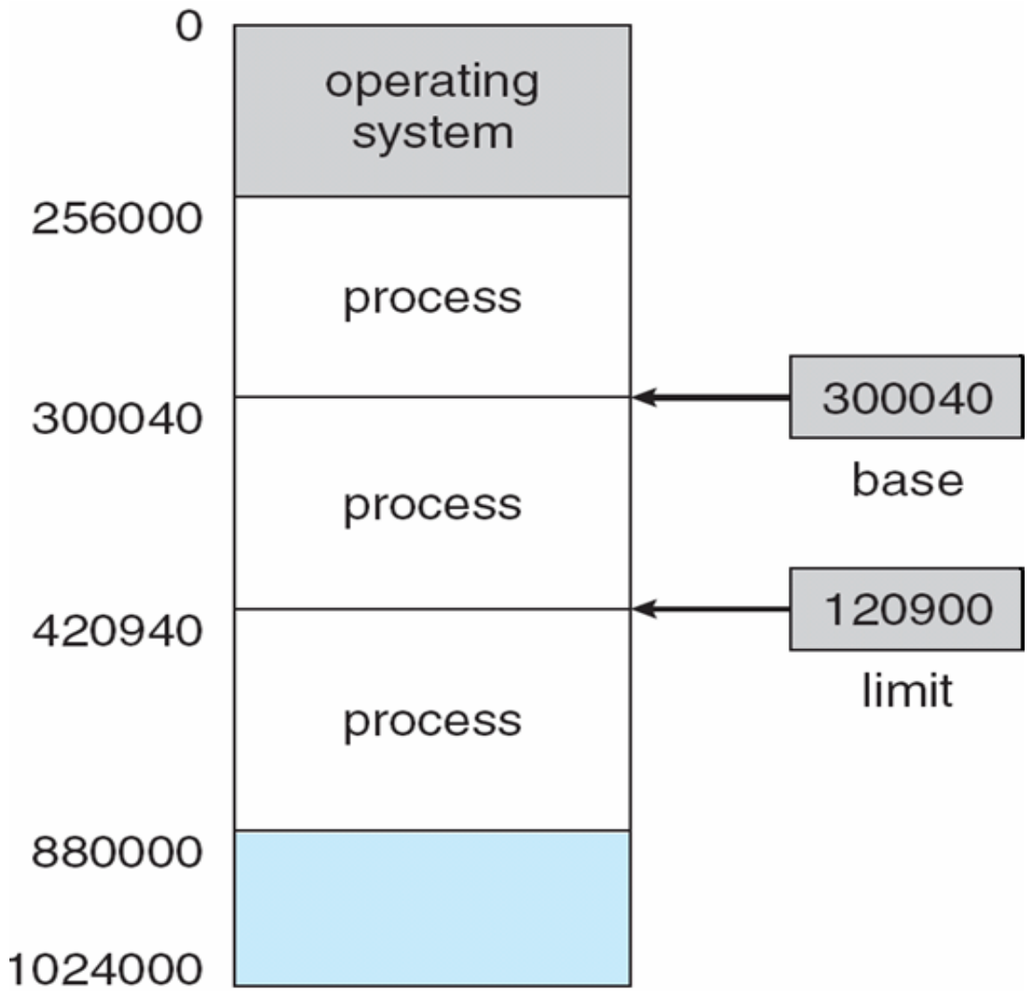
- Some concerns regarding memory accesses, besides access time:
  - Ensure correct operation to protect the OS from access by user processes.
  - Also, need to protect user processes from one another.
- Solution to ensure protection?
- Ensure that each process has a separate memory space.
- The process should be able to access only (the range of) memory addresses in its space.
- Note that this is a hardware solution (& OS does not intervene between CPU & it's memory accesses, due to performance overhead).
- Two registers are used for a process:
  - Base register: hold the smallest legal physical memory address.
  - Limit register: specifies the size of the range of the legal memory addresses.
- If base register holds 300040 and the limit register is 120900, then the process can legally access all addresses in the range (300040, 4209390).
- OS memory manager allocates the address space.

Protection

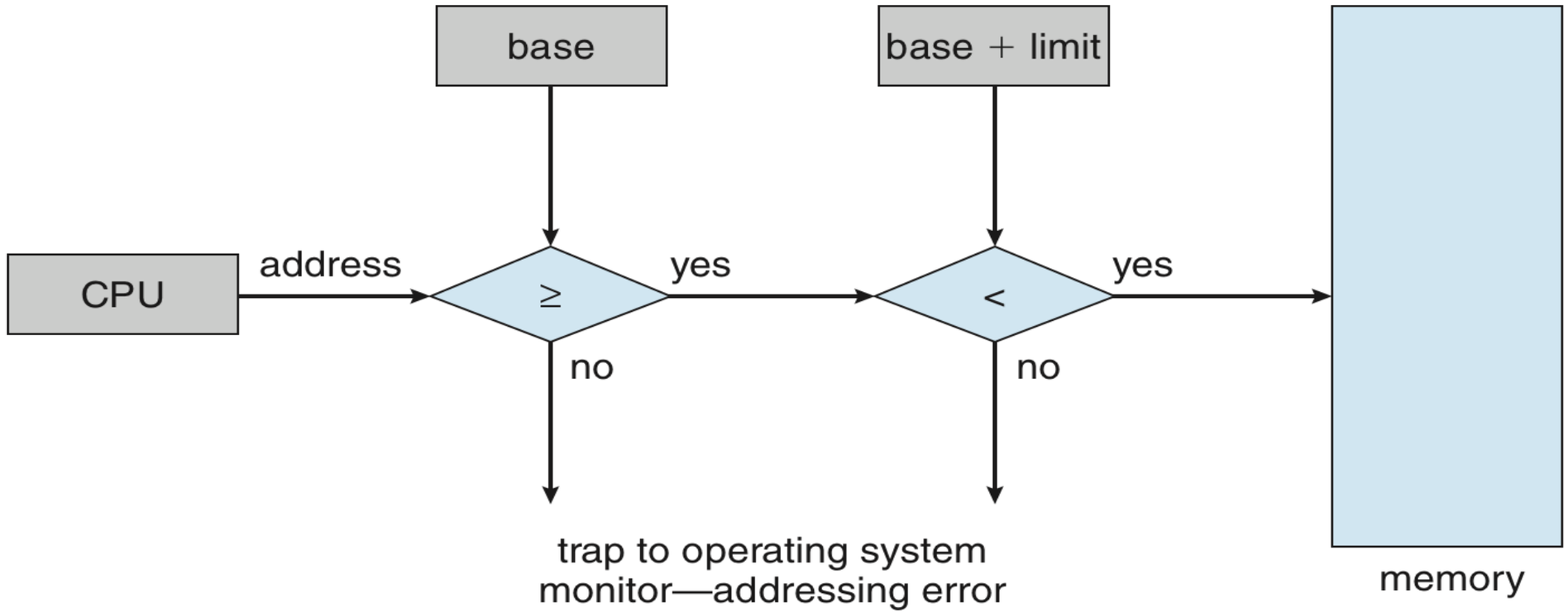
- How do we ensure protection of a process address space?
- We compare every address generated in user mode with these registers.
- Any attempt by program executing in user mode to access OS memory or other users' memory results in?
  - A trap, which is treated as an (addressing) error.
- Base and limit registers can only be loaded by the OS, which uses a special privileged instruction.
  - Privileged instructions only run in kernel mode.
  - Only the OS executes in kernel mode.
- Hence, the OS can change the register values, but prevents user programs from changing the register contents.
- OS is given unrestricted access to both kernel & user memory.
  - This allows it to: load user programs into user memory, dump out the programs in case of errors.

Base and Limit Registers

- A pair of **base** and **limit** registers define the address space.



Hardware Address Protection with Base and Limit Registers



Logical and Physical Addresses

- A process will have an address space allocated to it (1500, 2500). This is a nice, contiguous block of memory.
- Let us call this the logical/virtual address space. This is the space that the process can use.
- Now, this may not necessarily map to (1500, 2500) in the physical memory all the time.
- So, it becomes necessary for the OS to map this logical address space → a physical address space.
- The mapping is done by a hardware device called the memory management unit (MMU).
- This mapping is done at run-time.
- How this is done is what this chapter is all about.
- Logical (virtual) address: address generated by the CPU.
- Physical (real) address: address seen by the memory unit (actual address).

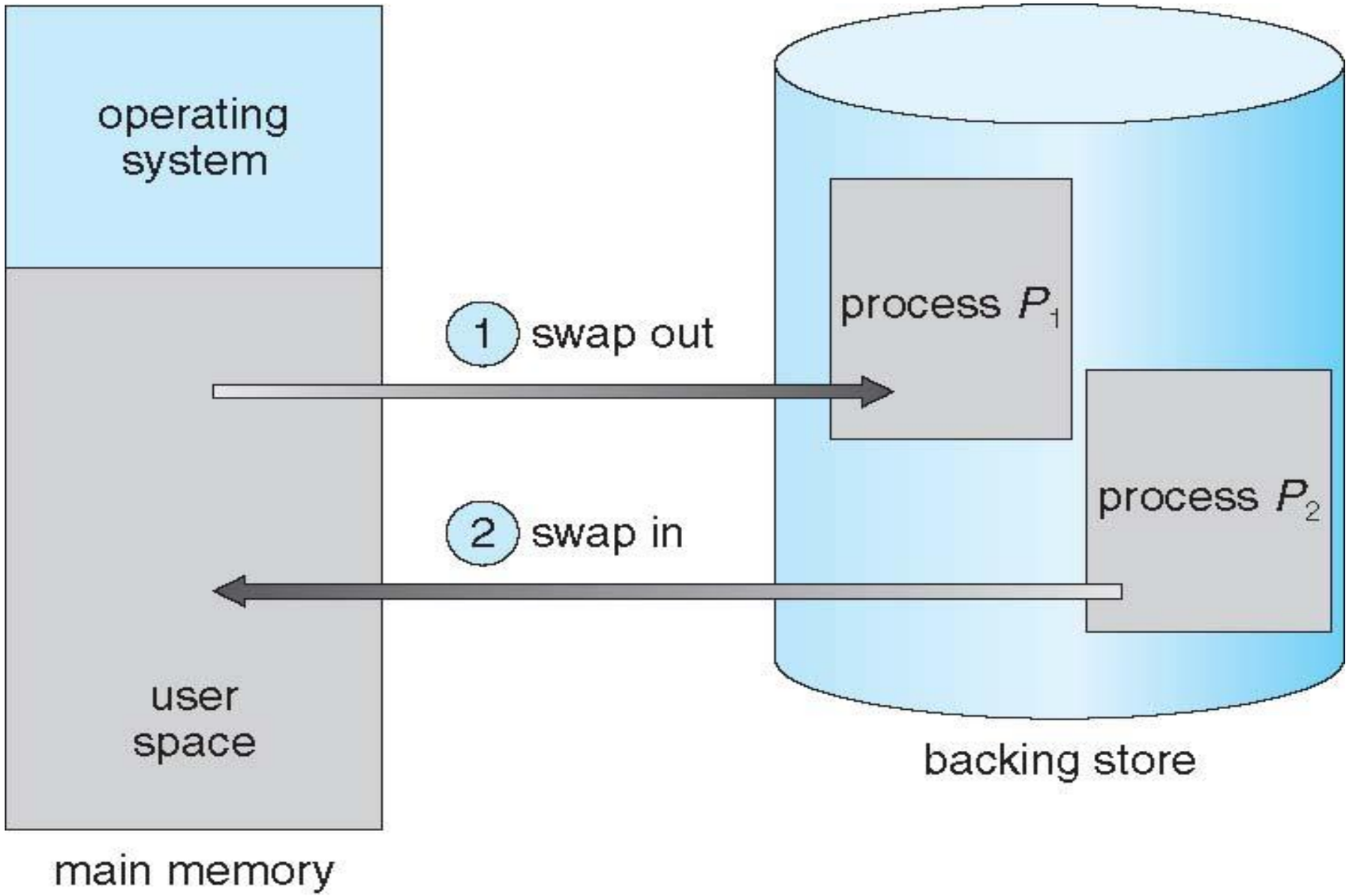
Process Relocation

- We would like to be able to swap processes in and out of main memory.
  - Why swapping?
- Once a program is swapped out, it is very limiting to have the restriction that it should be swapped back into the same memory region as before.
  - That region may have been taken by another process.
- Hence, there is a need to relocate processes to different memory regions from time to time.

Swapping

- Is physical memory large enough to store all the processes?
- May have 50-100 process even before user starts a process!
  - Check apps for updates, check for email.
- Hence, main memory becomes overloaded. How to support all these processes?
- One solution: bring each process in it's entirety, run it for a while, & then, put it back on disk (to make way for another process).
- This is called swapping. Which processes to swap?
- Another solution: virtual memory (next chapter).
- Set of processes in ready queue, scheduling algorithm picks order.
- Say, scheduler picks process P. If it is not in memory & if there is no memory region big enough, swap out a current process from memory -> disk & swap in the scheduled process.
- Processes are swapped out and swapped in from/to memory.

Schematic View of Swapping



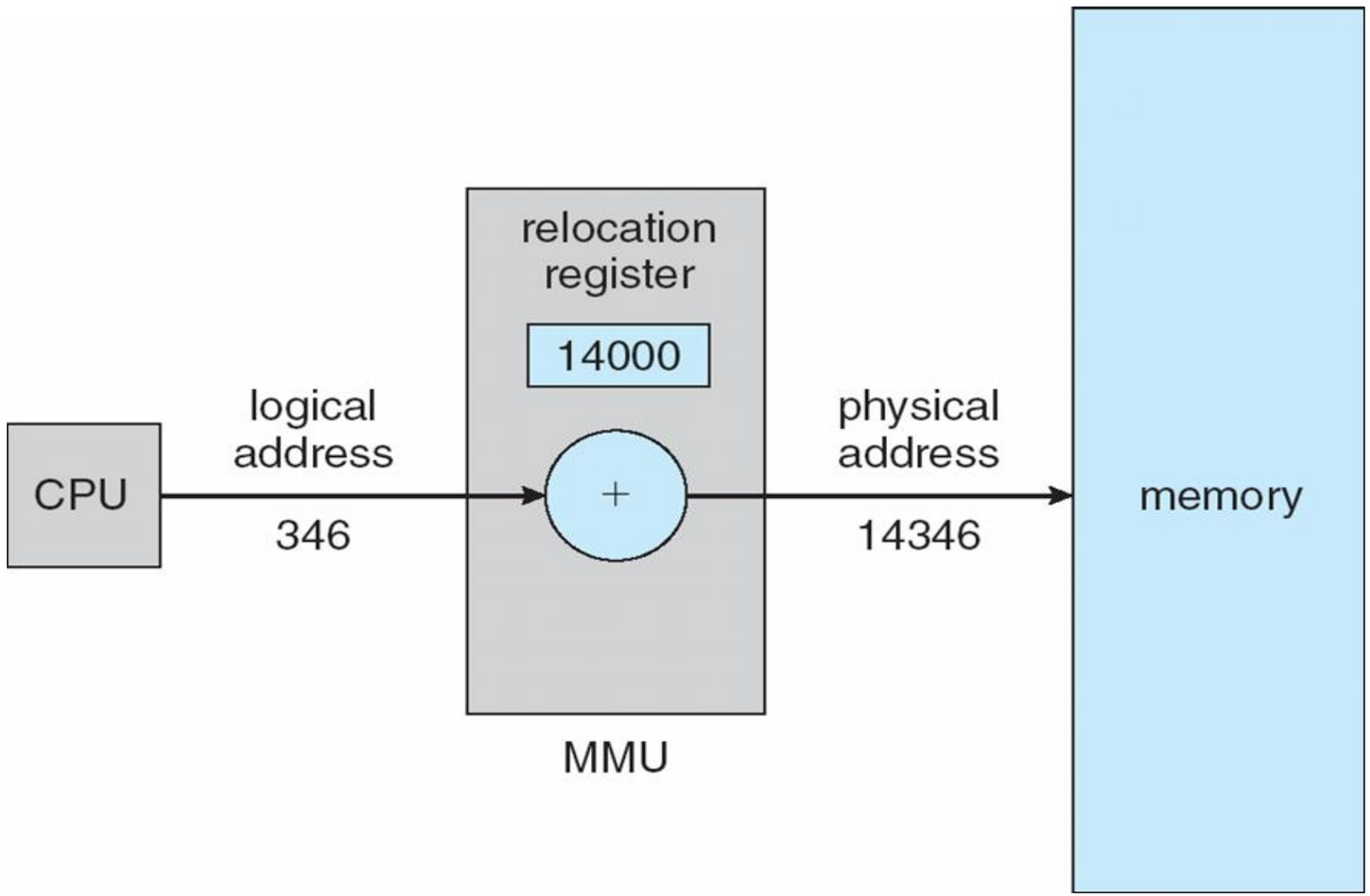
Context Switch Time including Swapping

- A performance concern: the context switch time can be very high.
- Consider a 100 MB process swapping to hard disk with transfer rate of 50MB/sec.
  - Transfer (swap) to or from main memory takes = 100/50 = 2 sec.
  - What is the total swap time?
  - It is 4 seconds (2 sec. each for swap in and swap out).
- Of course, transfer time is proportional to the amount of memory swapped.
  - Lets day we have 4 GB of main memory and that the OS takes up 1 GB.
  - The amount available for user processes is, then, 4 – 1 = 3 GB.
  - Many processes are smaller than this, say 100 MB.
  - A 175 MB can be swapped out in how many seconds?
    - 3.5 seconds.
  - A 3 GB process will take how many seconds?
    - 3000/50 = 60 seconds.

How is address relocation implemented?

- This process is called dynamic relocation of addresses.
  - Virtual address is relocated to a physical address.
- Let's discuss a simple relocation scheme modeled on the base register concept.
- We can have a variation of the base register: called a relocation register.
- The value in the relocation register is added to every virtual address.
- For example, if the base is at 14000, then an attempt by the user to address virtual location 0 is dynamically relocated to 14000.
- Similarly, an attempt to access 346 is mapped to 14346.
- The user program never "sees" the real, physical addresses.
- **Physical address is abstracted by the OS memory manager.**
- We now have two types of addresses: virtual addresses (0, max) and physical address (R + 0, R + max).
- Next slide shows a pictorial representation.

Dynamic relocation using a relocation register



Contiguous Memory Allocation

- Goal: allocate memory to processes efficiently.
- One early solution proposed –> contiguous memory allocation.
- Main memory usually consists of two partitions:
  - Resident operating system, usually held in low memory locations.
  - User processes then held in high memory.
  - Here, each process is contained in single contiguous (together in a sequence) section of memory.

Contiguous Memory Allocation

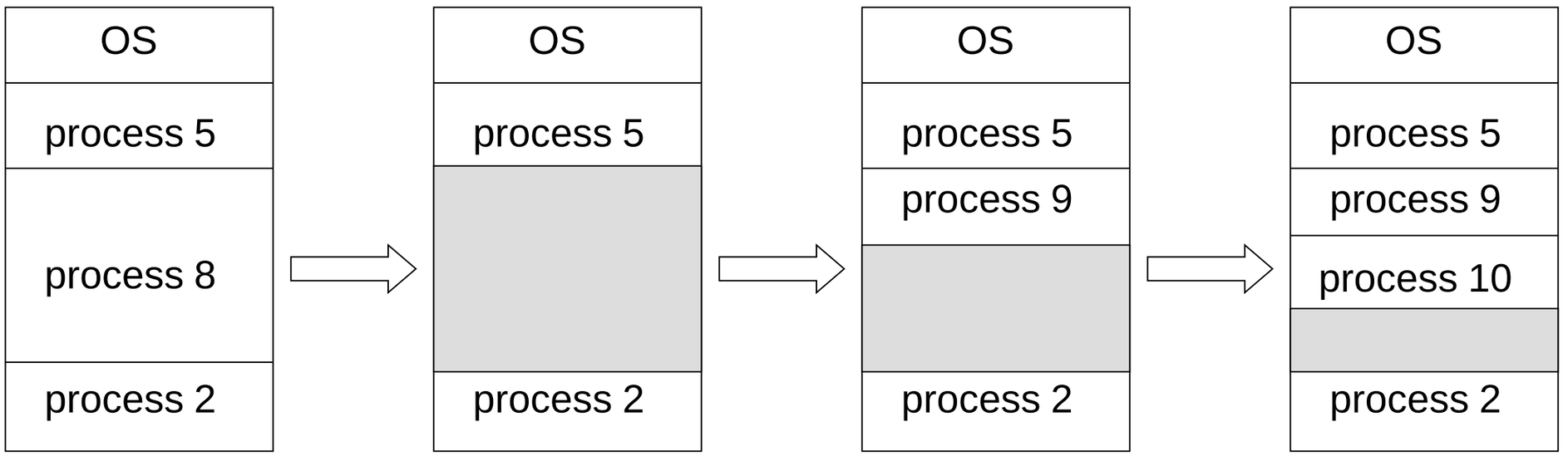
- One of the simplest methods for allocating memory to processes is to divide memory into several fixed size partitions.
- Used by IBM OS/360, in the 1960s.
- Each partition may contain only one process.
- When a partition is free, a process is selected from the input/ready queue and is loaded into the free partition.
- Disadvantage of this approach?
- Another scheme employs variable size partitions.
- Keep a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is one large block of available memory, or a hole.
- Eventually, after many allocations and de-allocations, memory contains a set of holes (and processes) of various sizes.

Memory Allocation (continued)

- At any given time, we have a list of available block sizes and an input queue of ready processes.
- The OS can order the input queue according to a scheduling algorithm, say Shortest Job First.
- Memory holes are allocated to processes **based on some algorithm** until, finally, the memory requirements of the next processes cannot be satisfied.
  - There is no hole large enough to hold the process.
- OS can wait till a large enough hole becomes available.
- Or, it can skip down the input queue to see whether some other smaller process can fit in.
- This is the dynamic storage allocation problem.
  - How to satisfy a request of size *n* from a list of free holes?
- Several algorithms exist.

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions.
  - Hole – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Process exiting frees its partition, adjacent free partitions may be combined.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first fit search ended.

- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Advantage of this algorithm?
  - Produces the smallest leftover hole.

- **Worst-fit**: Allocate the *largest* hole; must also search entire list, unless sorted by size.
  - Produces the largest leftover hole (may be more useful than the smaller left over hole from best fit).
- Let us look at an example problem now.

What about the Performance?

- Storage Utilization?
  - Would be best for best-fit and worst for worst-fit.
  - First fit would be between the two.
- Speed?
  - First fit would be fast vs. the other two.
- Tradeoff between utilization and speed.

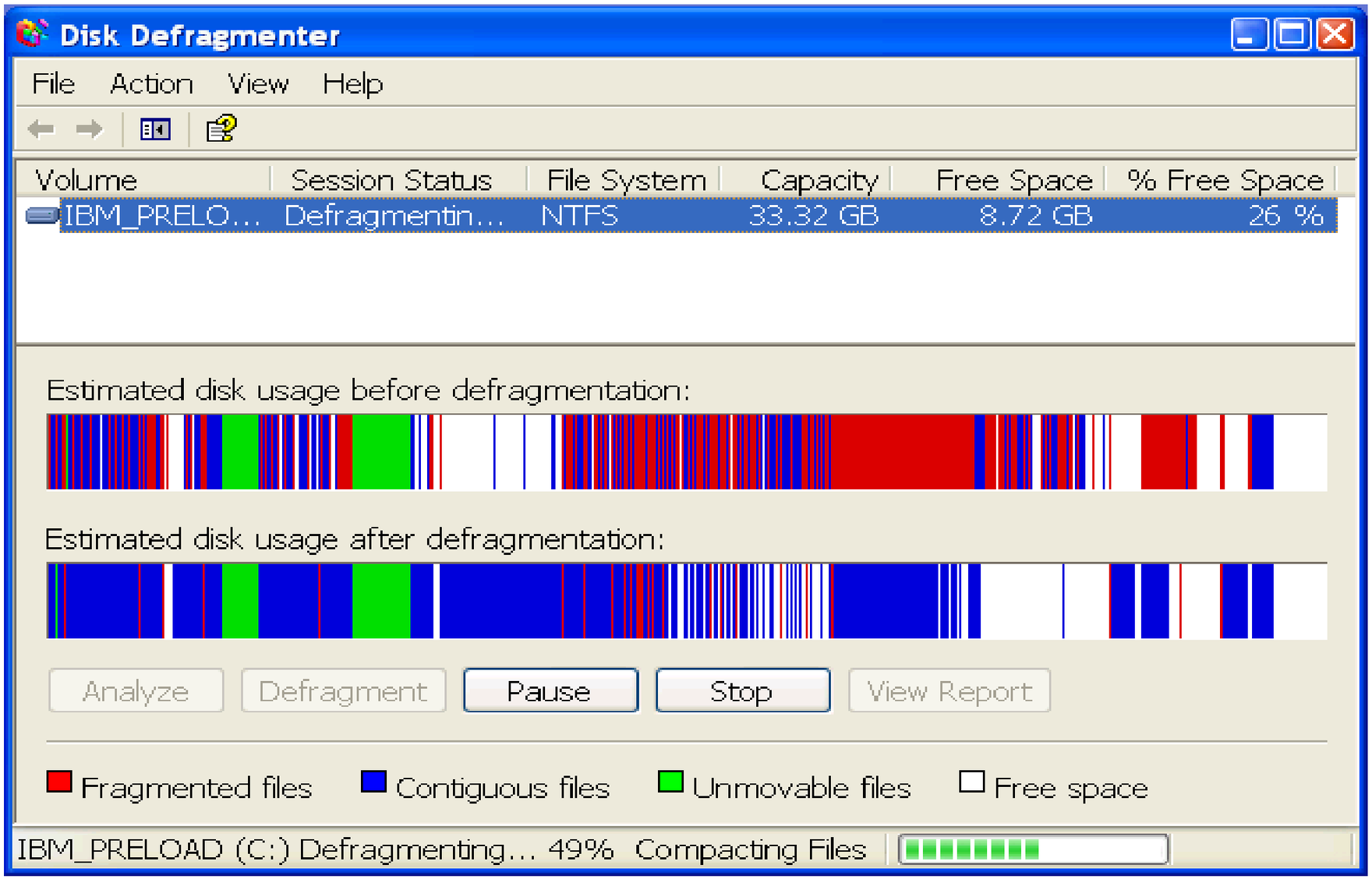
Fragmentation

- Initially, all the memory is one big contiguous hole.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
  - Holes of various sizes.
- This can be a problem.
  - We might have enough memory space to satisfy a process request, but the space may not be available as one contiguous block.
  - Can have a large number of small holes.
- This is called external fragmentation (external to a partition).
- Worst case → one small free block between every two processes!
- Fragmentation can also be internal (to a partition).
- Consider a hole of 18,464 bytes.
- Lets say we allot this to a process of 18,462 bytes.
- Now, we have fragmentation within a partition.
  - A 2 byte hole in the partition.

Solving the Fragmentation Problem

- Fragmentation refers to the building up of holes which individually are not of much use in allocation to processes.
- How to solve fragmentation?
- **Solution 1**: compaction.
- Shuffle the memory contents so as to place all free memory together in one large block.
- Windows XP's disk defragmenter on next slide.
- An expensive operation, especially for disk (much slower access).
- **Solution 2:** permit the address space of a process to be non-contiguous.
  - This allows a process to be allocated physical memory "**wherever**" it is available.
- Technique that achieves this is called paging.

Windows XP's Disk Defragmenter



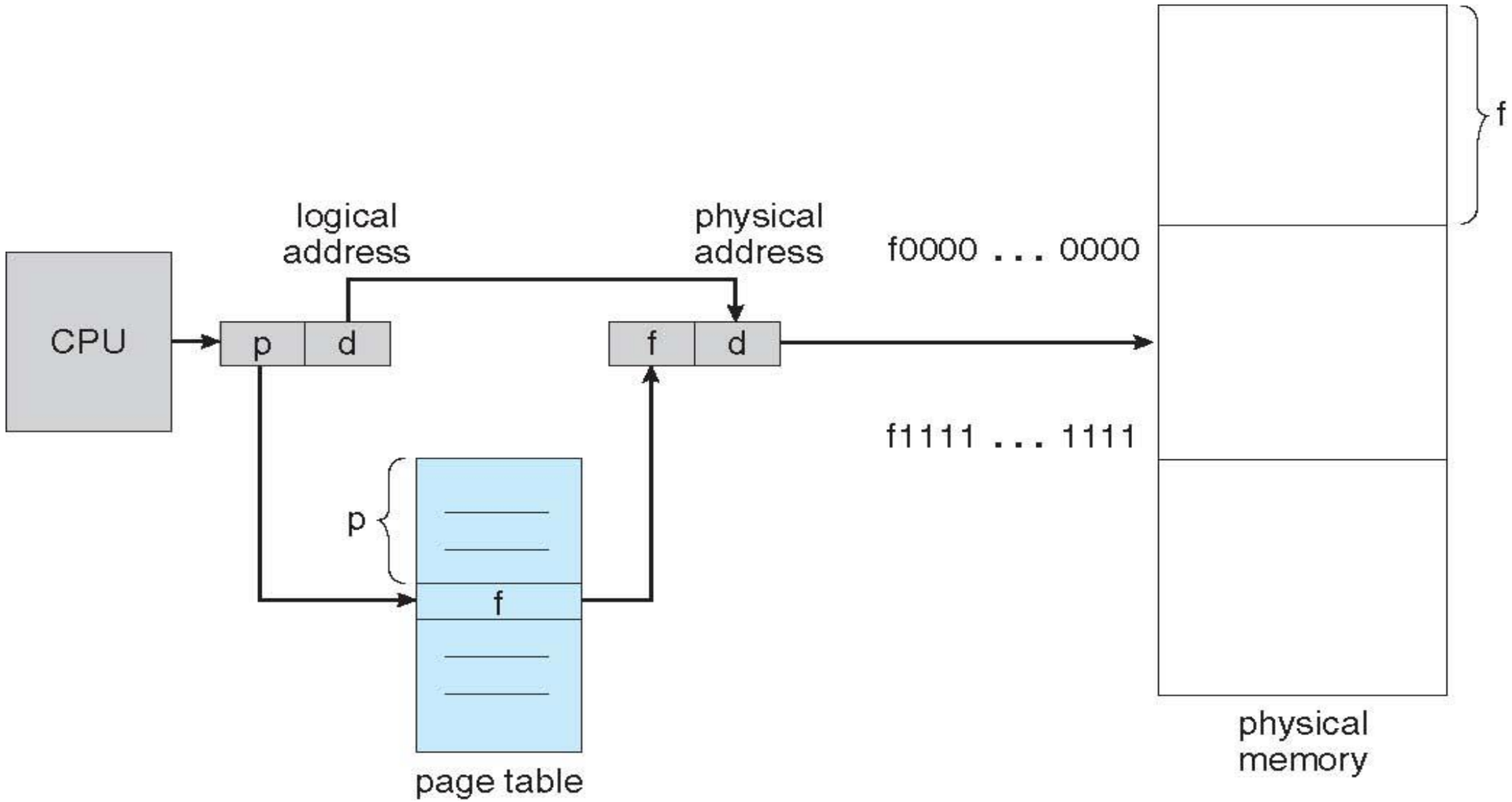
Paging

- Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous.
- In doing so, it prevents (external) fragmentation and leads to better memory utilization.
- The scheme works as follows:
  - Physical memory is broken into fixed size blocks called **frames**.
  - Logical/virtual memory is broken into blocks of the *same size* called **pages**.
- When a process is to be executed, its pages are loaded into any available memory frames.
  - May be non-contiguous.
- The disk (virtual addresses) is also divided into fixed size blocks that are of the same size as the frames (physical addresses) in main memory .

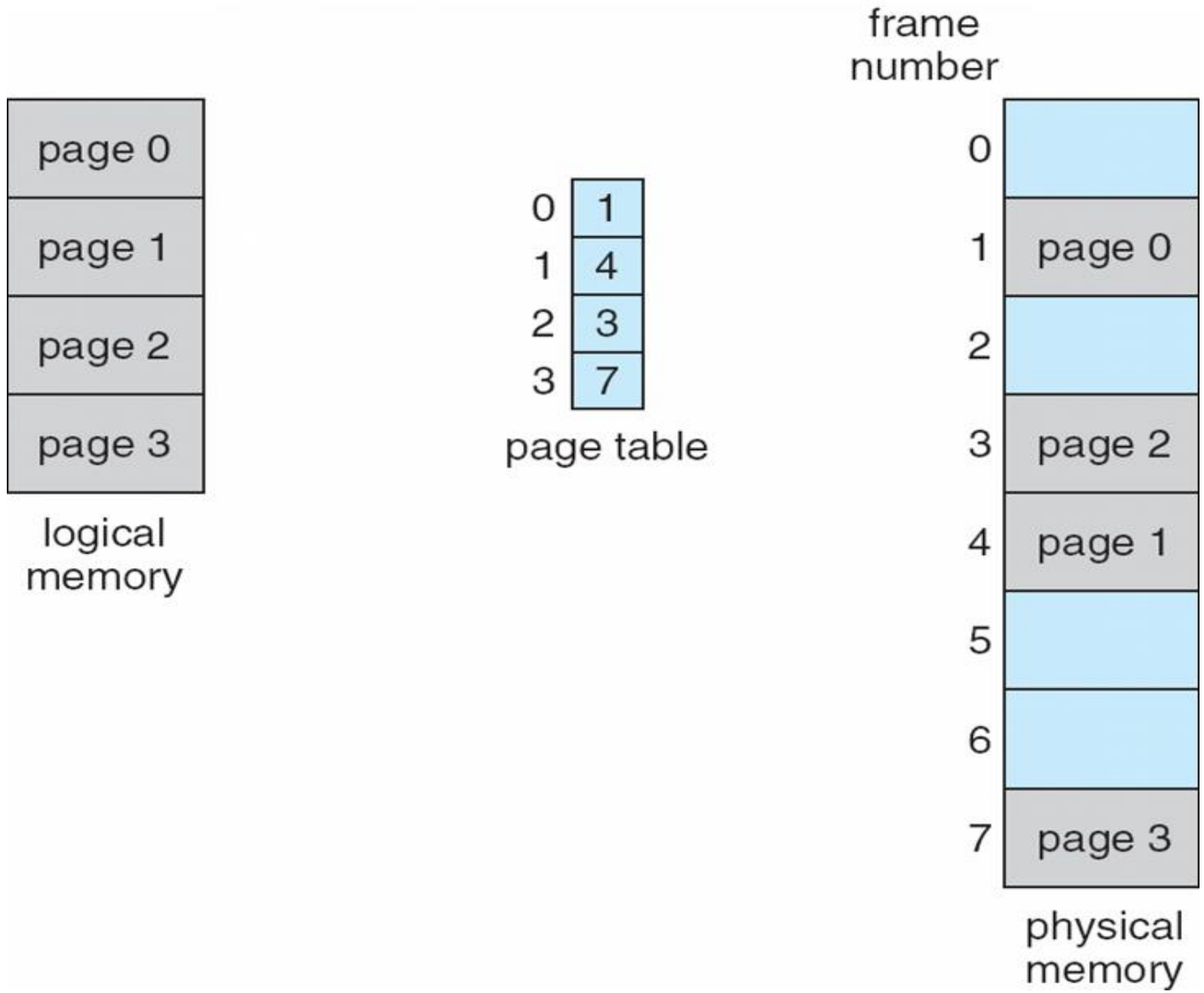
Paging Hardware

- Let's see the paging hardware diagram on the next slide.
- Every logical/virtual address is divided into two parts:
  - **A page number (p).**
    - This is used as an index into a page table.
    - The page table contains a list of base addresses for pages in the physical memory.
    - Each entry in the page table is the base address of a page in physical memory.
  - **A page offset (d).**
    - The offset value is added to the base address to define the physical memory address.
- Note that the offset values of each pair of (logical address, physical address) are the same. Why?

Paging Hardware



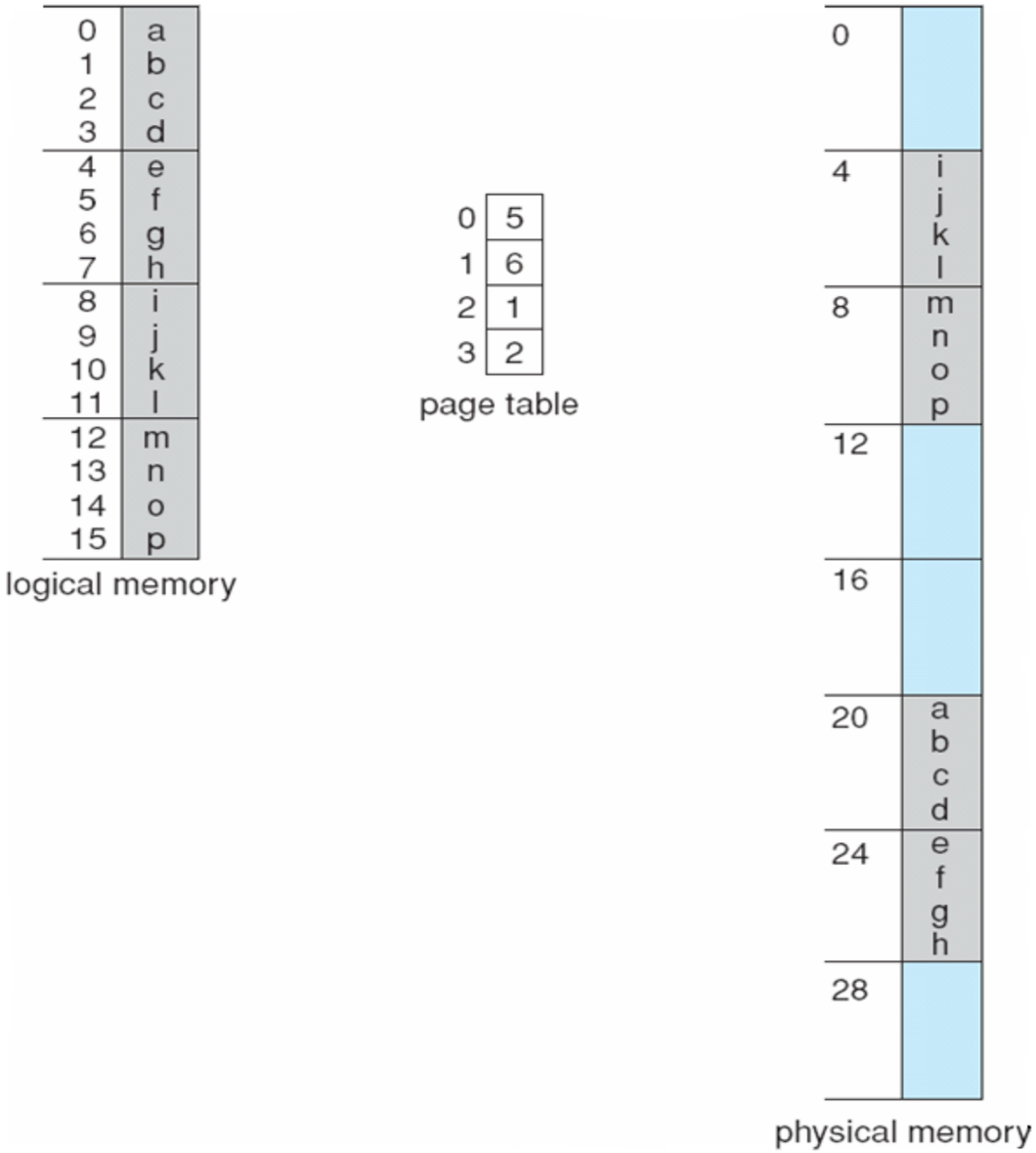
Paging Model of Logical and Physical Memory



A Simple Example.

- Lets say we have a physical memory of 32 bytes and a page size of 4 bytes.
- How many pages can we fit?
  - We can fit 32/4 = 8 pages.
- Lets assume that we have 4 pages (page 0 to page 3) and 8 frames (frame 0 to frame 7).
- Size of a page and a frame is?
- Now, logical address 0 (storing a) = page 0, offset 0. Looking at the page table, this maps to frame 5, offset 0. So, the physical address that logical address 0 maps to is = 5 x 4 = 20.
- Similarly, logical address 3 (storing d) = page 0, offset 3. Hence, it maps to physical address = (5 x 4) + 3 = 23.
- Logical address 4 (storing e) maps to physical address = (6 x 4) + 0 = 24.
- Logical address 13 (storing n) maps to physical address = (2 x 4) + 1 = 9.

Paging Example



32-byte memory and 4-byte pages

Page Table

- Lets say that the size of the logical address space is 2^m bytes and the size of a page is 2^n bytes.
- What were the values of *m* and *n* in the previous example?
  - *m* = 4 and *n* = 2.
  - Logical address space was = 2^4 = 16 bytes and page size was 2^2 = 4 bytes.
- Now, the higher order *m* – *n* bits of a logical address designate the page number and the *n* low order bits designate the page offset.
  - In our example, *m* – *n* = 4 – 2 = 2. So, the 2 high order bits represent the page numbers: 00, 01, 10, 11 (4 pages).
  - Similarly, *n* = 2. So, the 2 low order bits represent the page offset. Within a page, the offset values can be 0, 1, 2, 3.
- How to find page size on your system?
- On a UNIX based system, type: `getconf PAGESIZE` on the command line.
- Can also use the `getpagesize()` system call.

Address Translation Scheme

- Logical address is divided into:
  - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	*d*
m - n	*n*

- For given logical address space 2^m and page size 2^n

Paging & Fragmentation

- Paging ensures that we do not have external fragmentation. How?
- External fragmentation occurs because of the restriction of allotting processes to contiguous memory holes.
- In paging, that restriction is relaxed.
  - It is all right for a process to be assigned memory that is non contiguous.
  - Any free frame can be allotted to a process that needs it.
- What about internal fragmentation (within a partition)? Can we still have that?
- Yes.
- Frames are allotted as units.
- How much wastage in terms of number of frames can we have for a process?
- If the memory requirements of a process do not coincide with the page boundaries, then the last frame allotted may not be full.

Paging & Fragmentation

- Lets say that the page size is 2,048 bytes (2KB).
- How much memory will a process of 72,766 bytes need?
  - 72766 = (35 x 2048) + 1086.
  - This is 35 pages and 1086 bytes.
  - In the 36th page, 2048 – 1086 = 962 bytes are unused.
- This is internal fragmentation.
- What about the worst case?
  - A process would need *n* pages + 1 byte.
  - Would need to allocate *n* + 1 frames.
  - Wastage of almost 1 frame.

Virtual – Physical Address Translation

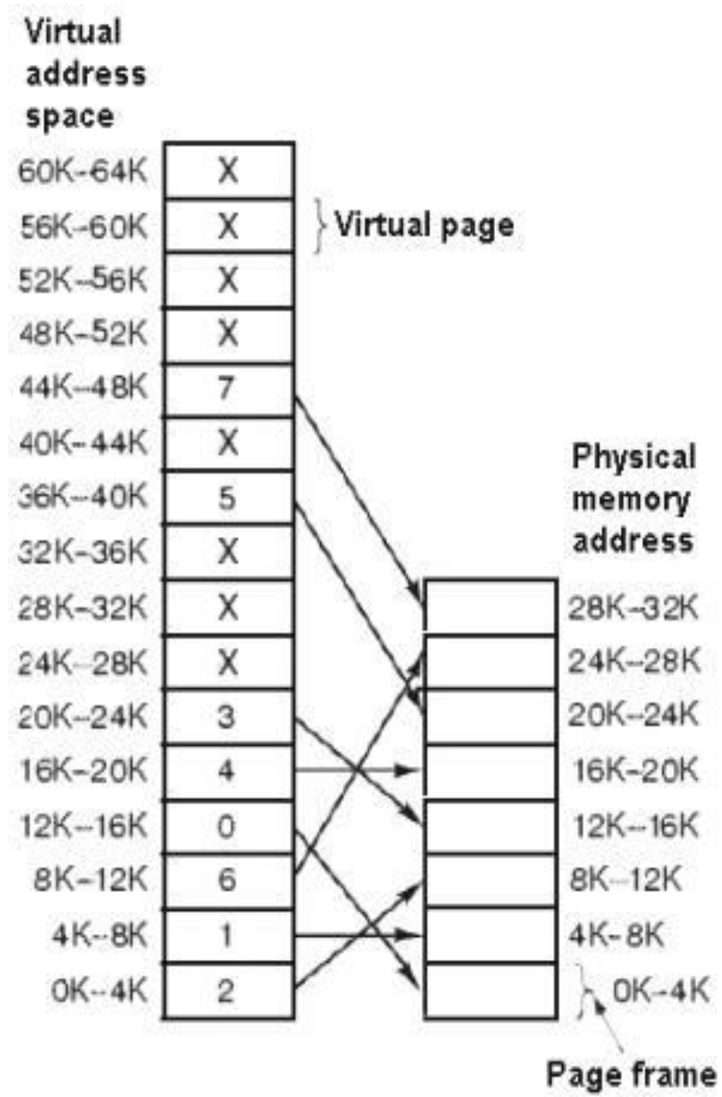


Figure 2. The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K to 12K means 8192-12287.

Use of a Page Table in Address Translation

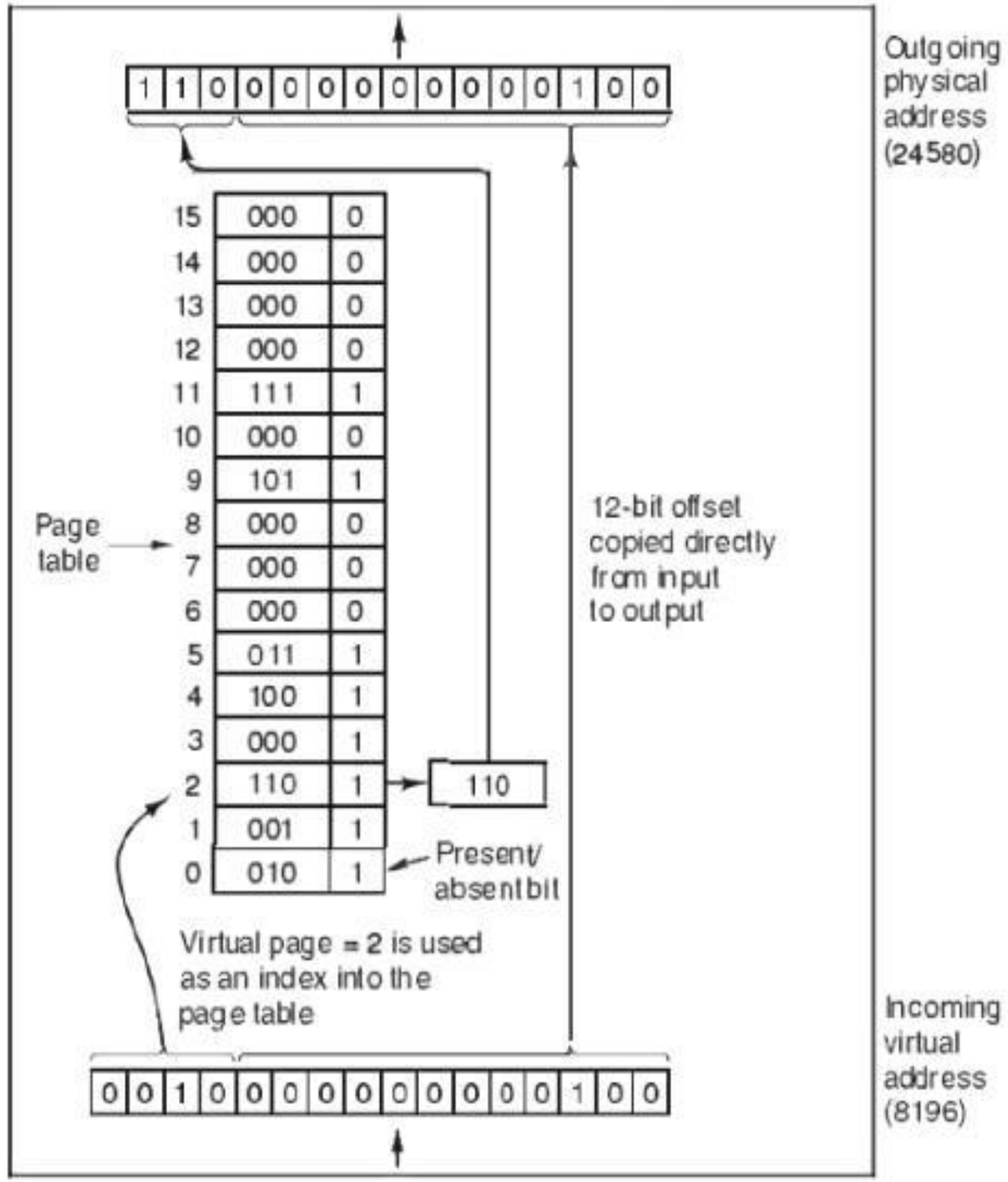
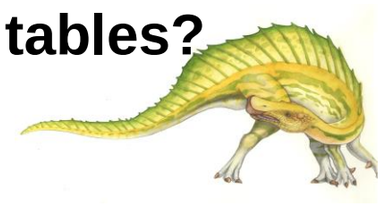


Figure 3. The internal operation of the MMU with 16 4-KB pages

Some Exercises

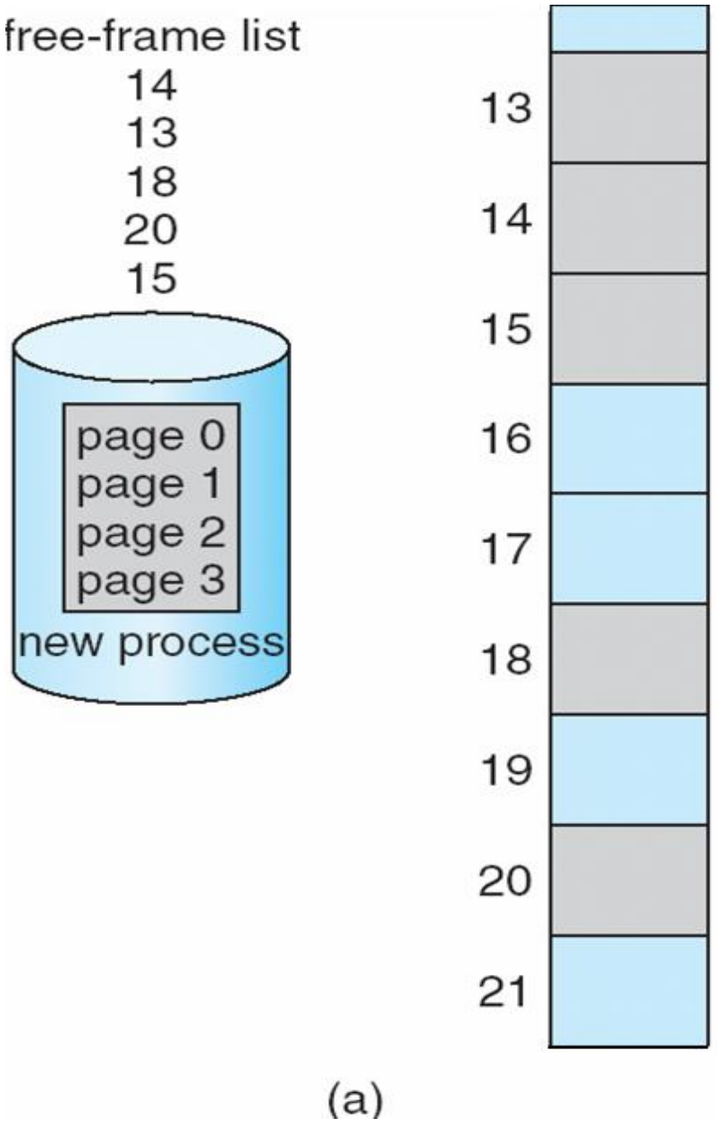
- Assuming a 1-KB page size, what are the page numbers & offsets for the following address references (provided as decimal numbers)?
  - 2357.
  - 256.
- Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
  - How many bits are required in the logical address?
  - How many bits are required in the physical address?
- Consider a computer system with a 32-bit logical address and 4-KB page size. How many entries are there in the page table?
- A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8-KB. How many entries in the page table?
- A machine has 32-bit address space & 8-KB pages. Page table has one 32-bit word per entry. When process starts, it's page table is loaded, at 1 word every 100 nsec. If each process runs for 100 msec (including the time to load the table), what fraction of the CPU time is devoted to loading the page tables?



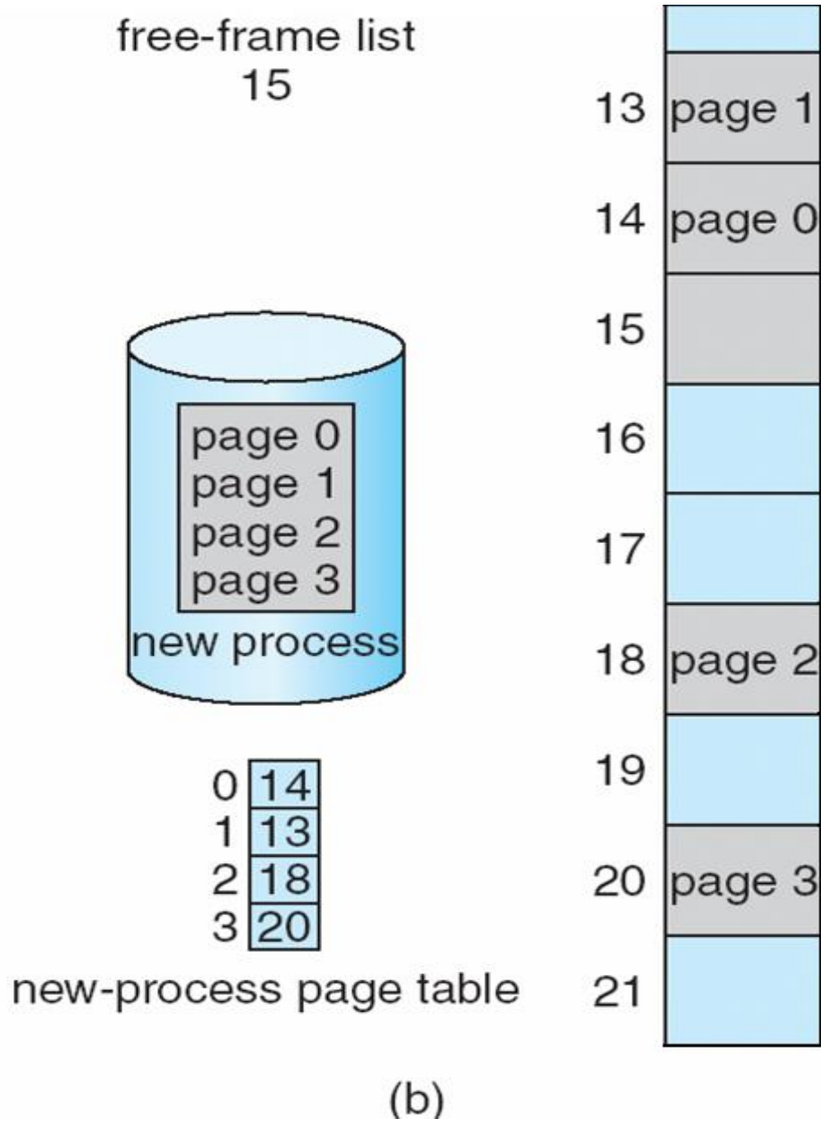
Allocation of Frames

- How are pages allocated to frames?
- When process arrives in system to be executed, its size, expressed in pages, is examined.
- Requirement of *n* pages → *n* free frames.
- If frames are available, they are allocated to the process.
- Two steps need to be performed:
  - (i) the 1st page is loaded into one of the allocated frames, and (ii) the frame number is added in the page table for this process.
- Same procedure is repeated for the other **{page, frame}** combinations.
- Note that the OS must be aware of the allocation details of the physical memory.
  - Which frames are available?
  - Which frames are allocated and to which process?
  - A frame table (one entry per frame) that provides this information.
- Frames allocation algorithms are discussed in the next chapter.

Free Frames



Before allocation



After allocation

Memory Abstraction

- Note the abstraction between the virtual (logical) memory and the real (physical) memory.
- The user program views memory as one single space.
  - In reality, the user's program is scattered around the real memory.
- This detail is hidden from the user.
- The abstraction/hiding is provided by the address translation layer and controlled by the OS.

Page Table Hardware Implementation

- A page table is allocated for each process.
- A pointer to the page table is stored in the process control block (PCB).
  - Page table values are stored and loaded as part of the context switch.
- How can the page table be implemented?
- The answer depends on the size (say, 256 entries) and number of page tables.
- If the page table is small (less number of entries), it could be stored in registers.
  - Fast access.
- E.g.: DEC PDP-11: address has 16-bits, page size is 8 KB. Page table has how many entries?
- What if the size of the page table is large (say, a million entries)?
  - Not feasible to implement as registers.
- Then, we need to store them in main memory.
- A register called page table base register (PTBR) points to the page table.
  - Value changed on every context switch. Why?

Problem with Page Table in Memory

- Do you see any problem with this approach?
- Note that now, both the page table and the frames are in main memory.
- How many memory accesses for each memory translation?
  - If we want to access location *i*, we must first index into the page table (location from PTBR register). This requires one memory access.
  - The page table provides us with the (address of) frame number, which is combined with the page offset to produce the actual address.
  - We can then access the desired place in memory. This is another memory access.
  - So, we need 2 memory accesses: 1 for the page table entry, one for the actual data.
- Each memory access is slowed down by a factor of 2!
- Can we do better?

Solution

- Use a small, fast hardware cache, called a translation look-aside buffer (TLB) .
- Each entry in the TLB has two parts:
  - A key
  - A value
- When the TLB is presented with an item, the item is compared with the keys.
- If the item is found, the corresponding value field is returned.
- All items can be searched simultaneously.
- The search is fast, the hardware expensive.
- Typically, the number of entries in the TLB is small: from 32 – 1024.
- TLB lookup on modern hardware is part of the instruction pipeline, adding essentially no penalty time wise.

TLB Usage

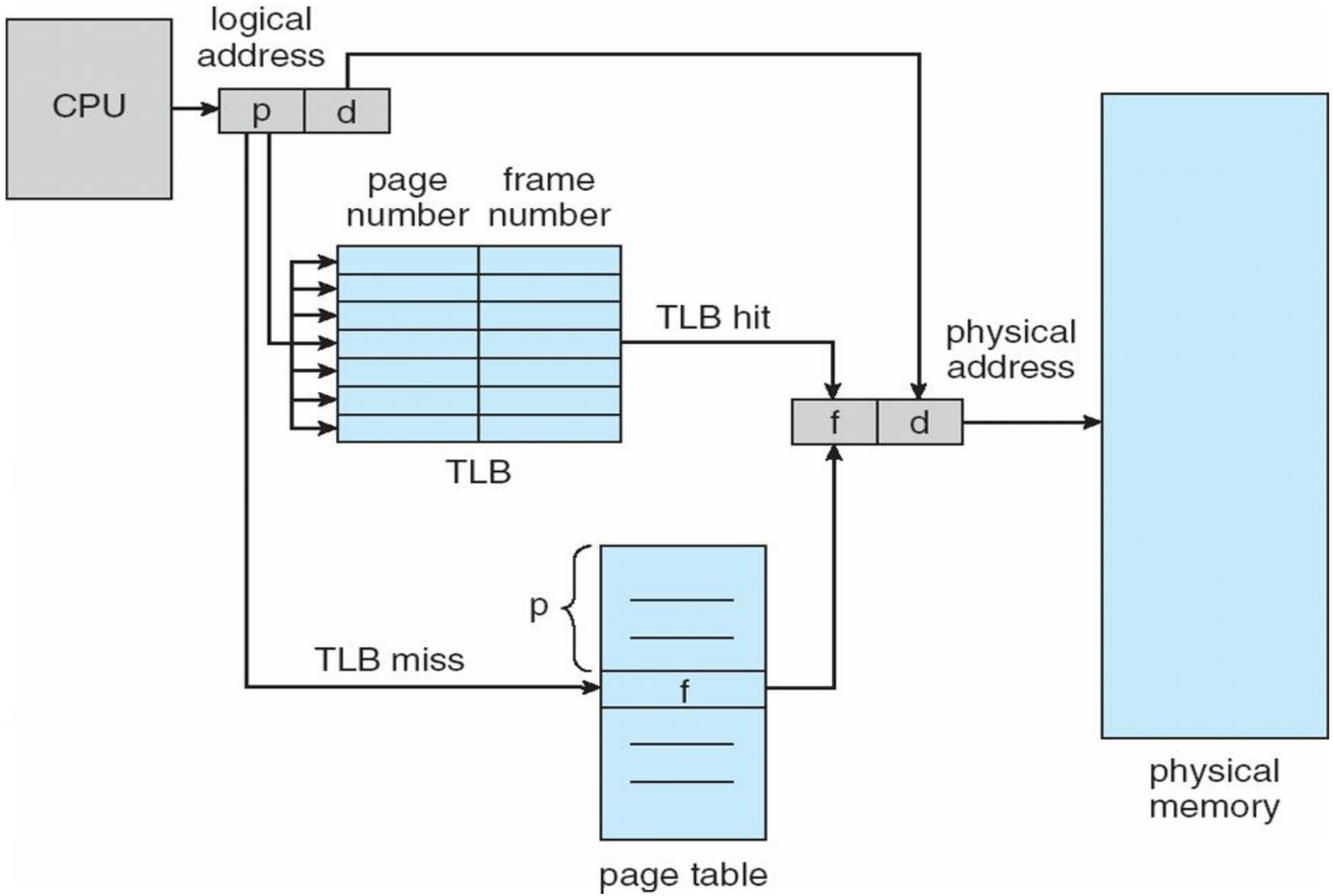
- The TLB contains a small number of page table entries. It is used with pages as follows:
  1. When a logical address is generated, its page number is presented to the TLB.
  2. If the page number is found in the TLB (a TLB hit), the frame number is immediately available and used to access memory.
  3. If the page number is not found (a TLB miss), a memory reference to the page table must be made. When the frame number is available, it can be used to access memory.
  4. In addition, we might add that frame to the TLB, if there is space.
     1. If there is no space, then we can evict/remove one TLB entry (let's say, least used).
- The % of time that *a particular* page number of interest is found in the TLB is called the hit ratio.
  - A hit ratio of 80 % means that the desired page number can be found in the TLB 80 % of the time.

TLB.

Page #	Frame #

- Address translation (p, d)
  - If p is in TLB, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



A Numerical Example

- Let's assume the following. :
  - Access to the TLB takes 0 nanoseconds.
  - It takes 100 nanoseconds to access memory.
  - In case of a hit, the access time is = 100 nanoseconds.
  - In case of a miss, the access time is = 100 + 100 = 200 nanoseconds.
- What is the effective mean access time (EMAT), assuming a hit ratio of 80 %?
  - EMAT = (0.80 x 100) + (0.20 x 200) = 120 nanoseconds.
- For a hit ratio of 98 %:
  - EMAT = (0.98 x 100) + (0.02 x 200) = 102 nanoseconds.
- Clearly, higher hit ratio translates to lower EMAT.

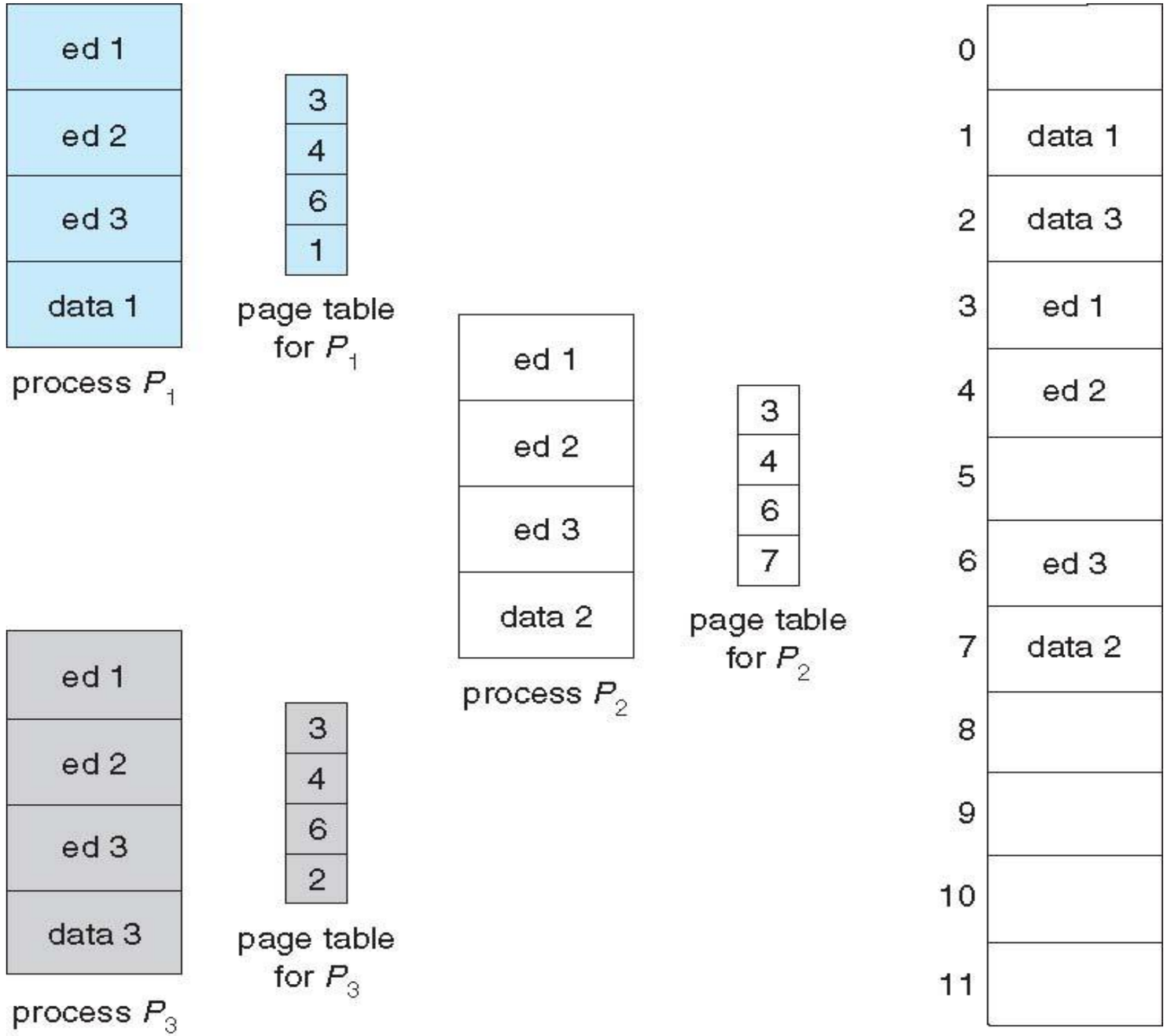
Some Exercises on TLB

- Consider a paging system with the page table stored in memory.
  - If a memory reference takes 50 nsec., how long does a paged memory reference take?
  - If we add TLB's, and 75% of all page table references are found in the TLBs, what is the effective memory reference time? Assume that finding a page table entry in the TLBs does not add any performance overhead.
- TLB access time is 1 nsec, memory access time is 5 nsec. What hit ratio is needed for an EMAT of 7 nsec?

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- We discussed this in chapter # 3.
- It is possible for several processes to share the same physical pages.
- Particularly useful in a time shared system.
- Consider such a time-shared system that supports 40 users, each of whom executes a text editor.
  - If the text editor has 150 KB of code and 50 KB of data space, what is the total memory requirement for all users?
    - 40 x (150 + 50) = 8,000 KB.
- If the text editor code could be shared among the 40 users, what is the total memory requirement?
  - (40 x 50) + 150 = 2150 KB.
- How do we do this?
  - By using the concept of shared pages.
- This is explained in the next slide.

Shared Pages Example



Shared Pages Example.

- The figure shows a 3-page editor (pages are: ed1, ed2 and ed3).
- Each page is 50 KB in size (for a total of 150 KB).
- The editor pages are shared among three processes: P_1, P_2 and P_3.
- Only 1 copy of the editor code needs to be kept in the physical memory.
- Each user's page table maps onto the same physical copy of the editor.
- Note that the data pages are mapped onto different frames.
- To support 40 users, what do we need?
  - We need 1 copy of the editor = 150 KB.
  - We also need 40 copies of the data page = 40 x 50 = 2000 KB (1 copy per user).
  - Total space required = 2000 + 150 = 2150 KB.

Hierarchical Paging

- Assume that *m* = 32, this means that we have a logical address space of size 2^{32}.
- If the page size is 4 KB, then *n* = 12, as 4 KB = 2^{12}.
- So, p = 20 and d = 12.
- Suppose that each page table entry = 4 bytes (= 32 bits).
- What is the size of the page table?
  - = number of pages * size of a page.
  - = $2^{32}/2^{12}$ * 4.
  - = 2^{20} * 4.
  - = 4194304 bytes.
  - = 4 MB.
- **Each** process may need up to 4 MB of physical address space for the page table **alone**.
- Clearly, we may not be able to allocate the page table contiguously in memory.

Hierarchical Paging

- It is not feasible to allocate the page table contiguously in memory.
- What's the way out?
- Divide the page table into pieces and then allocate the pieces in a non-contiguous manner.
  - Note that we adopted a similar approach with logical memory earlier.
- We can "page" the page table.
  - This is called two-level paging.
- Our logical address now has 3 sections:
  - Primary page #
  - Secondary page #
  - Offset.

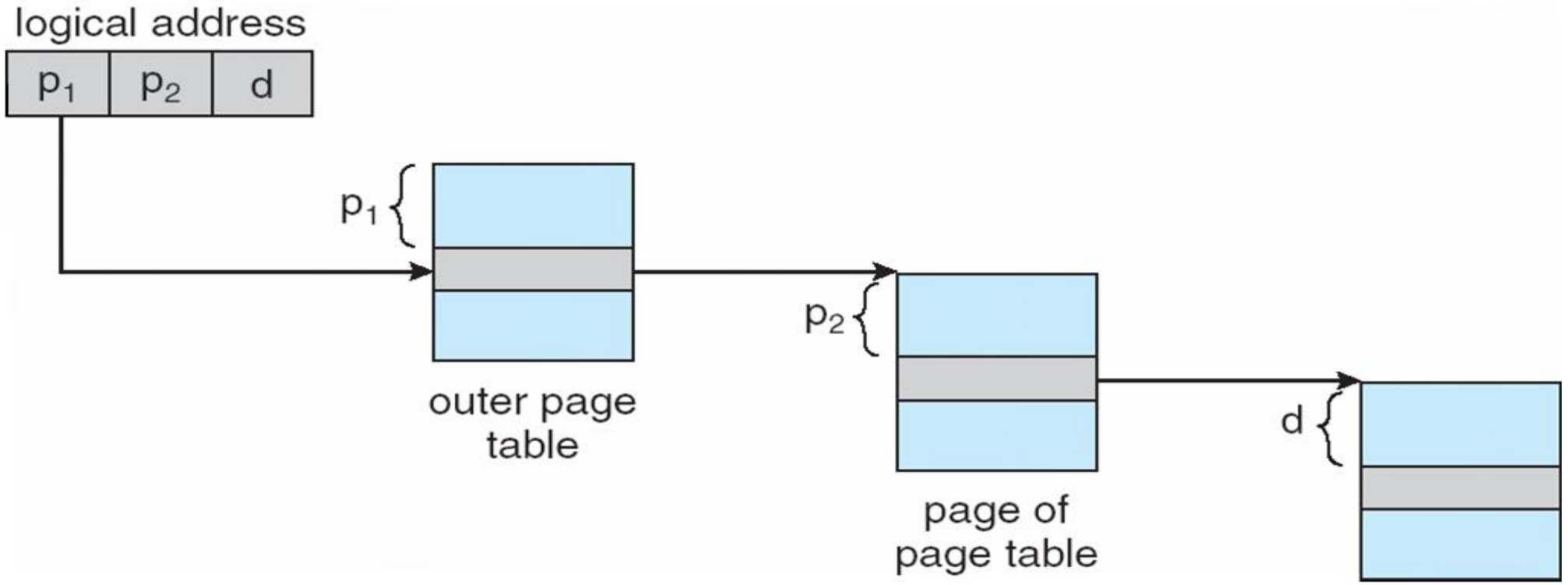
Two Level Page Table

- We have one primary page table and multiple secondary page tables.
  - The secondary page tables have been carved out of the primary page table.
- Step 1: first, we take the primary page # and use as index and look for the corresponding entry in the primary page table.
  - This gives us the secondary page table #.
- Step 2: next, we go to that secondary page table and within that table, we look up the physical frame number (using secondary page #).
- Step 3: the frame number is combined with the offset to get the physical address.
- Lets draw a diagram to illustrate this scheme.
- We call this “hierarchical paging”

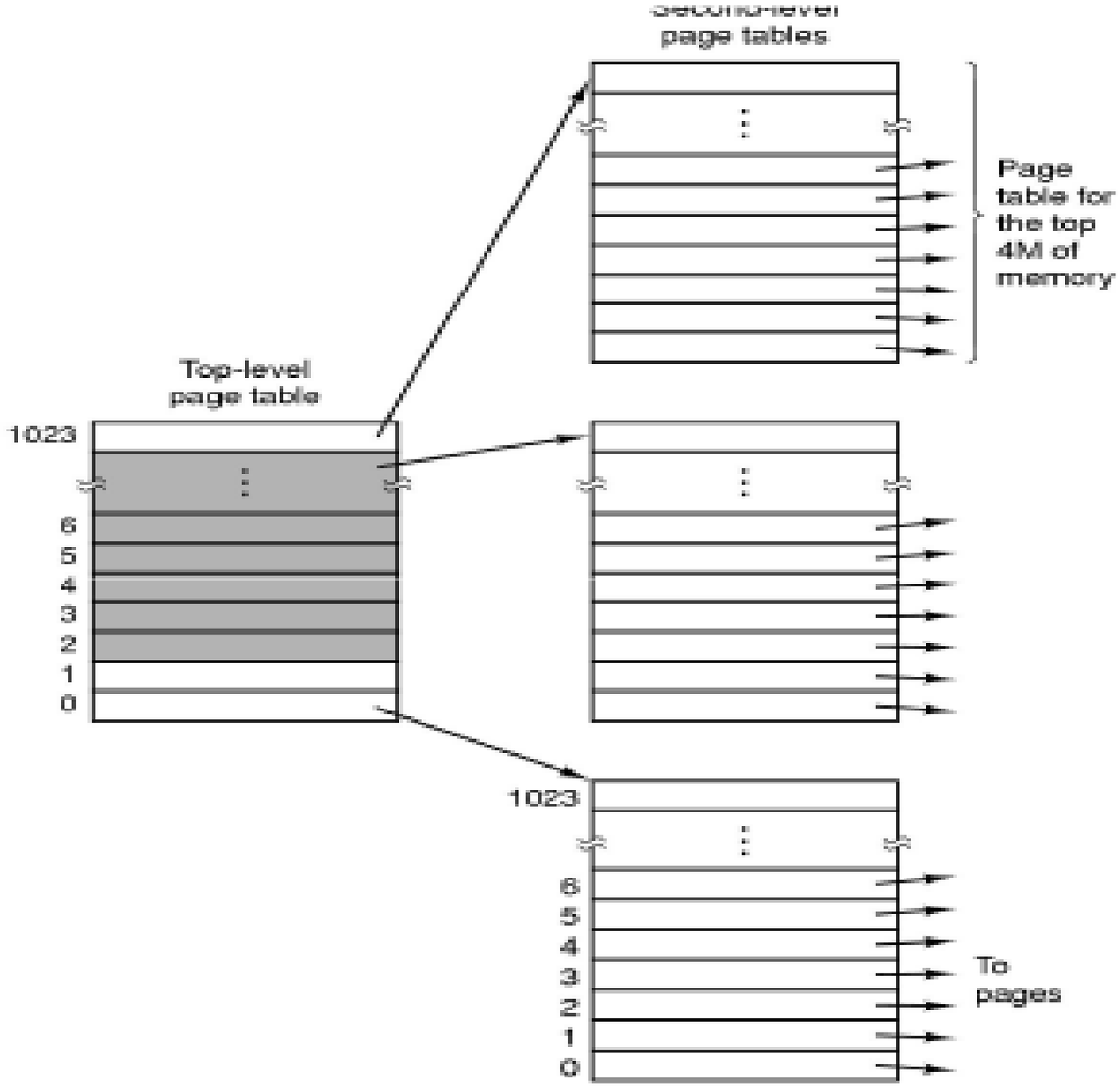
Logical Address in Two Level Paging

- Lets consider the previous example with a logical address space of 2^{32}, page size of 4 KB, page table of 4 MB.
- The logical address will now have three components.
- The total number of bits in the address = 32.
  - Primary page number (p_1) = 10 bits.
  - Secondary page number (p_2) = 10 bits.
  - Offset (d) = 12 bits.

Address-Translation Scheme



Two Level Page table



Example of VAX.

- VAX (Virtual Address Extension) architecture supports a variation of two level paging.
- Popular machine, sold from 1977 through 2000.
- It is a 32 bit machine with a page size of 512 bytes.
- Logical address space = 2^{32}.
- This address space is divided into 4 sections, of 2^{30} ($2^{32}/4$) bytes each.
- Each sections represents a different part of the logical address space.
- Logical address is as follows:
  - First 2 high order bits designate the appropriate section.
  - The next 21 bits represent the logical page number of that section.
  - The final 9 bits are for the offset in the desired page.
- Not all the address space of a process may be active at a point of time.
- Only the section (s) required is/are loaded into memory.
- No entries in page table for address spaces that are not required.
- More on this in the next chapter.

An Exercise on Two-Level Paging

- A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top level page table field, an 11 bit second level page table field, and an offset. How large are the pages and how many of them are there in the address space?
- **Lets see the process of translation of an actual address with 2-level paging. Assume that the virtual address is: 4,206,596.**

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 have 2^{10} entries
  - Address would look like:

outer page	inner page	offset
p_1	p_2	d
42	10	12

  - What is the issue here?
  - Outer page table has 2^{42} entries or 2^{44} bytes!
  - One solution is to add a 2^{nd} outer page table.
  - In other words, page the outer page table.

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

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

Inverted Page Table

- We have seen: page table has one slot per virtual page.
- Assume that *m* = 64, so logical address space = 2^{64}.
- If page size = 4 KB = 2^{12}, then # of pages = $2^{64}/2^{12} = 2^{52}$!
- Now, let's say we only have 512 MB of RAM available.
  - # of pages possible?
  - = 512 MB/ 4 KB = 128 K.
- We observe that the bottleneck is with the amount of available physical memory. Obviously, it can't hold all the pages at once.
- Key idea: how about storing a single page table entry per physical page (and not per virtual page)?
  - The page table size in this case would reduce considerably.
- Such a page table is called an inverted page table.
- Used in Sun UltraSPARC, Apple Power PCs, IBM PCs.

Inverted Page Table

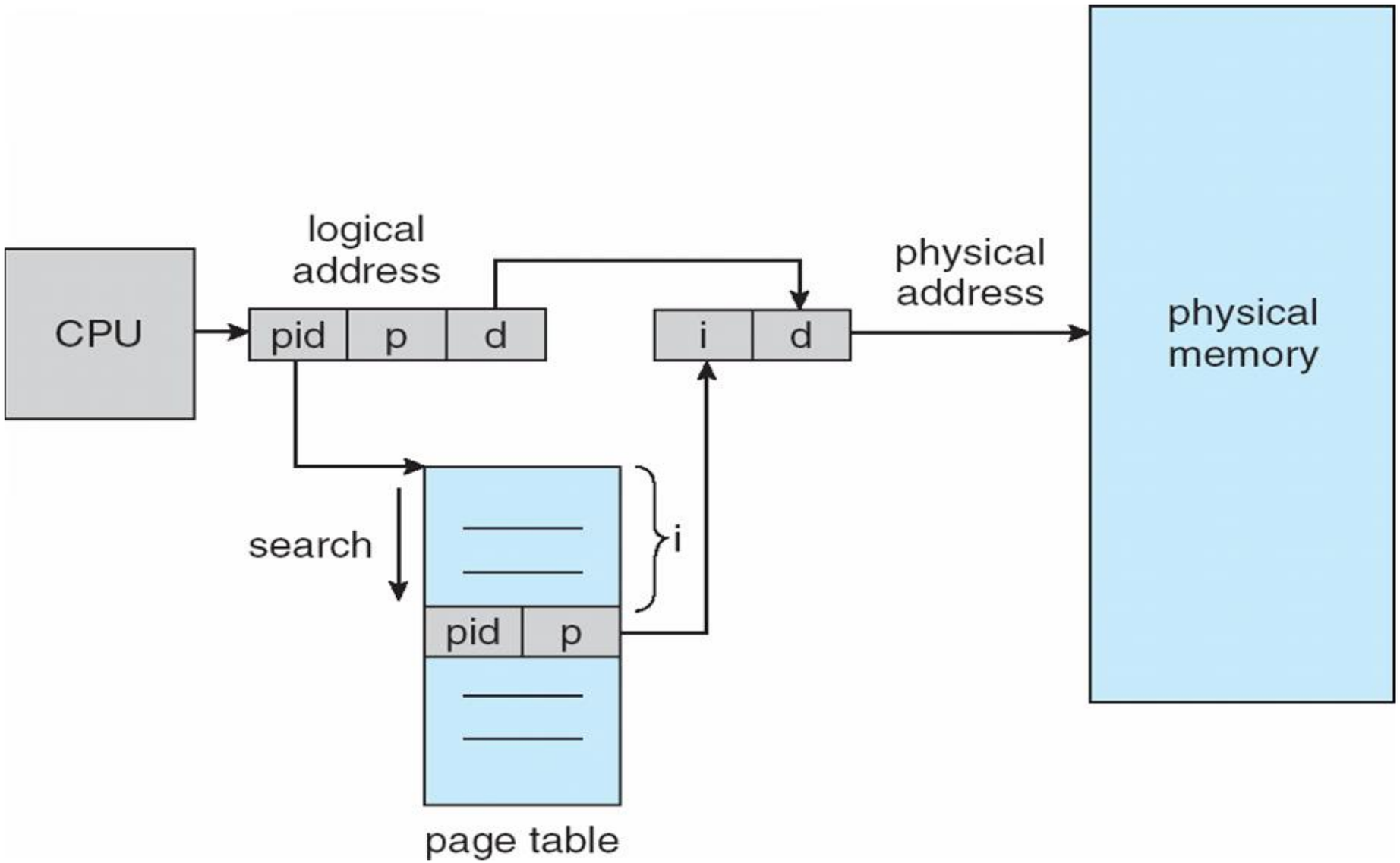
- We know that multiple processes use the physical memory at a time.
- Hence, we can have one global page table (1 entry per physical page).
  - Instead of one page table per process.
- How do we implement this?
- We can have the page table as a linear array that contains one entry per physical page.
  - Since the page table is shared among processes, each entry needs to contain the process ID (pid) of the page owner as well.
  - Each entry contains the virtual page number instead of the physical page number.
  - The physical page number is an index in the table.
- Let's illustrate this using a diagram for IBM systems.

Inverted Page Table

- In order to translate a logical address, the following steps are taken:
  1. The logical page number and current PID are compared against each entry in the page table.
  2. When a match is found, the index of the match replaces the physical page number in the address field and the physical address is obtained.

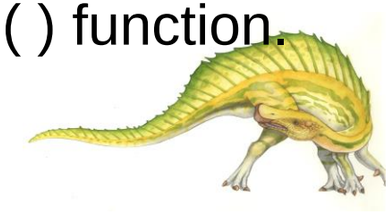
Exercise: 64-bit addresses, 4 KB page size, 1 GB of main memory. Size of - (i) conventional page table, (ii) inverted page table?

Inverted Page Table Architecture

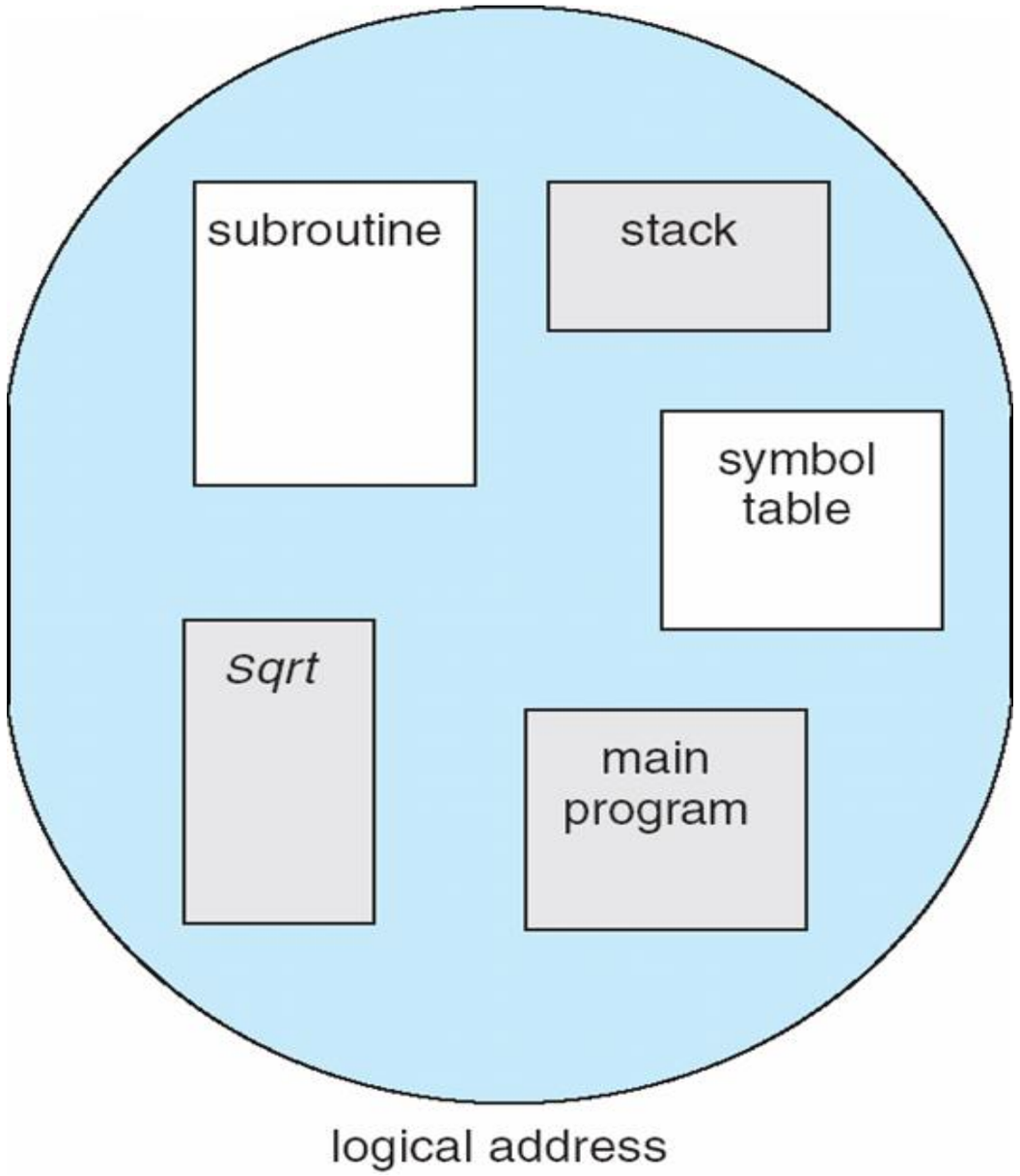


Segmentation

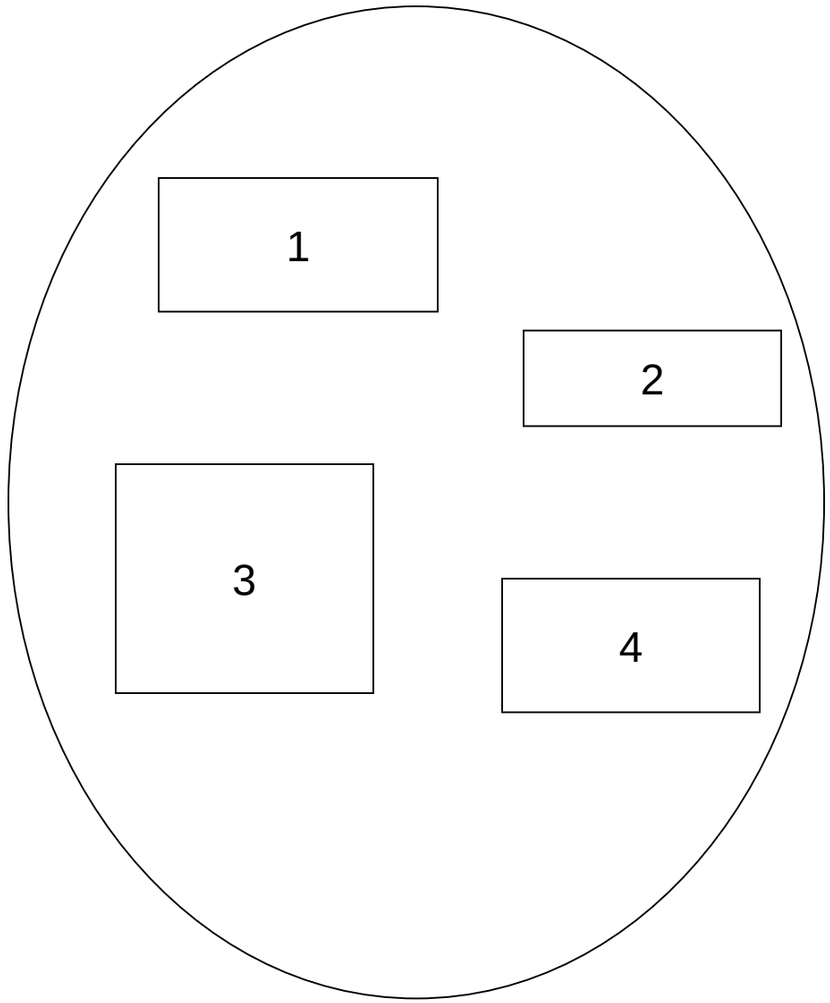
- The paging technique thinks of memory as a large linear array of bytes.
  - Some bytes contain instructions, others contain data.
- This concept of memory may be abstract to users/programmers.
- Programmers, in contrast, would think of memory as a collection of variable sized segments/parts, with no necessary ordering among them.
- Consider a program. To a programmer, it contains, for example:
  - Main function with a set of other functions.
  - Certain data structures like a stack, queue etc.
  - Variables.
  - Libraries.
- All these can be segments stored at certain memory locations.
  - Each segment has its own size.
- Elements within a segment are identified by their offset from beginning of segment.
  - 1st statement of program, 7th stack frame entry, 5th instruction in sqrt ( ) function.



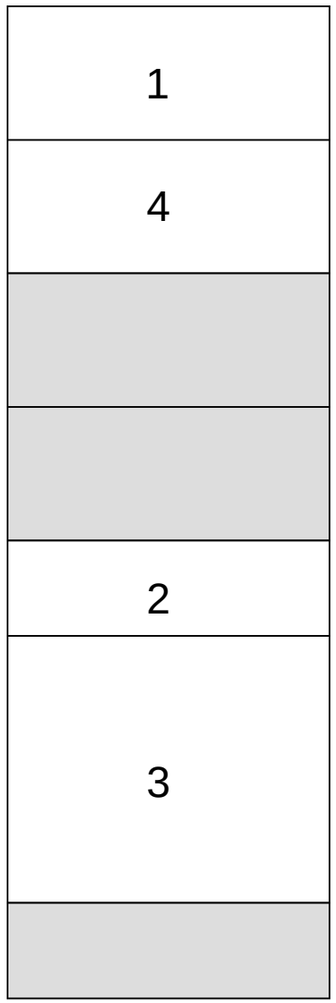
Programmer's View of a Program



Logical View of Segmentation



user space

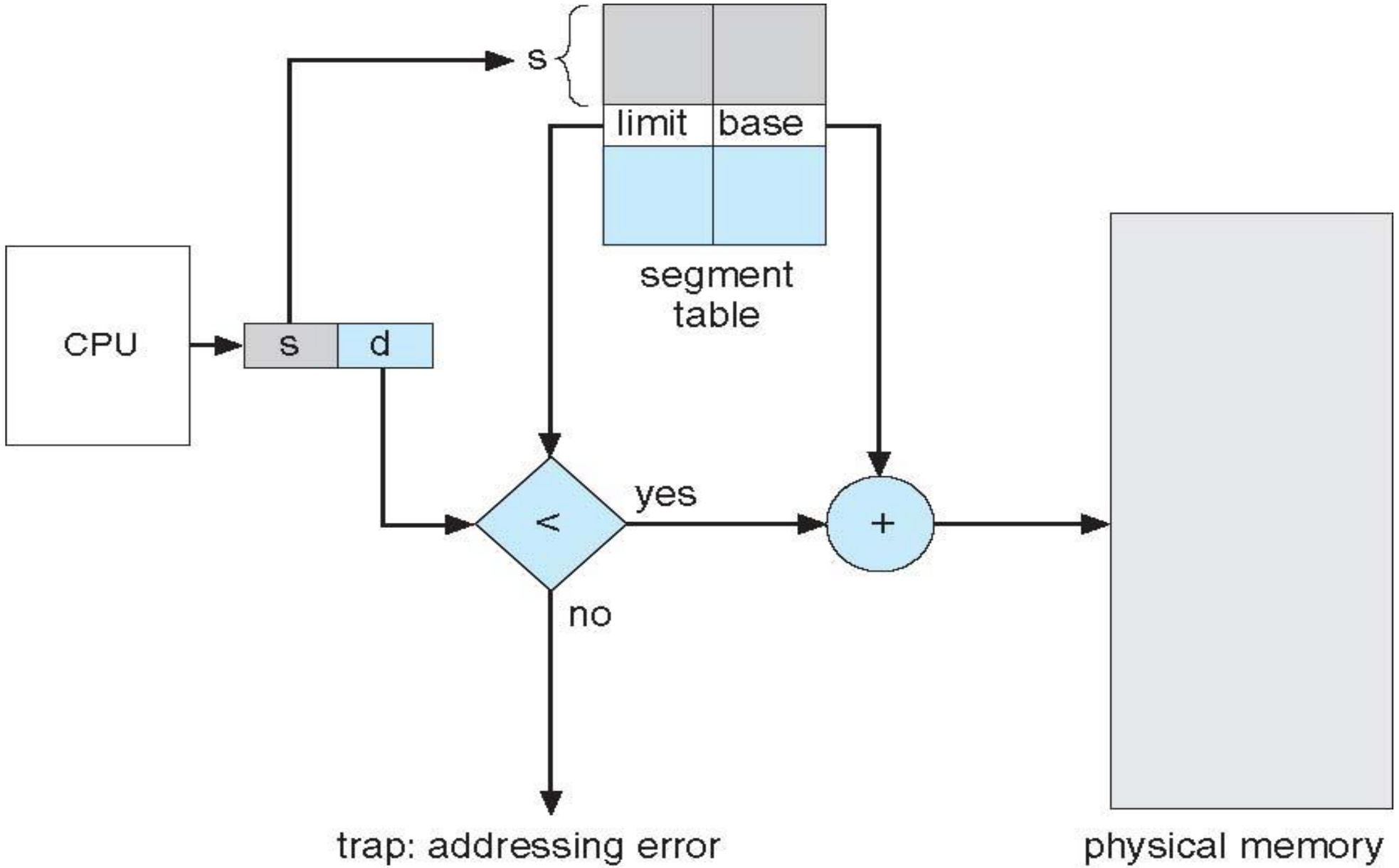


physical memory space

Segmentation

- Segmentation is a memory management scheme that supports this programmer's view of memory.
- **A logical address space is a collection of segments.**
  - ***Each segment has a name/number and a length.***
- A logical address consists of a *two tuple*: <segment-number, offset>.
- Clearly, we need a mapping between the logical addresses and physical addresses.
- This mapping is done by a segment table.
- Segment table is an array of **{base, limit}** pairs.
- Each entry in the segment table has:
  - Segment base: starting physical address where segment lives.
  - Segment limit: the length of the segment.
- Compiler automatically constructs the segments upon compiling.
- It also assigns the segment numbers.

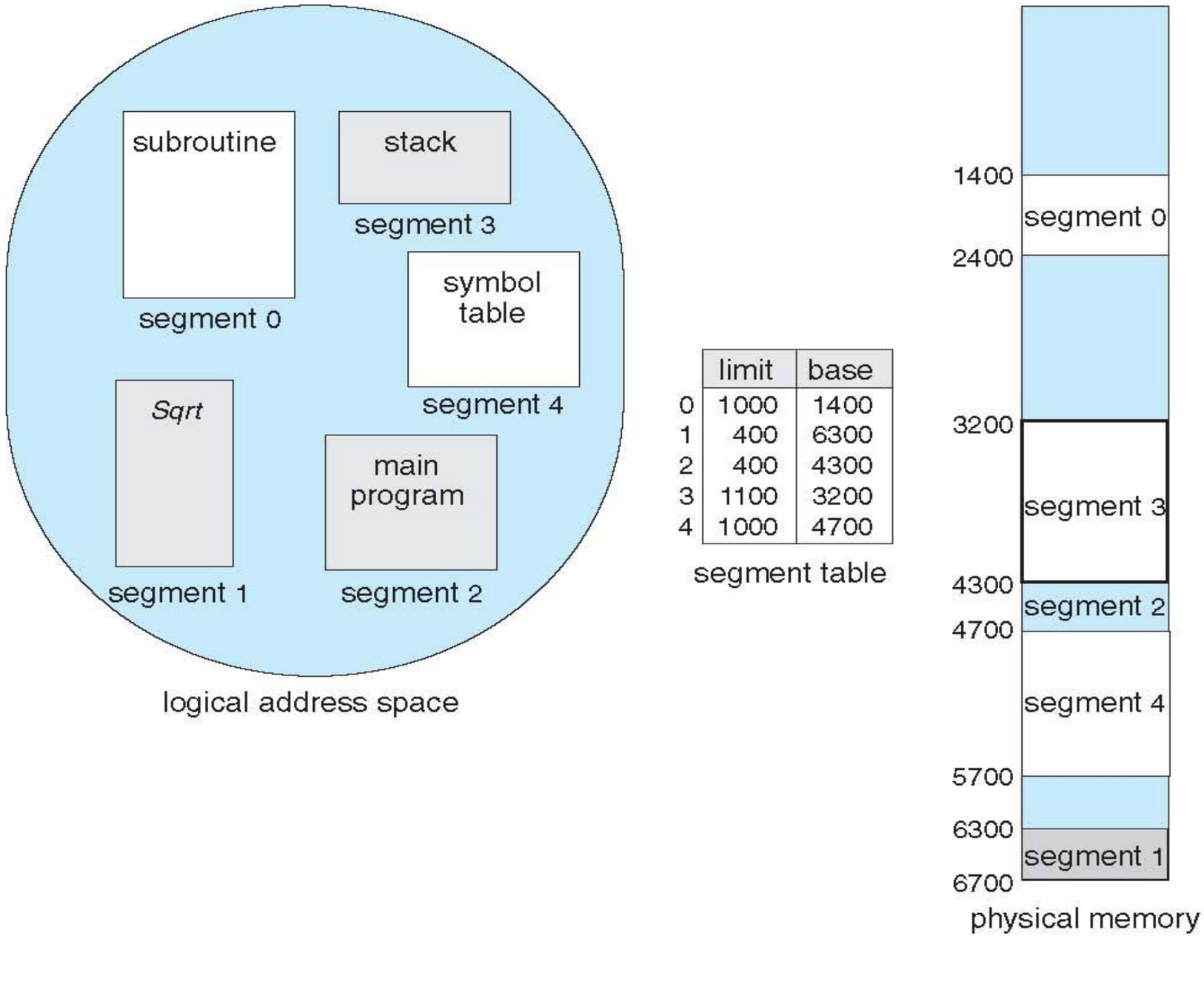
Segmentation Hardware



Mapping in Segmentation

- A logical address contains two parts: **a segment number (s)** and an **offset (d) into that segment**.
- The segment number is used as an index into the segment table to look up the corresponding segment start address and the segment limit.
- Obviously, the offset must be less than the limit, else it is an addressing error and traps to the operating system.
- Next slide shows an example.
  - We have five segments – 0 to 4.
  - Segments are stored in physical memory as shown.
  - Reference to byte 53 of segment 2 is mapped to:
  - 4353.
  - Reference to byte 852 of segment 3 is mapped to:
  - 4052.
  - Reference to byte 1222 of segment 0 is mapped to:
  - trap to the OS (addressing error).

Example of Segmentation



Segmentation

- A segment is allocated contiguous space in the physical memory.
- Good idea if segments can grow and shrink (data segment, stack segment).
  - They are assigned a specific space and can grow and shrink within that space.
- Disadvantage is that we can have external fragmentation.
  - We have holes in memory that may not be filled by segments.
  - Would need to use first fit, best fit algorithms for allocation.
- Both paging & segmentation have their pros & cons.
- Paging: << fragmentation.
- Issue with dynamically growing process parts, hard to visualize for programmers.
- Segmentation: intuitive model for programmers, better with dynamically growing process parts.
  - Fragmentation problem, need to keep whole segment in memory.

MULTICS Memory Management

- What if we can combine paging & segmentation?
- Get the best of both worlds.
- This is the approach taken by most modern operating systems.
- Basic idea as follows:
  - The logical memory is composed of segments.
  - Each segment is composed of pages.
- For large segments, not desirable to keep them whole in main memory.
- Idea is to page segments, so keep only pages that are needed.
- One example that takes this approach: MULTICS OS.
- Invented at MIT in 1969.
- Inspired concepts in UNIX, TLBs, cloud etc.
- Virtual memory composed of segments: 2^{18} of them.
- Each segment was up to 64K.
- Each segment was paged.

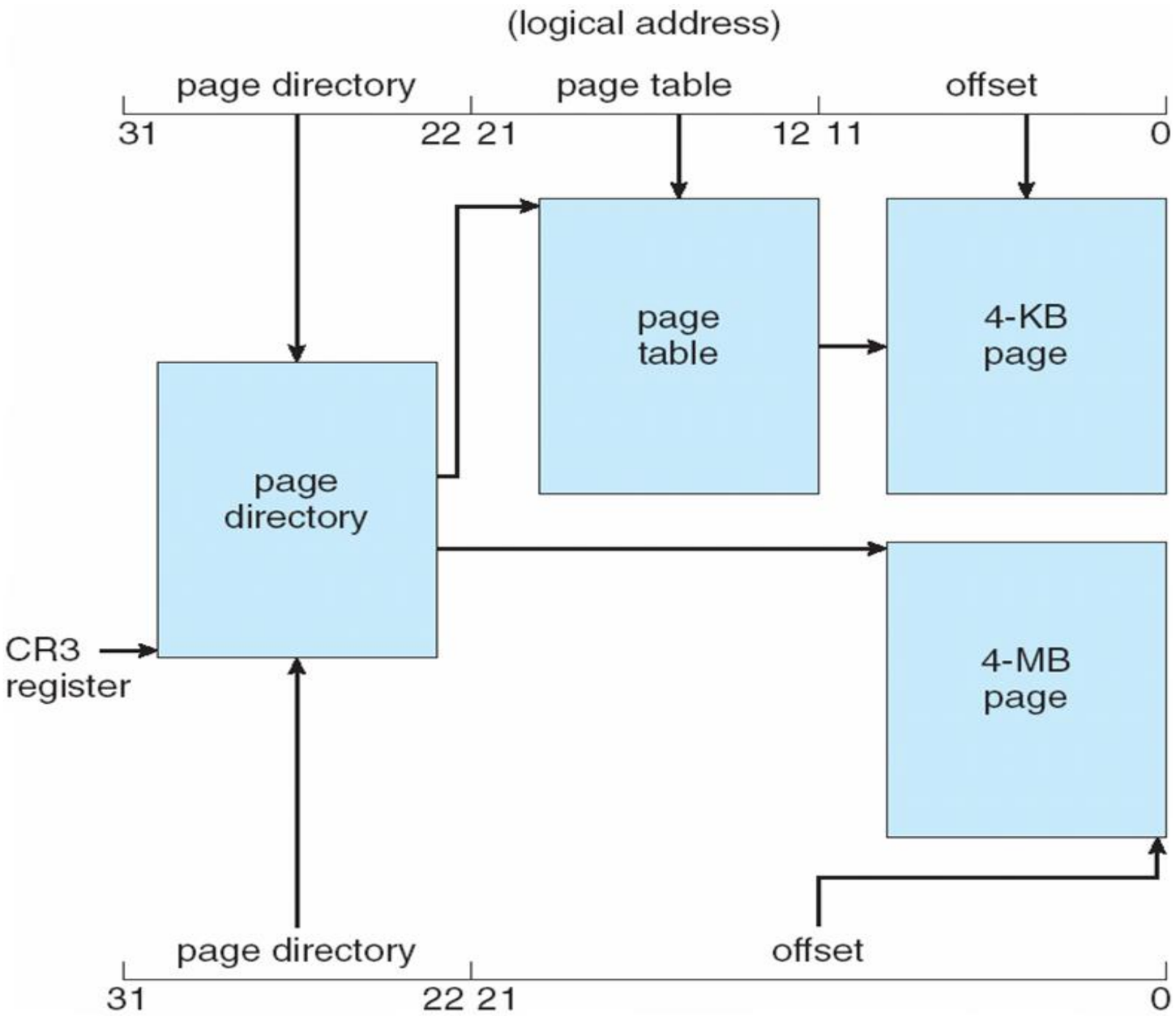
MULTICS Memory Management

- Each MULTICS program had a segment table, with one descriptor per segment.
- About ¼ million (2^{18}) segments, segment table was also paged.
- Each descriptor contained a pointer (address) to page table of that segment.
  - Also contained segment size, other bits.
- MULTICS virtual address had three components:
  - 18-bit segment number.
  - 6-bit page number.
  - 10-bit offset.
- Address lookup as follows:
  - Get segment number & look in segment descriptor.
  - Locate page table for segment & get PTE.
  - Get physical frame address & add offset to get physical address.

Intel Pentium Paging

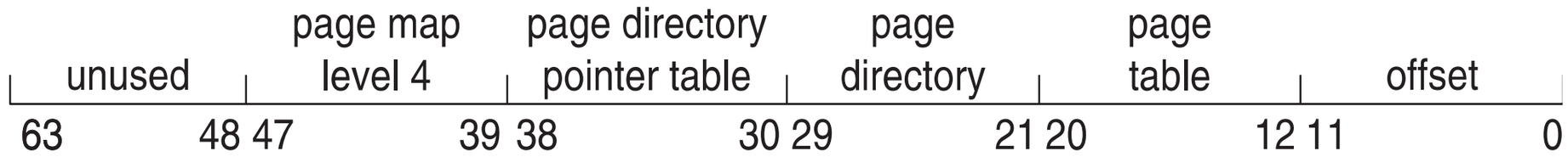
- Used page sizes of 4 KB or 4 MB.
- Two level paging for 4 KB pages (we have discussed this).
- The 10 highest order bits of an address refer to an entry in the outer page table called page directory.
  - The CR3 (control register for virtual memory) points to the page directory for current process.
- Page directory points to inner page table that is indexed by next 10 bits in the address.
- Finally, last 12 bits provide the offset in the page.
- Have a page_size flag which indicates page size.
- For 4 MB page, inner page table is bypassed.
  - May be used for fixed size, large kernel data structures.
- Address is (10, 22) for (page directory, page table).
- Logic for using multiple page sizes: to adjust the TLB reach.
- TLB reach = # of TLB entries * page size.

Pentium Paging Architecture



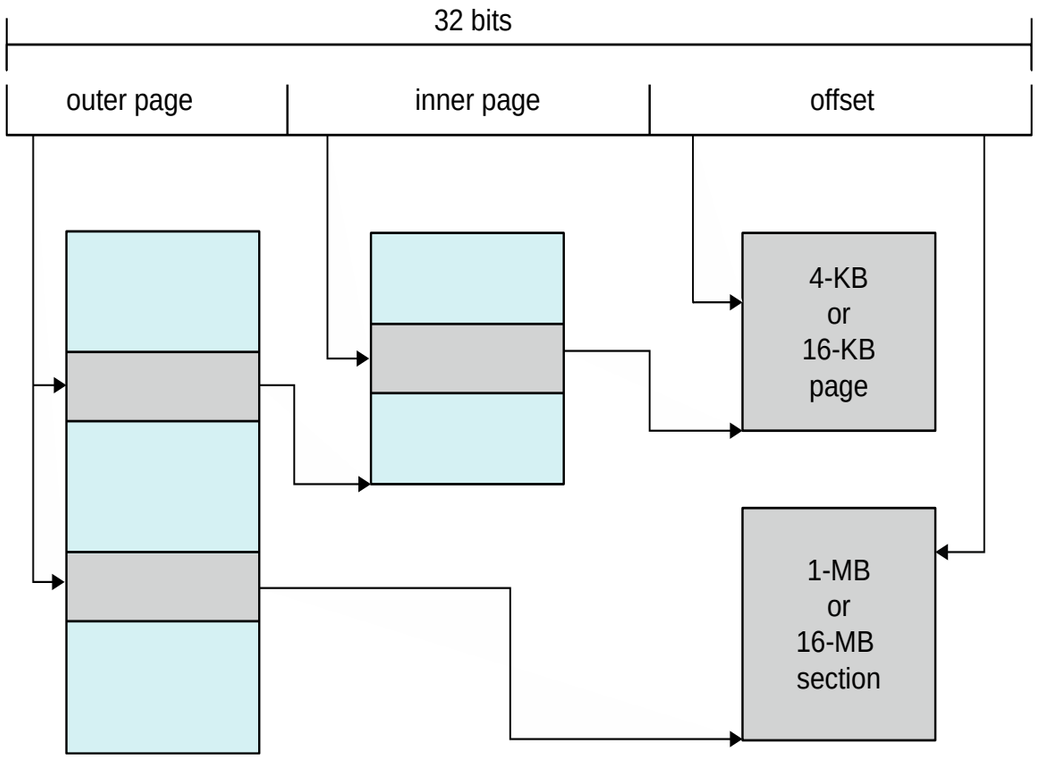
Intel x86-64

- Current generation Intel x86 architecture.
- 64 bits is ginormous (> 16 exabytes).
- In practice only implement 48 bit addressing.
  - Page sizes of 4 KB, 2 MB, 1 GB.
  - Four levels of paging hierarchy.



ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android).
- Modern, energy efficient, 32-bit CPU.
- 4 KB and 16 KB pages.
- 1 MB and 16 MB pages (termed **sections**).
- One-level paging for sections, two-level for smaller pages.
- Two levels of TLBs:
  - Outer level has two micro TLBs (one data, one instruction).
  - Inner is single main TLB.
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU.



End of Chapter 8