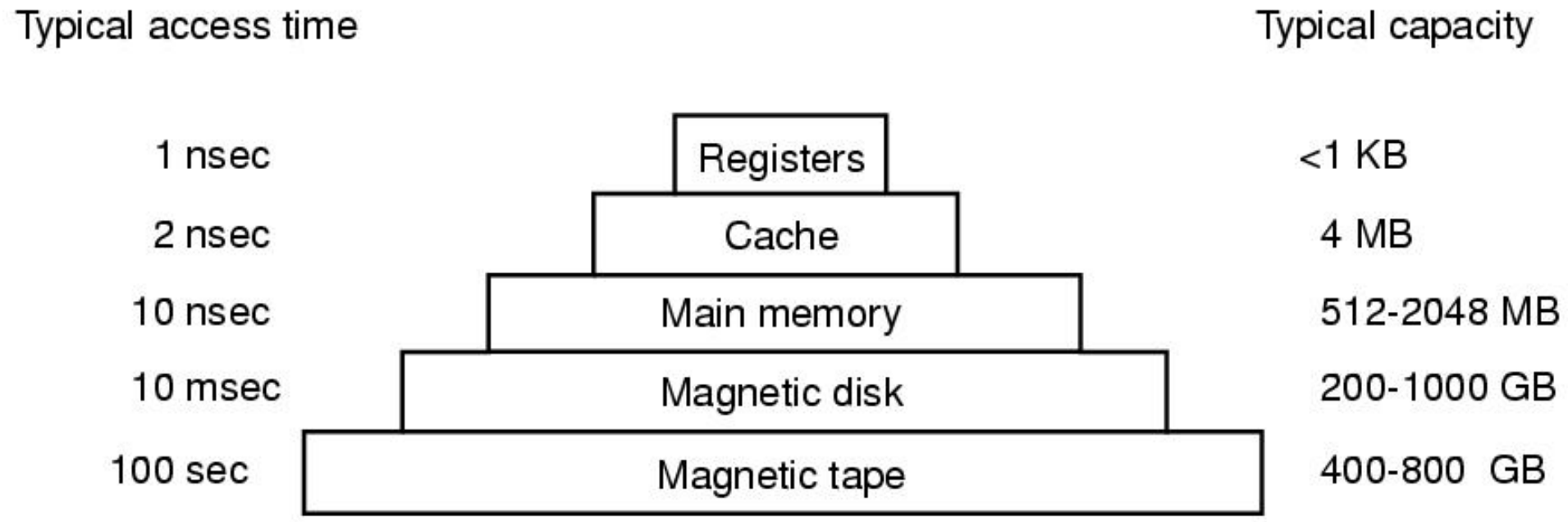
Chapter 3
Memory Management

Storage Hierarchy

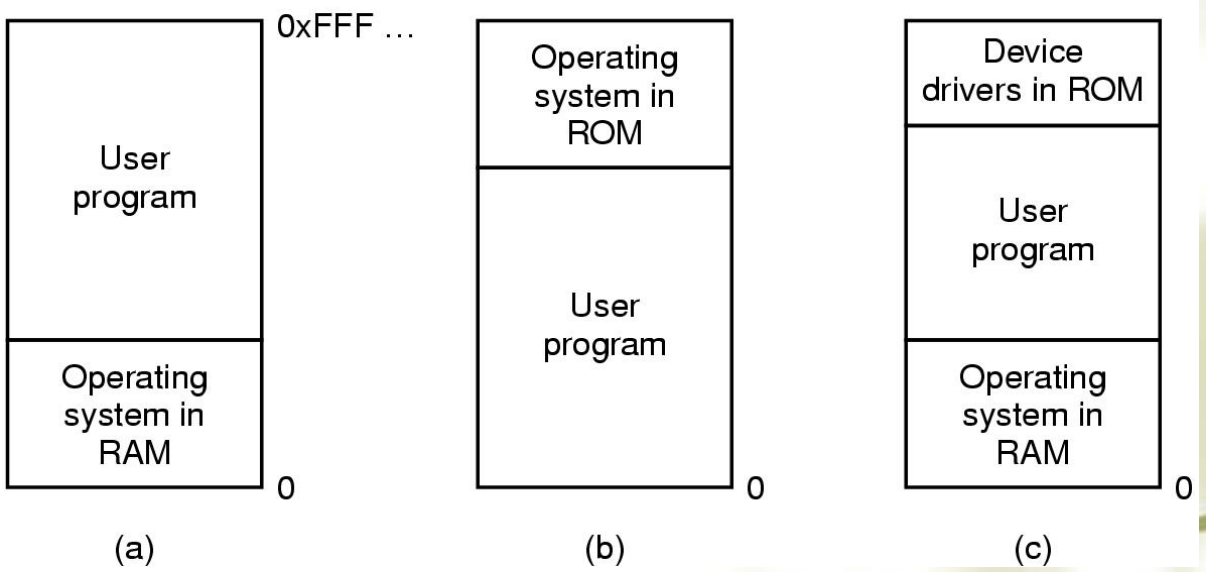
- Ideally programmers want memory that is
  - Large
  - Fast
  - Nonvolatile
  - Inexpensive
- Memory hierarchy
  - A few MBs of fast, expensive, volatile cache memory
  - A few GBs of medium-speed, medium-price, volatile main memory
  - A few TBs of slow, cheap, nonvolatile disk storage
- Memory manager abstracts the memory hierarchy into a useful model and then manages the abstraction.

Storage Hierarchy



No Memory Abstraction

- The simplest memory abstraction is no memory abstraction. Every program simply saw the physical memory.
- It is not possible to run two programs in memory at the same time.
- Three different ways to organize memory with OS and one user program.



Multiple Programs in Memory

- To allow multiple applications to be in memory at the same time without interfering with each other, two problems must be solved:
  - Protection
    - How does the system prevent processes interfering with each other?
  - Relocation
    - Process may not be placed back in same main memory region
    - How does a task or process run in different locations in main memory?

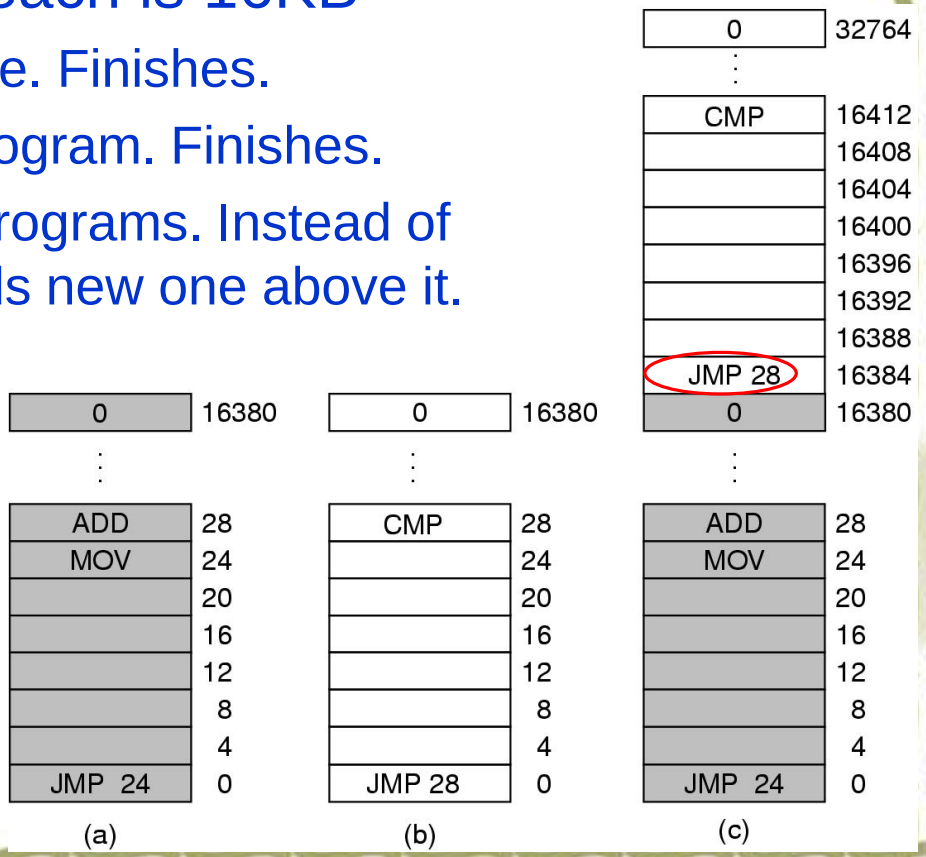
Multiple Programs without Memory Abstraction

- With the addition of some special hardware, it is possible to run multiple programs concurrently.
- Protection Solution of Early IBM 360
  - Divide memory into 2-KB blocks
  - Assign each block a 4-bit protection key, which saved in special registers of CPU
  - Each process also has a protection key value associated with it. (kept in PSW)
  - When a process access memory, compare the protection code of the memory block with the PSW key.

Multiple Programs without Memory Abstraction

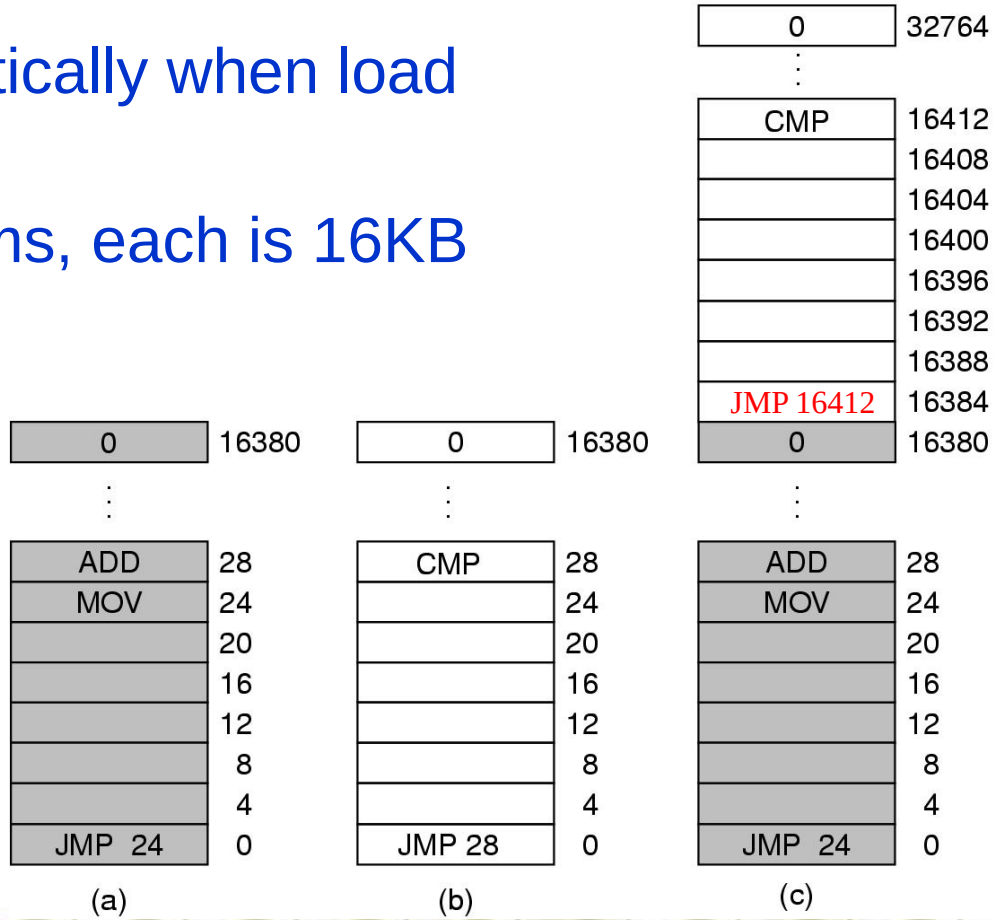
- Relocation Problem of Early IBM 360
  - Example: two programs, each is 16KB
    - (a) 16 KB program runs alone. Finishes.
    - (b) OS loads a new 16KB program. Finishes.
    - (c) OS tries to run multiple programs. Instead of evicting old program, adds new one above it.

      JMP 28 causes an error!
  - Problem
    - Two programs both reference physical addresses.



Multiple Programs without Memory Abstraction

- Static Relocation
  - Modify addresses statically when load process.
  - Example: two programs, each is 16KB
    - ञ़त़ऱझढ़ट़त़ठ़ष़झढ़ठ़झञ़



Multiple Programs without Memory Abstraction

- Static Relocation
  - Advantage
    - Requires no hardware support.
  - Disadvantages
    - Slows down loading.
    - Once loaded, the code or data of the program can't be moved into the memory without further relocation.
    - The loader needs some way to tell what is an address and what is a constant.
      - E.g. MOVE REGISTER1, 28

Address Spaces

- Recall: Memory needs two things for multiprogramming.
  - Protection
    - How does the system prevent processes interfering with each other?
  - Relocation
    - How does a task or process run in different locations in main memory?
- A better solution: **Address Space**

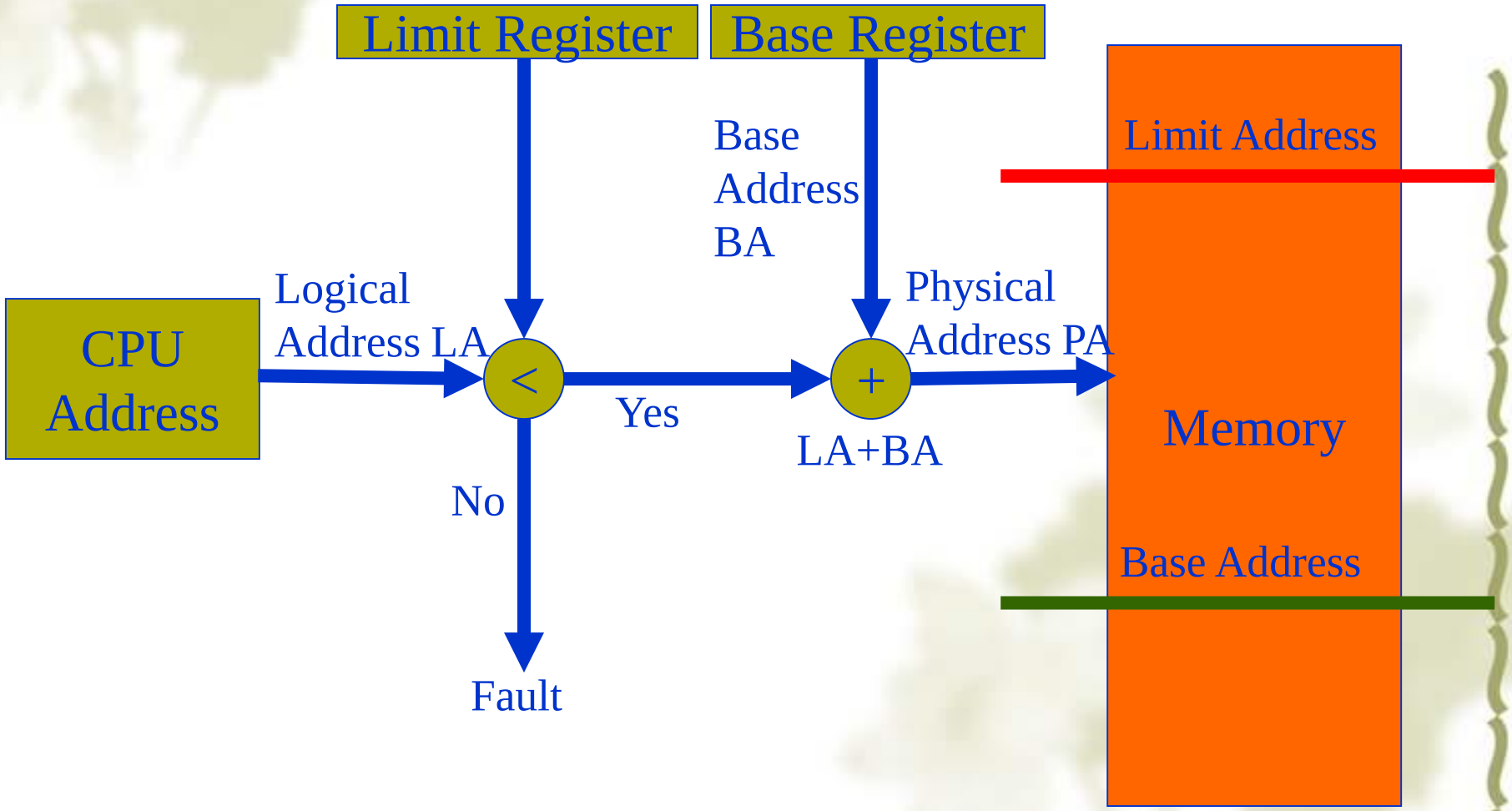
Address Spaces

- Address Space
  - Address space is a set of addresses that a process can use to address memory.
- Each process has its own address space, independent of those belonging to other processes.
  - Logical address space, range (0 to max)
    - Logical addresses, Virtual addresses
- Address Binding
  - The process of associating program instructions and data (addresses) to physical memory addresses is called address binding, or relocation.

Address Spaces

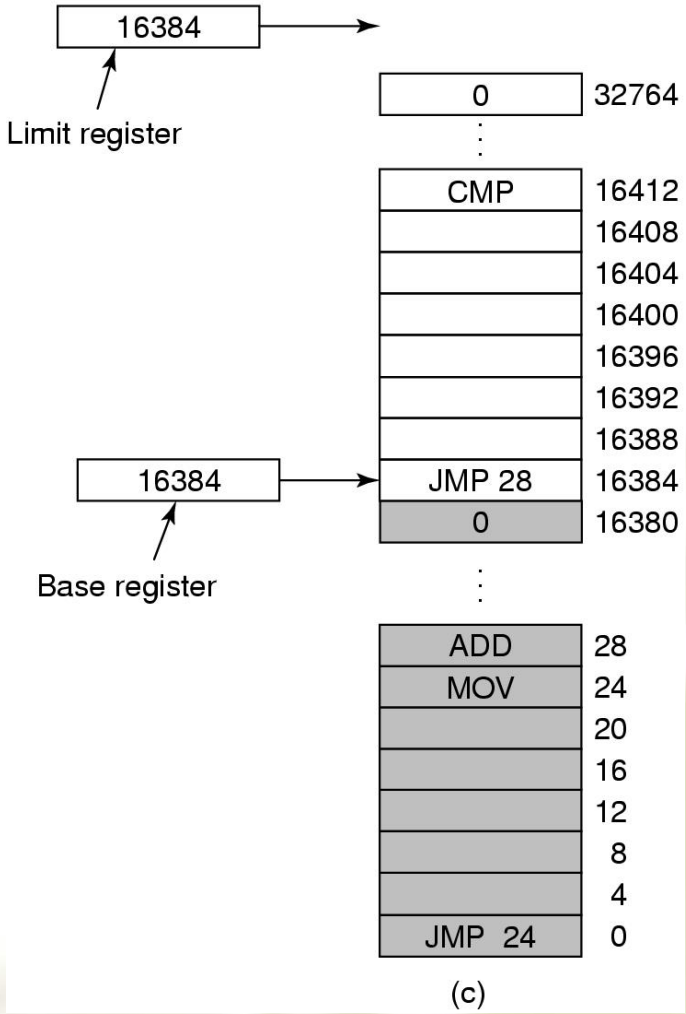
- Dynamic Relocation
  - Map each process' address space onto a different part of physical memory.
  - Physical address space, range (R+0 to R+max) for base value R
    - Physical addresses, Real addresses
  - Require Hardware Support: Equip CPU with two special hardware registers: base and limit
    - Base register: start location for address space
    - Limit register: size limit of address space

Base and Limit Registers



Base and Limit Registers

- Example: two programs, each is 16KB
  - First program
    - Base register: 0
    - Limit register: 16384
  - Second program
    - Base register: 16384
    - Limit register: 16384



Base and limit registers can be used to give each process a separate address space.

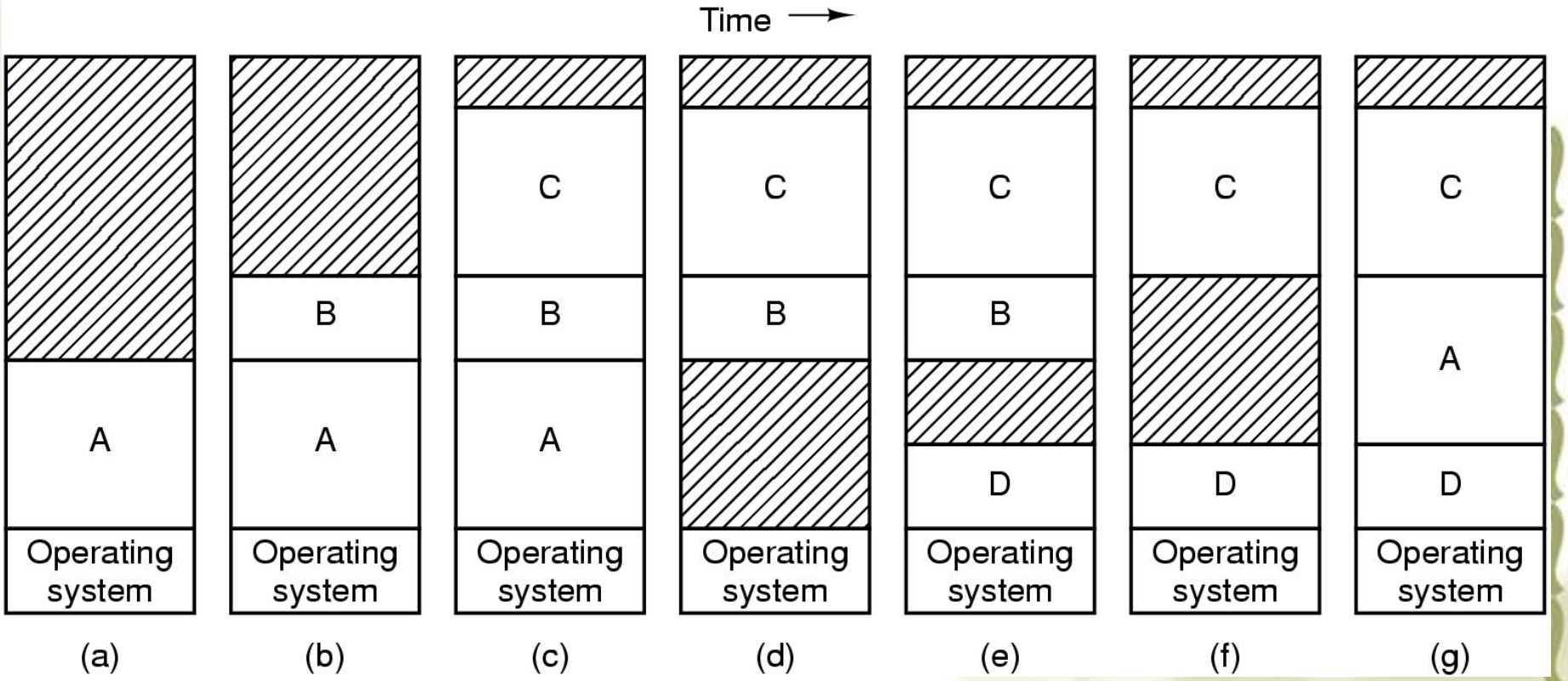
Address Spaces

- Dynamic Relocation
  - Advantages
    - OS can easily move process during execution.
    - OS can allow process to grow over time.
      - OS just changes limit register or moves it.
    - Simple, fast hardware
      - Two special registers, add & compare
  - Disadvantages
    - Slows everything (add on every reference).
    - Can’t share memory between processes.
    - Process limited to physical memory size.
    - Complicates memory management.

Swapping (1)

- Lots of programs, total size exceeds memory
  - **Swapping**: brings the whole process into memory, runs it for a while, and then move it back on the disk.
  - **Virtual memory**: allows programs to run even when they are only partially in main memory.
- Swapping allows several processes to share a partition

Swapping (2)

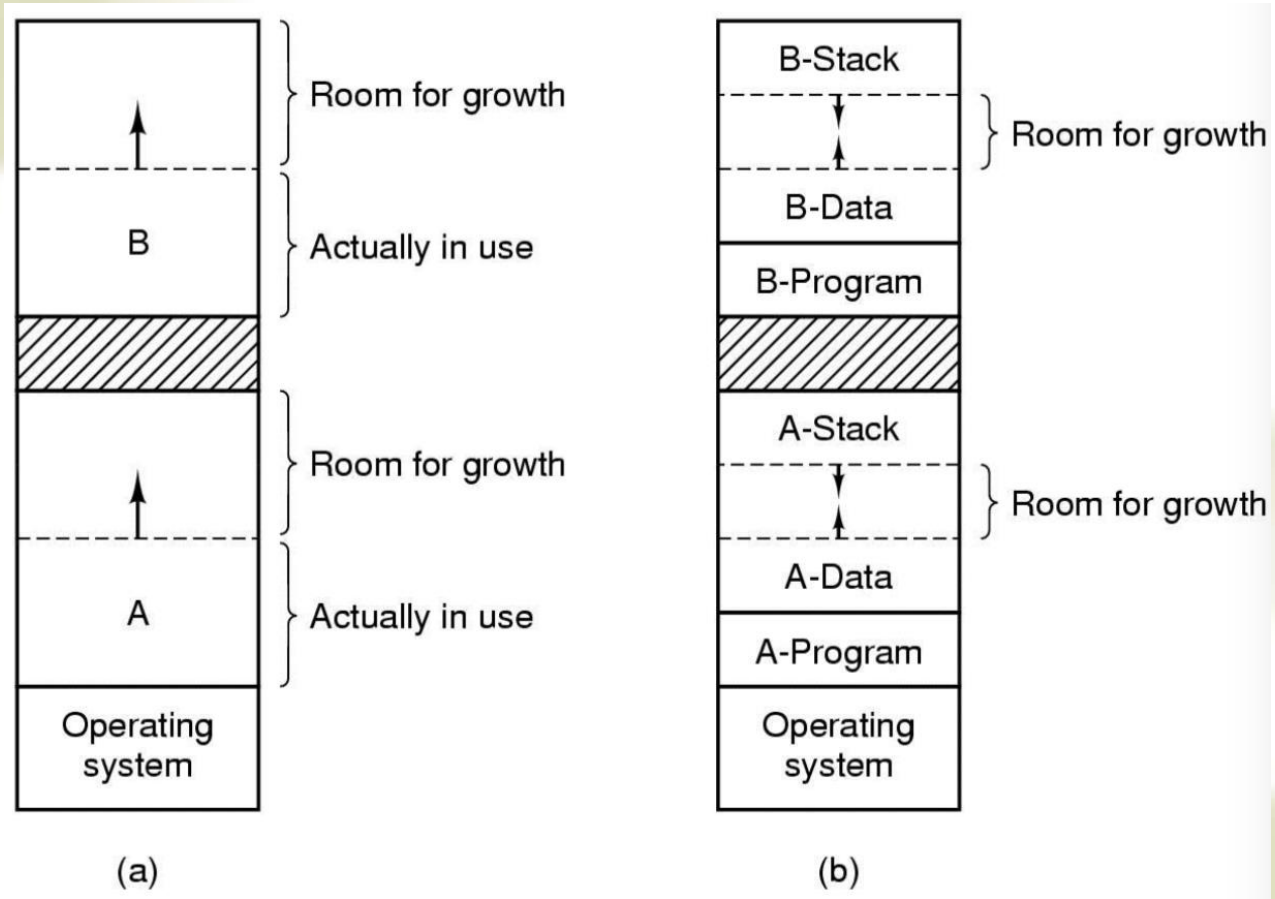


Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Swapping (3)

- Allocating memory dynamically
  - When a process come, allocate space as it demand if have enough space;
  - Or process wait on disk;
- What if most processes will grow as they run?
  - Processes that grow can be swapped out and swapped back in a bigger partition
  - But swapped in & out cost too much
- Solution for Growing Process

Swapping (4)

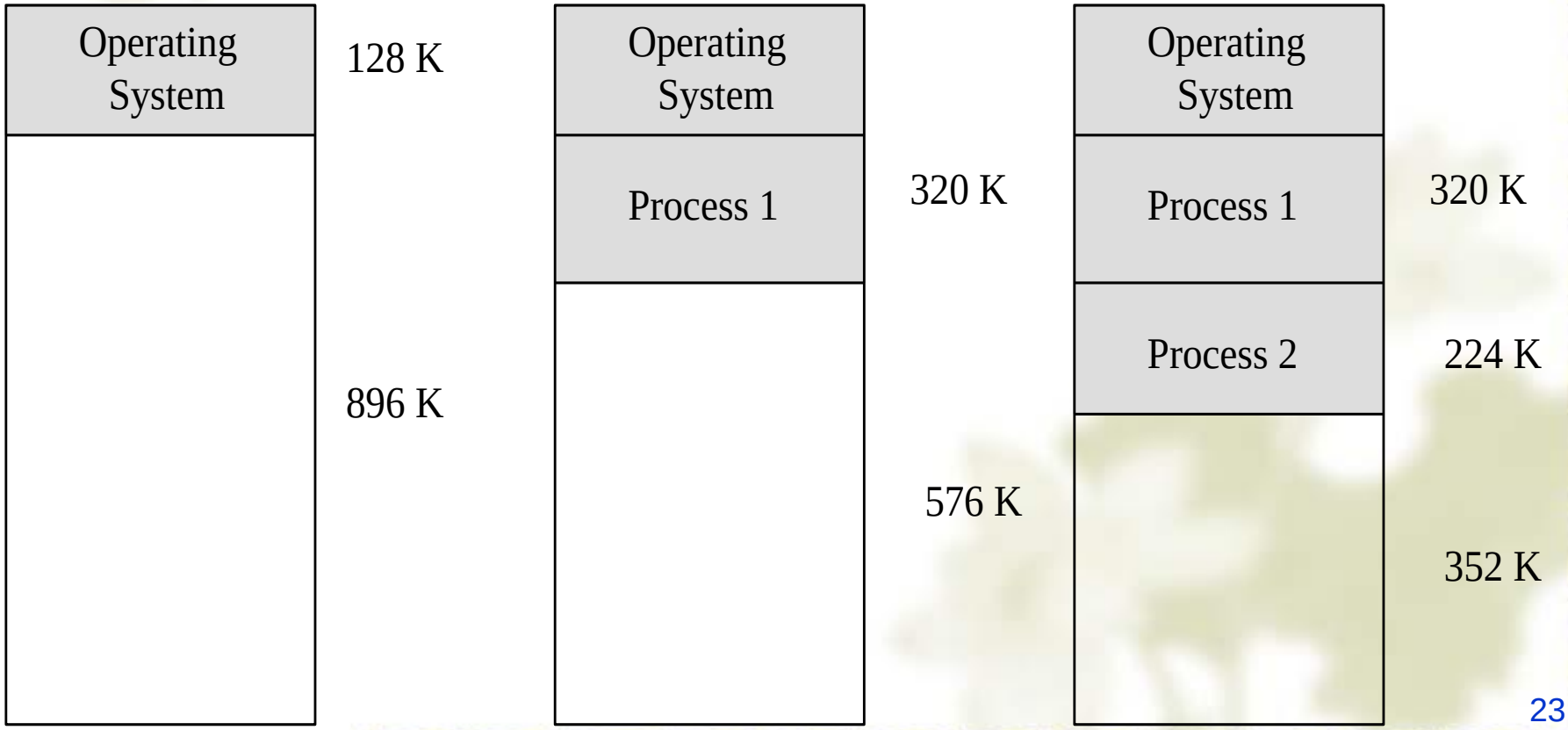


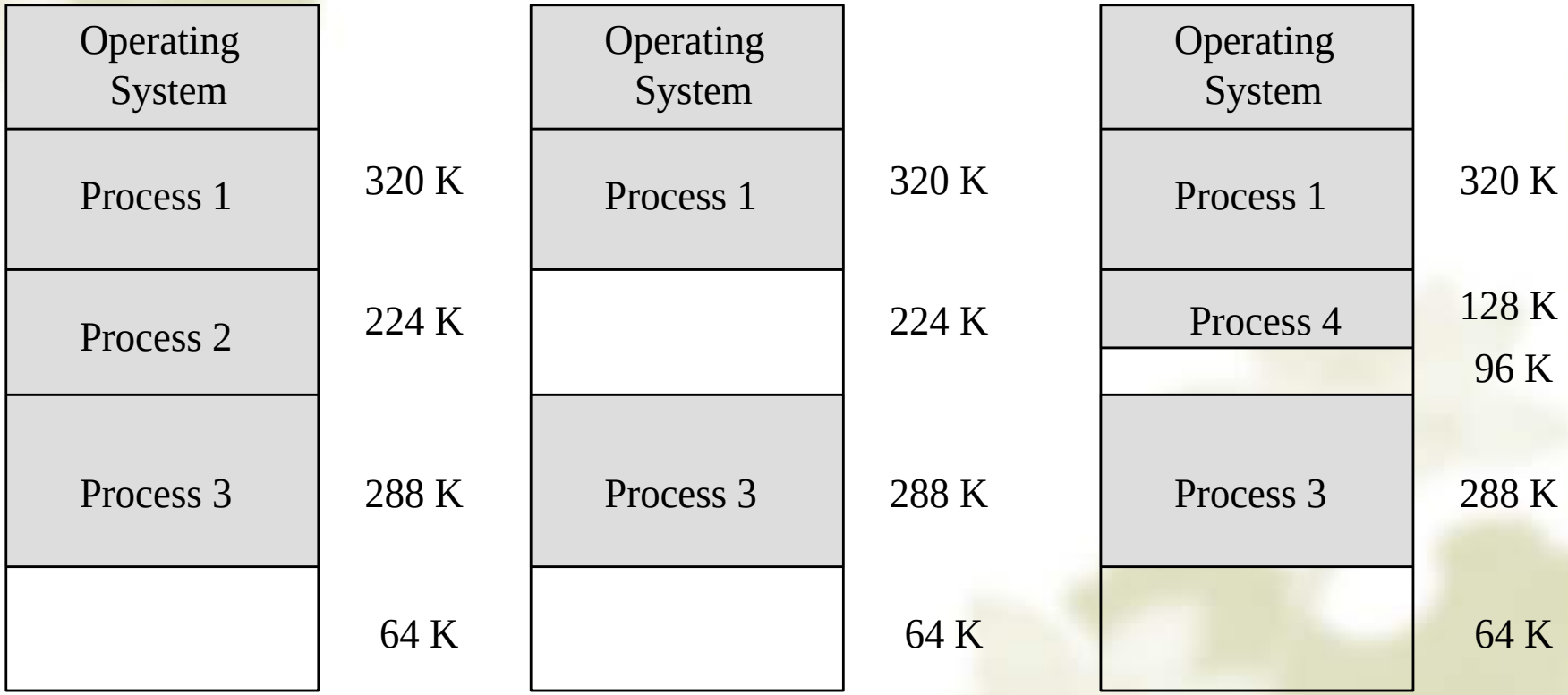
(a) Allocating space for growing data segment.

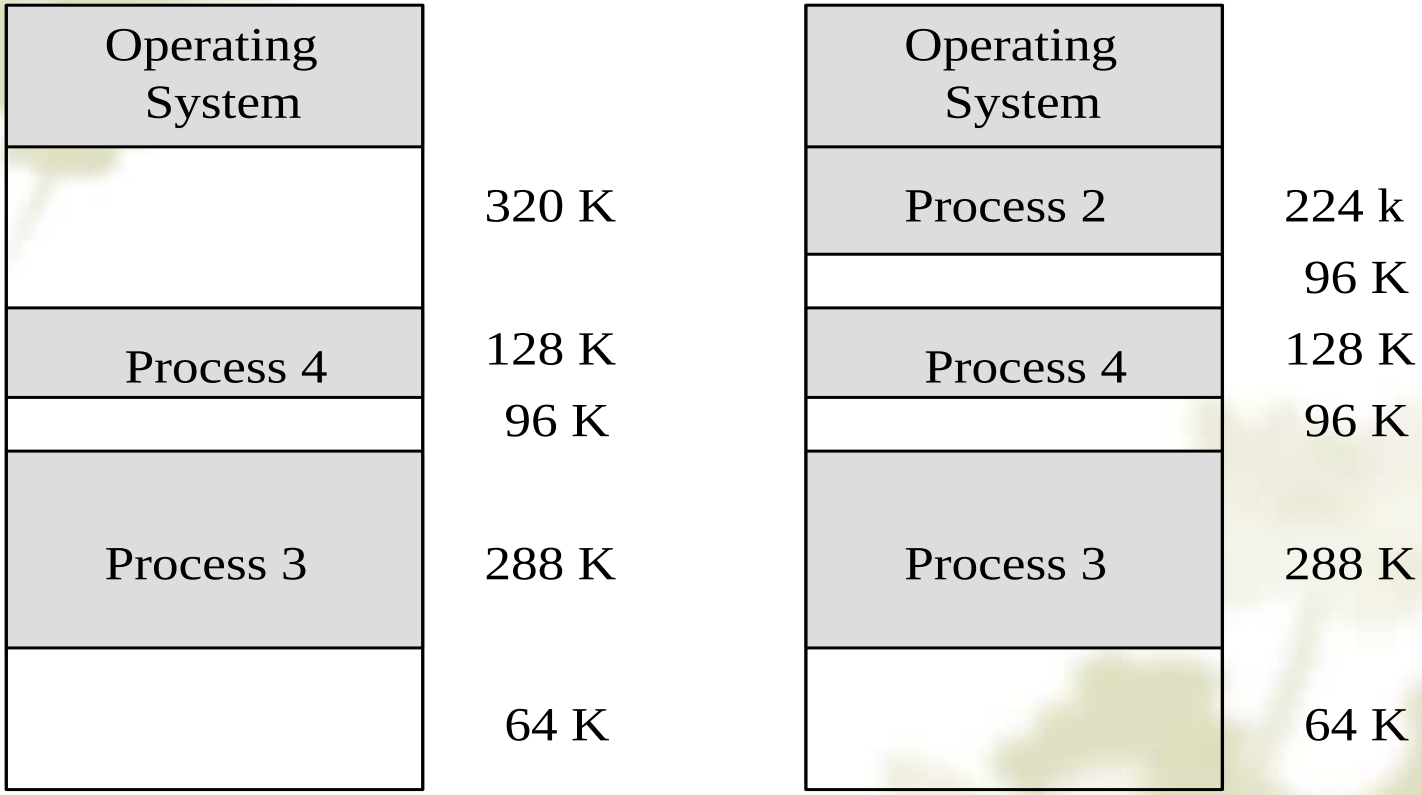
(b) Allocating space for growing stack, growing data segment.

Swapping (5)

- With processes swapping in and swapping out, there will be many **holes** (External Fragmentation)




Operating System
Process 1
320 K
Process 2
224 K
Process 3
288 K
64 K
Operating System
Process 1
320 K
224 K
Process 3
288 K
64 K
Operating System
Process 1
320 K
Process 4
128 K
96 K
Process 3
288 K
64 K

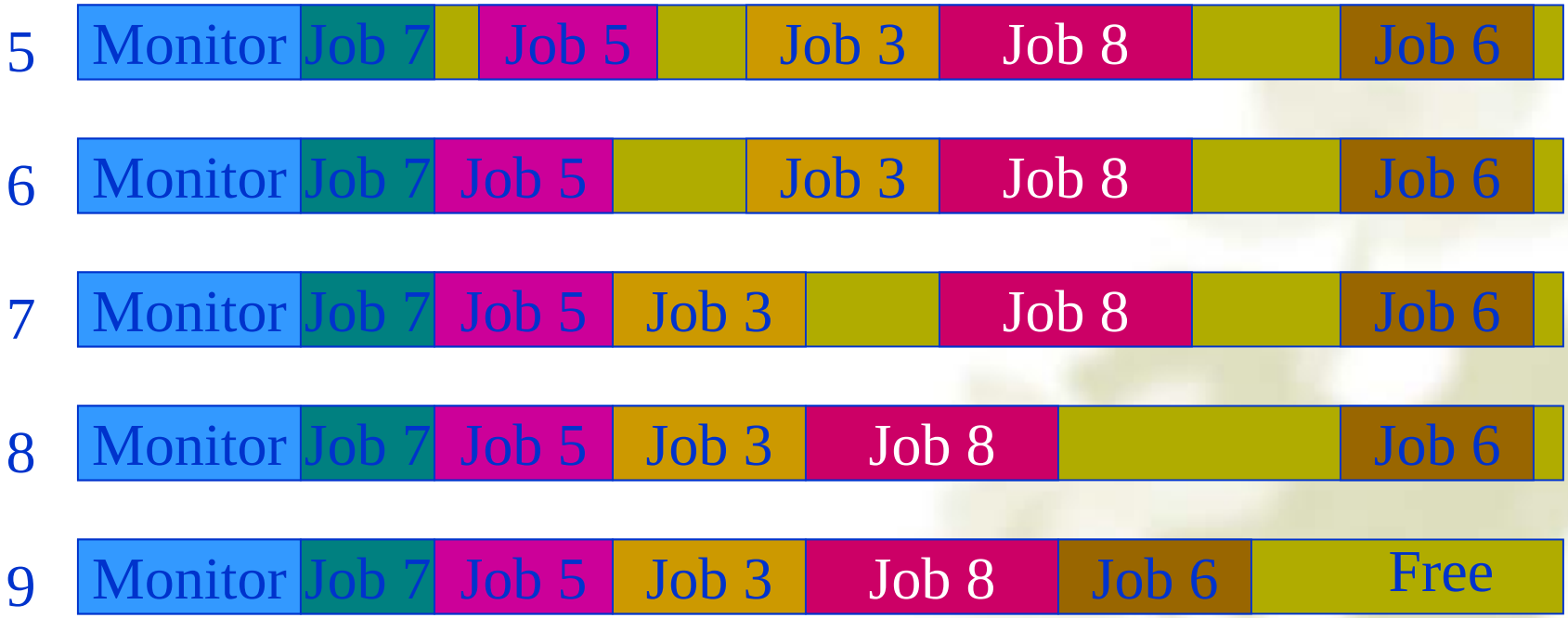


- The holes need be compacted together to generate a large free space.

Compaction

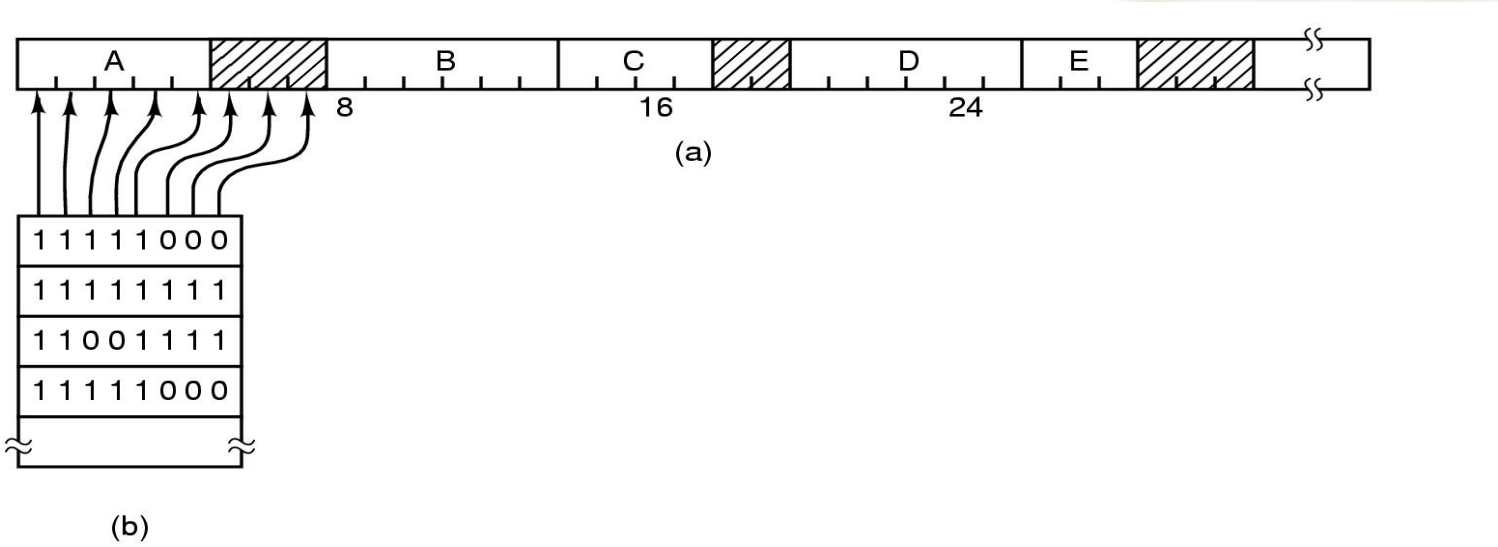
(Similar to Garbage Collection)

- Assumes programs are all re-locatable
- Processes must be suspended during compaction
- Need be done only when fragmentation gets very bad



Memory Management

- How to keep track of memory usage?
  - Bitmap or linked lists
- Memory is divided into allocation units.
- Bitmap
  - One allocation unit corresponds to 1bit in the bitmap
  - 0: free, 1: allocated

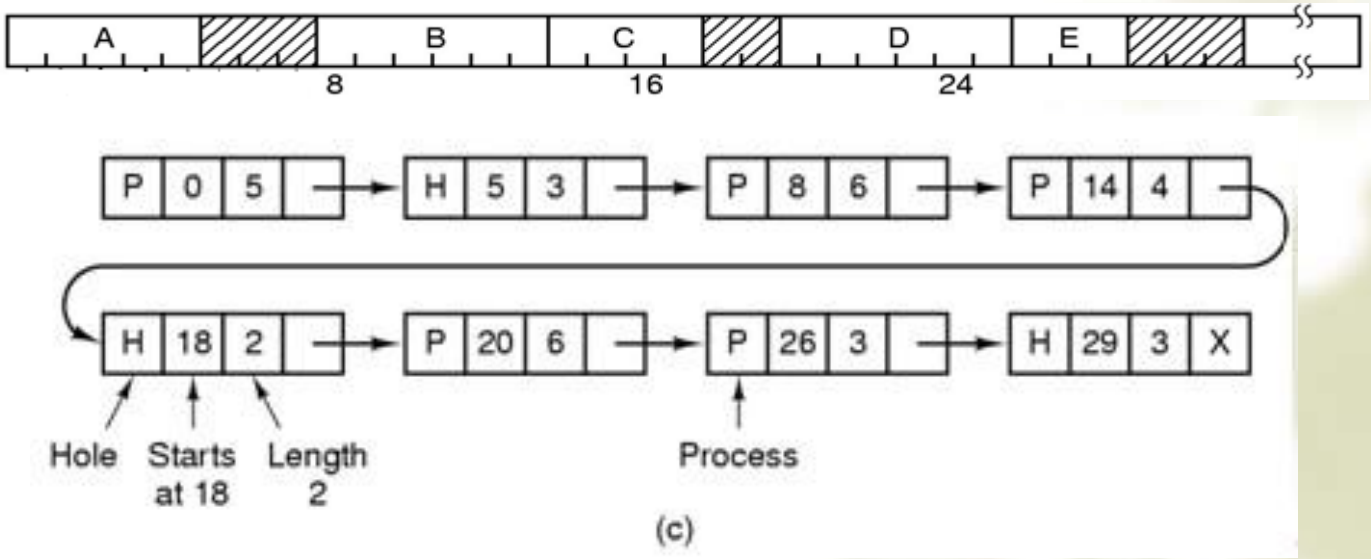


Memory Management with Bitmap

- Size of allocation unit
  - The smaller the allocation unit, the larger the bitmap.
- Problem: allocation
  - When a new process arrives, the manager must find consecutive 0 bits in the map.
  - Searching a bitmap for a run of a given length is a slow operation.

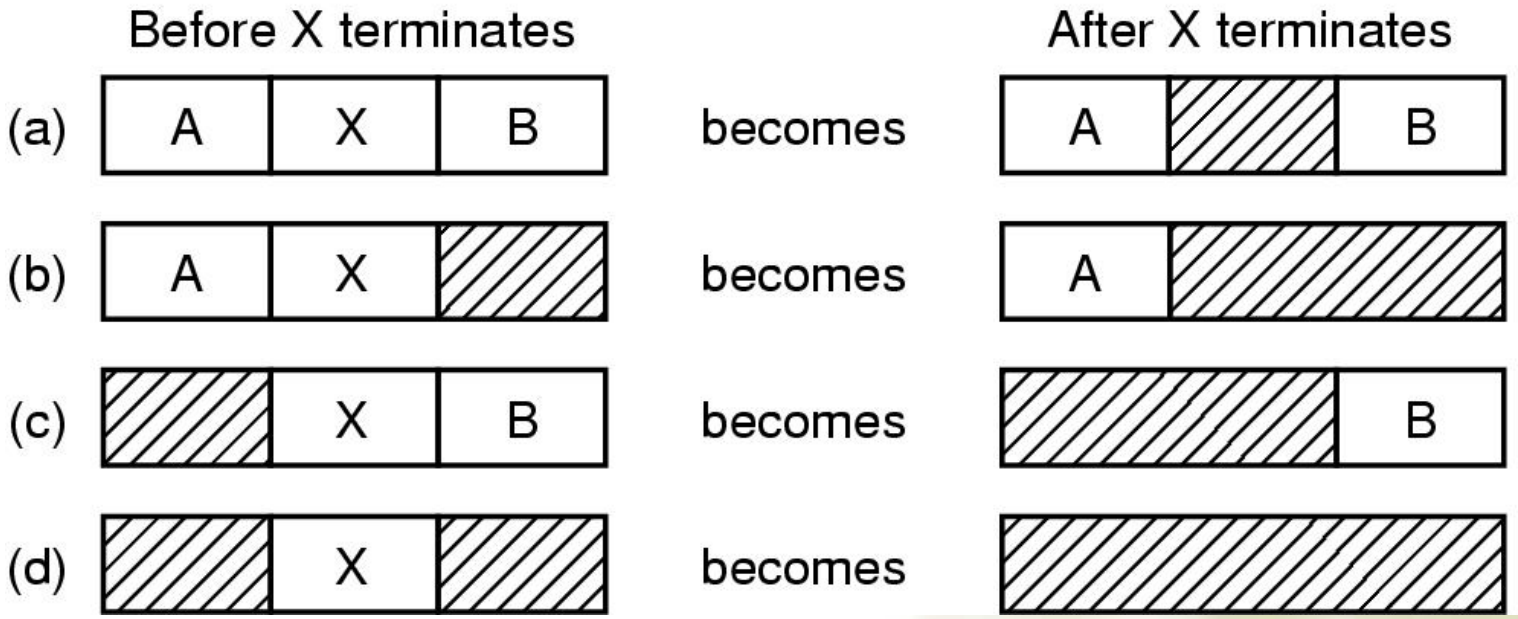
Memory Management

- Linked Lists
  - Each entry in the list specifies a hole (free segment) or a process (allocated segment), the address it starts, the length, and a pointer to the next entry.
  - sorted by the address or by the size



Memory Management with Linked Lists

- Sorting by address has the advantage:
  - When a process terminates or is swapped out, updating the list is straightforward.



Four neighbor combinations for the terminating process X

Storage Placement Strategies

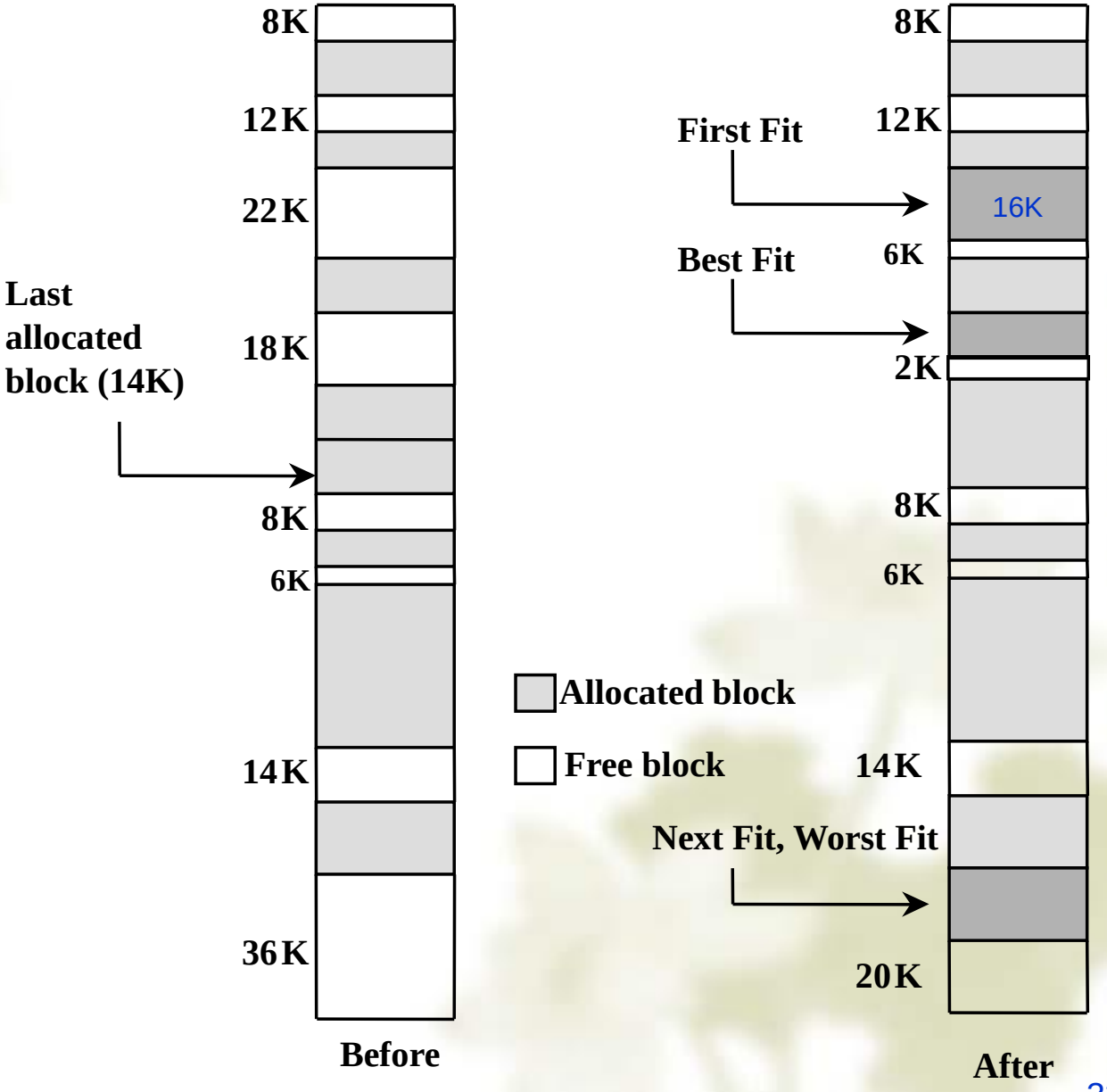
How to satisfy a request of size *n* from a list of free holes.

- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.
- **Next Fit**
  - Minor variation of first fit: search for the last hole stopped.
  - Problem: slightly worse performance than first fit.

Storage Placement Strategies

- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.
- **Worst Fit**
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.
- **Quick Fit**
  - maintains separate lists for some of the more common sizes requested.
  - When a request comes for placement it finds the closest fit.
  - This is a very fast scheme, but a merge is expensive.

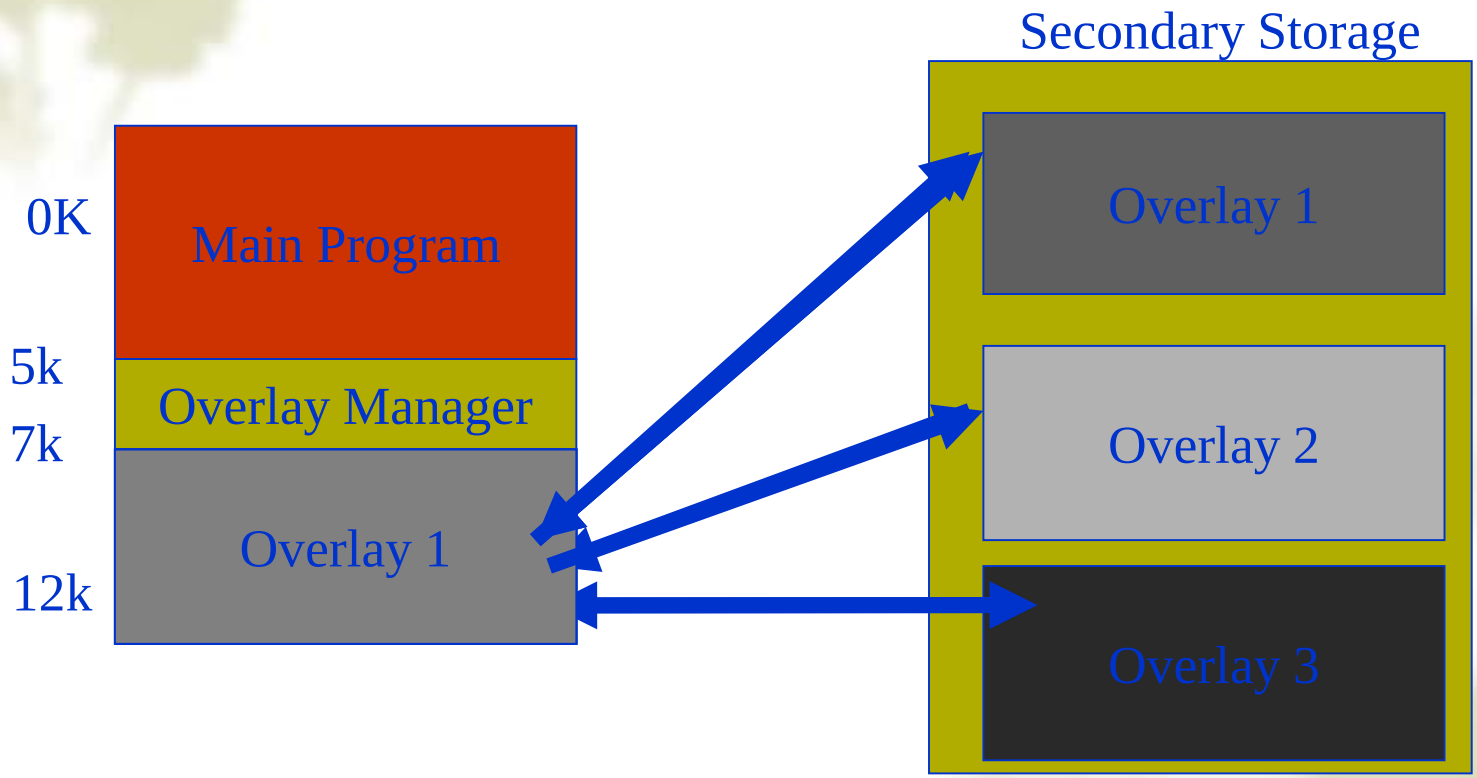
- Example:
  Request 16K



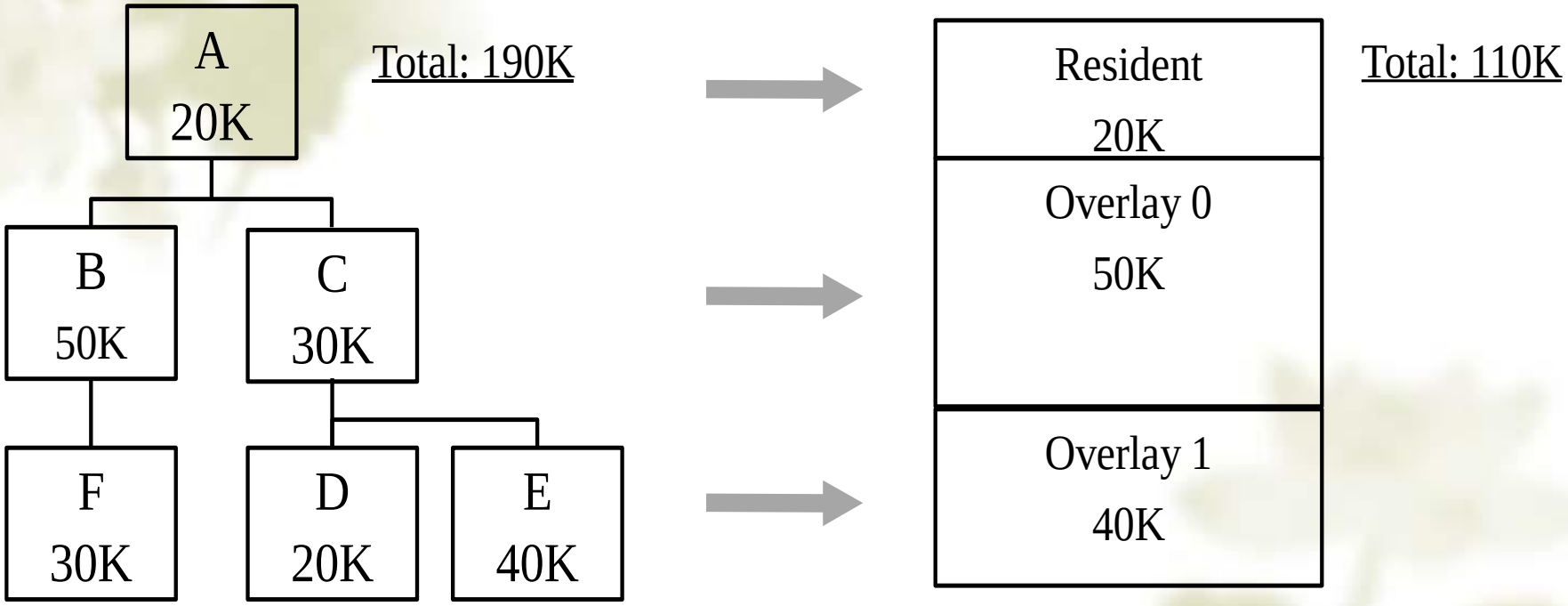
Overlaying

- One program, using lots of memory
  - Overlaying (1960s): split programs into little pieces, called overlays.
  - Allow one or some overlays in memory at the same time, many overlays can use common memory space. (see next page)
  - The work of swapping overlays in and out is done by OS, but the programmer should split the program into pieces manually, which increases the complexity of programming.

Overlaying


Secondary Storage
0K
Main Program
5k
Overlay Manager
7k
Overlay 1
12k
Overlay 1
Overlay 2
Overlay 3

Overlaying Example



An alternative approach：(100K)

- A(20K)： 20K；
- B(50K)、D(20K)、E(40K)： 50K；
- C(30K)、F(30K)： 30K；

Virtual Memory

- Virtual memory – separation of user logical memory from physical memory
  - Provide user with virtual memory that is as big as user needs
  - Logical address space can therefore be much larger than physical address space.
  - Store virtual memory on disk
  - Only part of the program needs to be in memory for execution.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
  - Load and store cached virtual memory without user program intervention

Principle of Locality

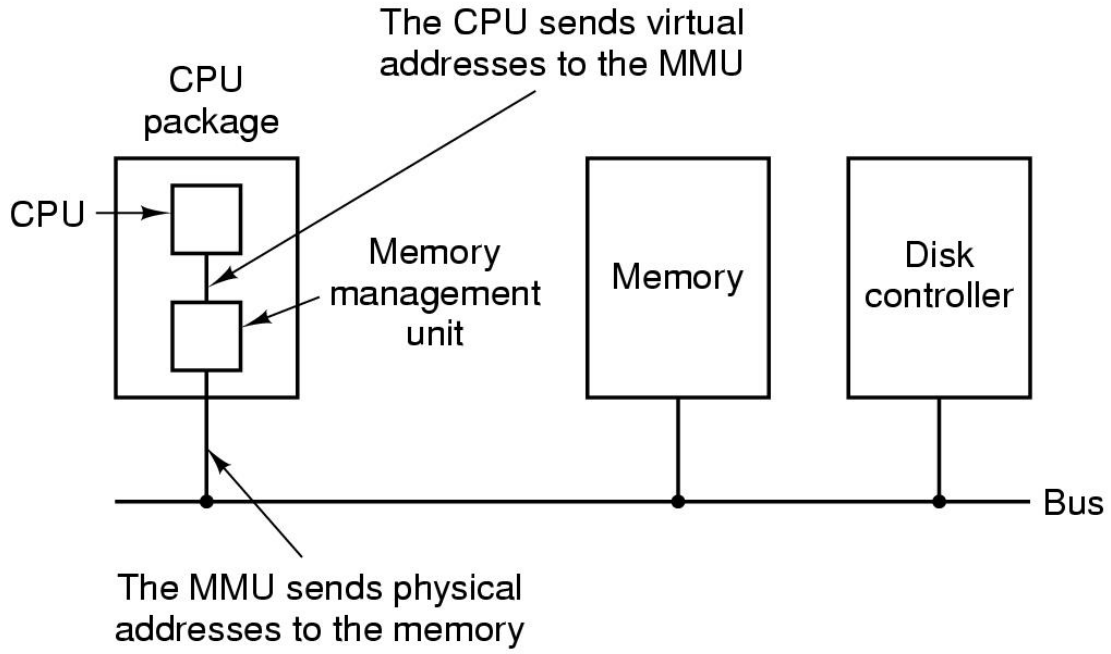
- Locality of reference:
  - During the phase of execution the process references relatively small fraction of its pages.
  - Time locality
  - Space locality

Implementation of Virtual Memory

- Paging
  - Modern approach
  - Paging is presently most common
- Segmentation
  - Basic early approach
  - Typically very simple
  - Probably better suited to behavior of process, although it can be harder to use and harder to implement
- Combined paging and segmentation

Paging

- Paging is a technique used to implement virtual memory.
  - Pages: the virtual address is divided up into units. Its size is power of 2, e.g. 512 bytes –64KB
  - Page frames: the corresponding units in the physical memory.
- The **MMU** (memory management unit) translates a virtual address into a physical address



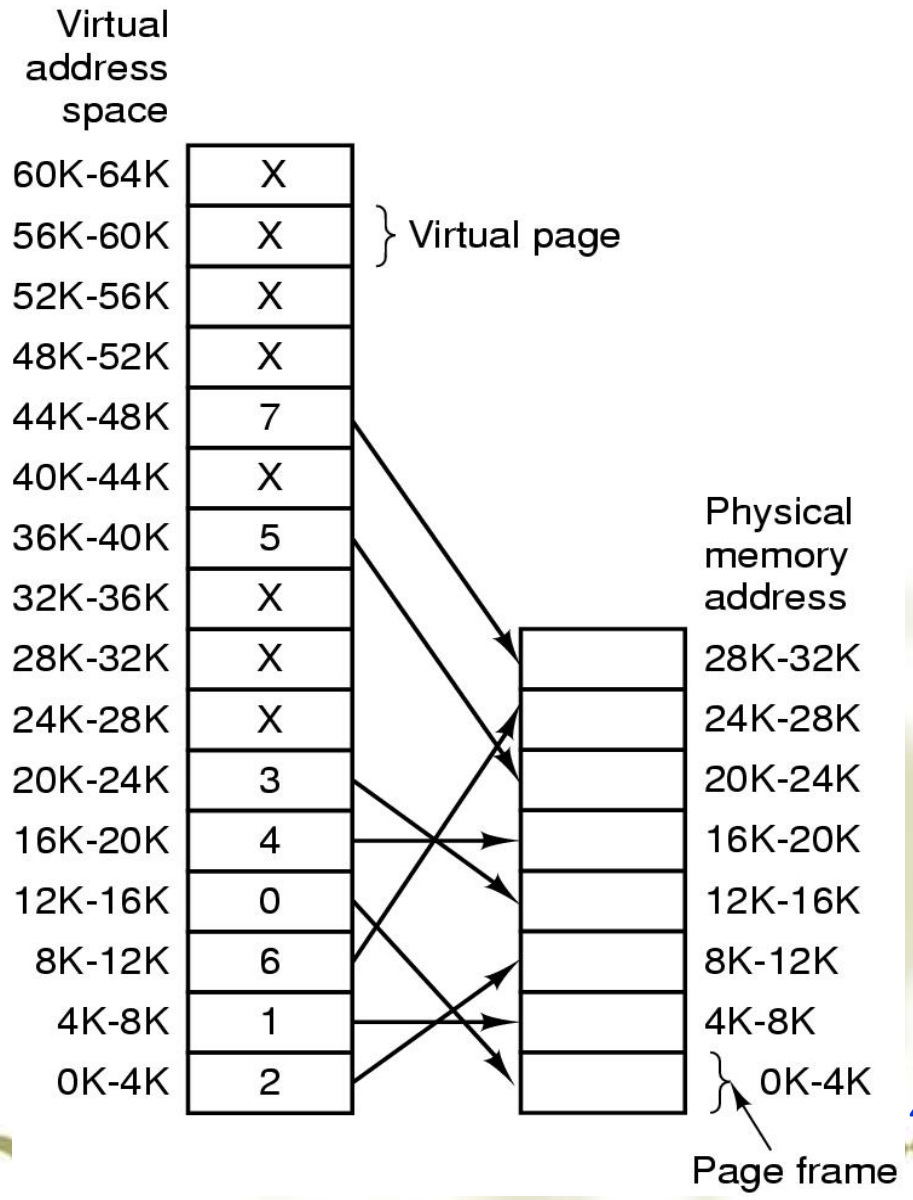
Paging

- A virtual address is a memory address that a process uses to access its own address space
  - The virtual address is not the same as the physical RAM address in which it is stored.
  - When a process accesses a virtual address, the MMU hardware translates the virtual address into a physical address
  - The OS determines the mapping from virtual address to physical address.

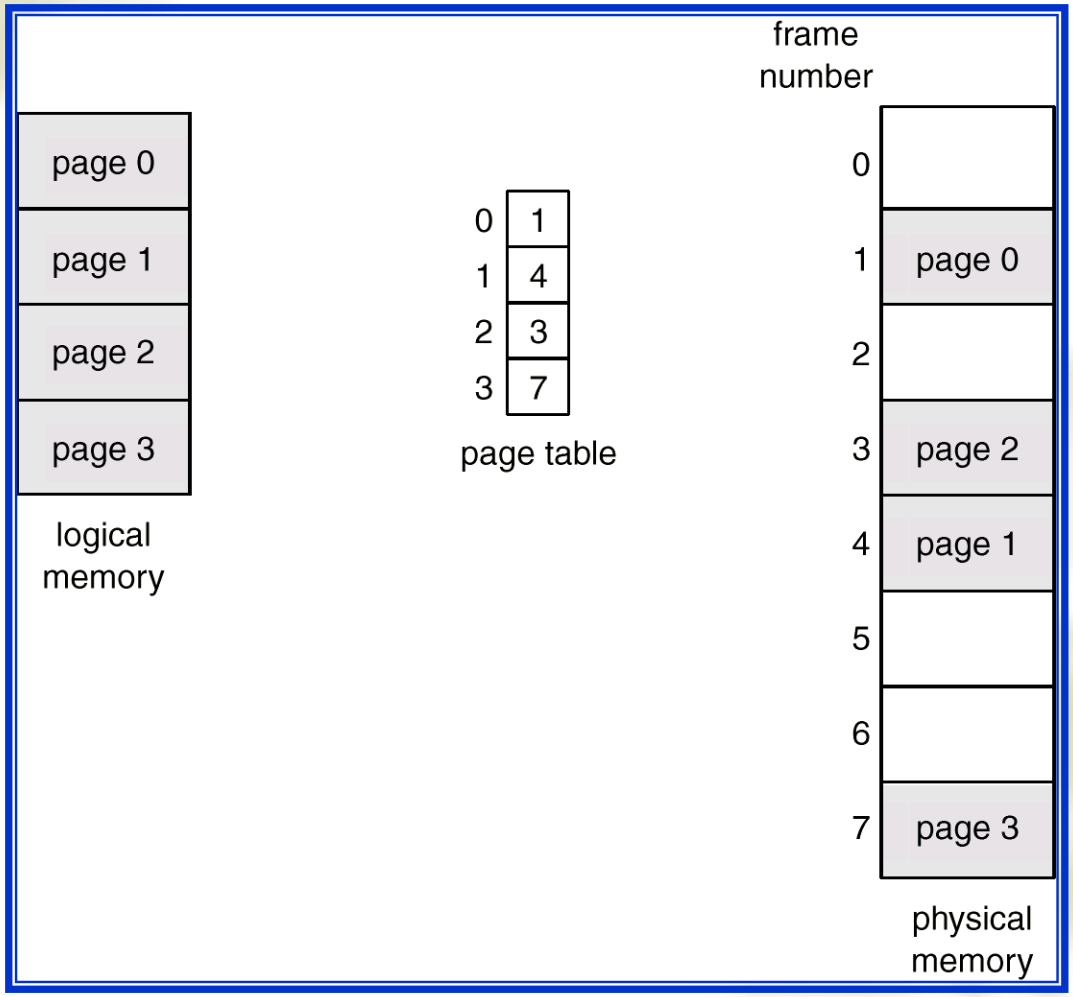
- Suppose the computer can generate 16-bit addresses, (0-64k). However, the computer only has 32k of memory ➔ 64k program can be written, but not loaded into memory.
- A Present/Absent bit keeps track of whether or not the page is mapped.
- Reference to an unmapped page causes the CPU to trap to the OS.
- This trap is called a Page fault. The MMU selects a little used page frame, writes its contents back to disk, fetches the page just referenced, and restarts the trapped instruction.

Paging

- The relation between virtual addresses and physical memory addresses given by **page table**.

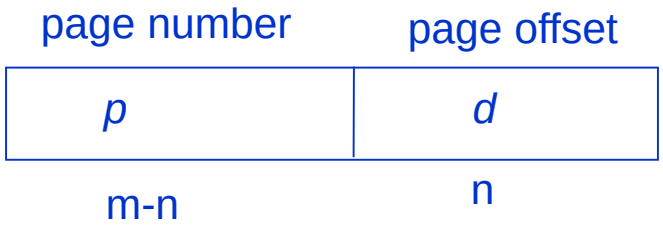


Paging Model Example



Paging

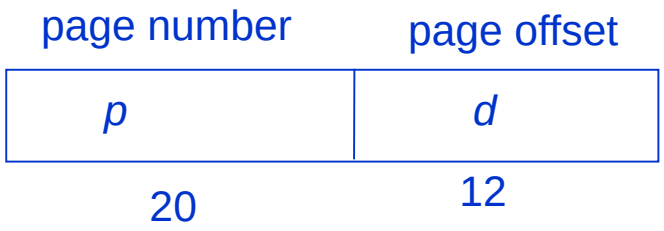
- Virtual address (generated by CPU) is divided into page number (high-order bits) and an page offset (low-order bits)
  - For a given logical address space 2^m and page size 2^n



  - Page number: Used as an index into the page table

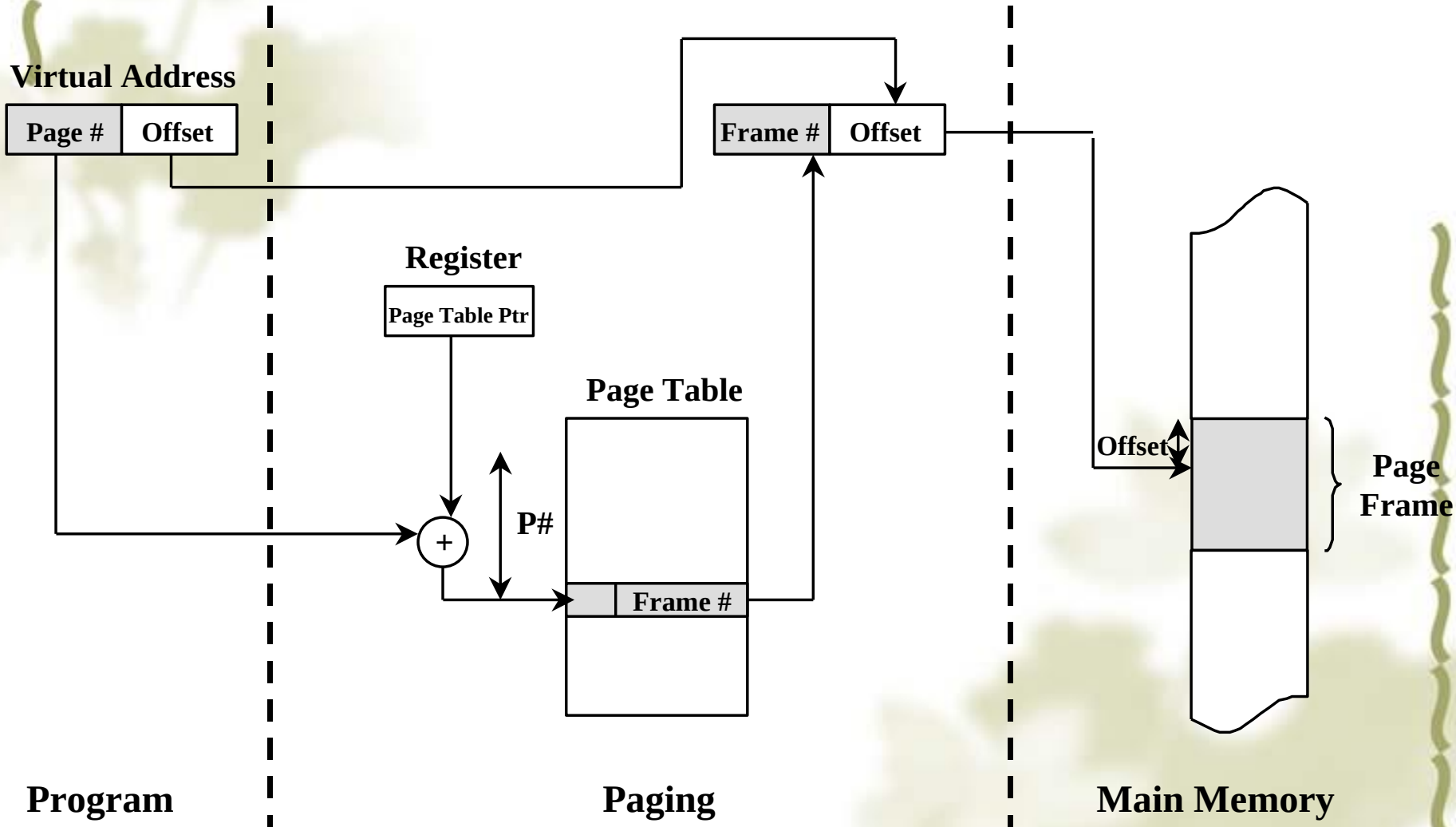
Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Thus, a logical address is as follows:

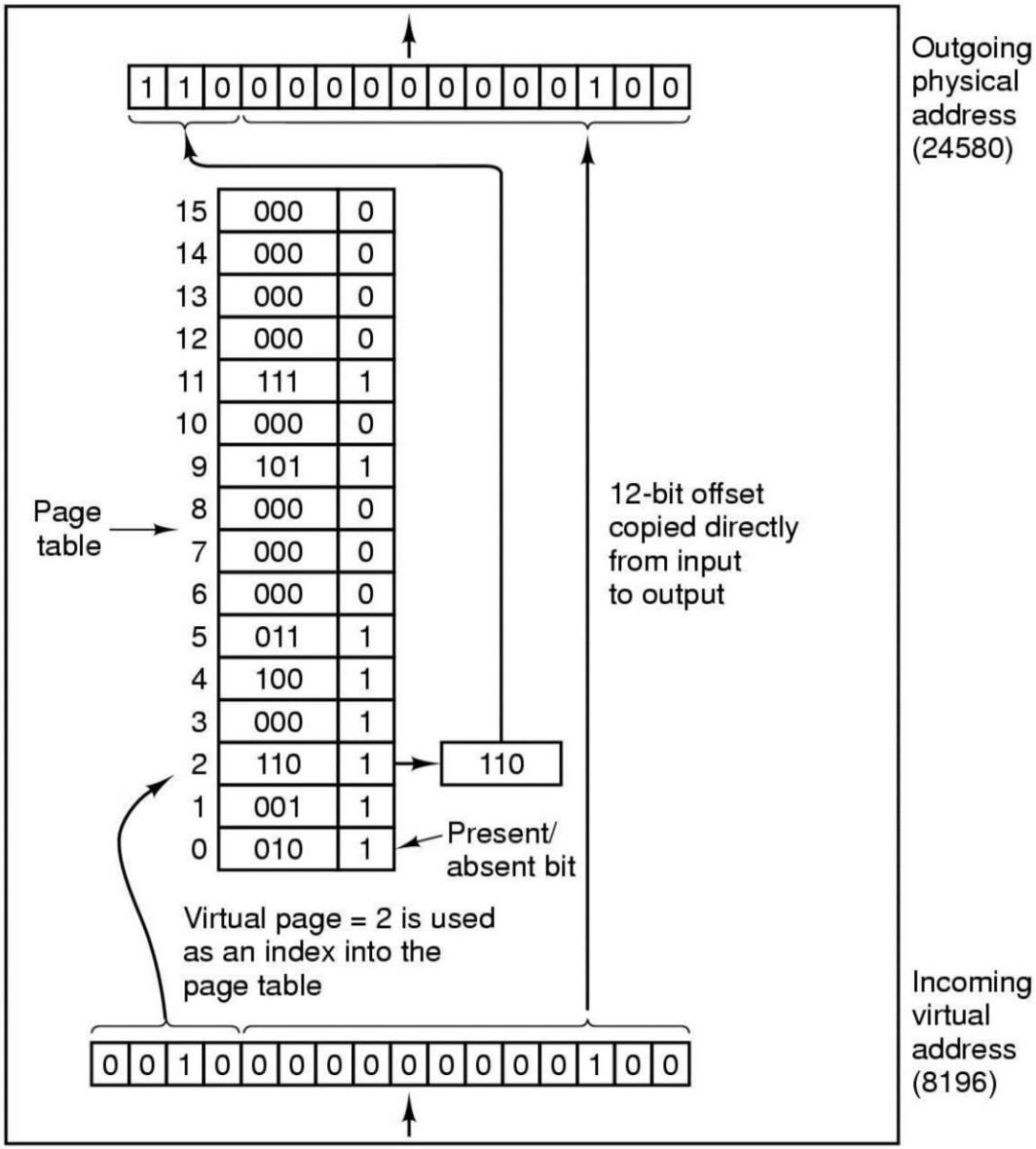


Paging

- Physical address is divided into
  - Frame number
  - Page offset
- The purpose of the page table is to map virtual pages onto page frames.


Virtual Address
Page #
Offset
Register
Page Table Ptr
Page Table
P#
+
Frame #
Offset
Frame #
Offset
Page
Frame
Program
Paging
Main Memory

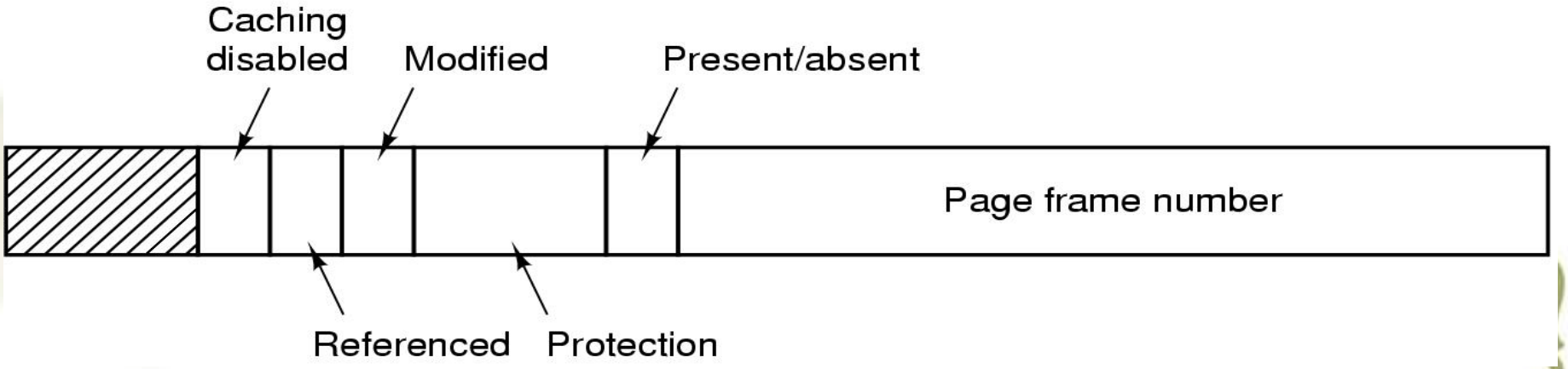
Example


Outgoing physical address (24580)
1 1 0 0 0 0 0 0 0 0 0 1 0 0
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

Page Table

- Managed by OS
- Map VPN (Virtual Page Number) to PFN
  - VPN is the index into the table that determines PFN
- One page table entry (PTE) per page in virtual address space, i.e. one PTE per VPN.
- Most operating systems allocate a page table for each process.

Typical page table entry



- Page frame number - map the frame number
- Present/absent bit - 1/0 indicates valid/invalid entry
- Protection bit - what kinds of access are permitted.
- Modified bit (**dirty bit**) - set when modified and writing to the disk occur
- Referenced bit - Set when page is referenced (help decide which page to evict)
- Caching disabled - Cache is used to keep data that logically belongs on the disk in memory to improve performance.

Page Tables Issues

- Two major issues of the page tables are faced
  - The mapping must be fast because it is done on every memory access!!
  - Page tables may be extremely large (e.g. most computers use)
    - 32-bit address with 4k page size, 12-bit offset
      - 20 bits for virtual page number
      - 1 million entries!

Issue 1 (Virtual-to-Physical must be fast)

- Single page table consisting of an array of hardware registers. As a process is started up, the registers are loaded with page table.
  - Advantage - simple
  - Disadvantage - expensive if table is large and loading the full page table at every context switch hurts performance.
- Leave page table in memory - a single register points to the table
  - Advantage - context switch cheap
  - Disadvantage - one or more memory references to read table entries

Virtual-To-Physical Lookups

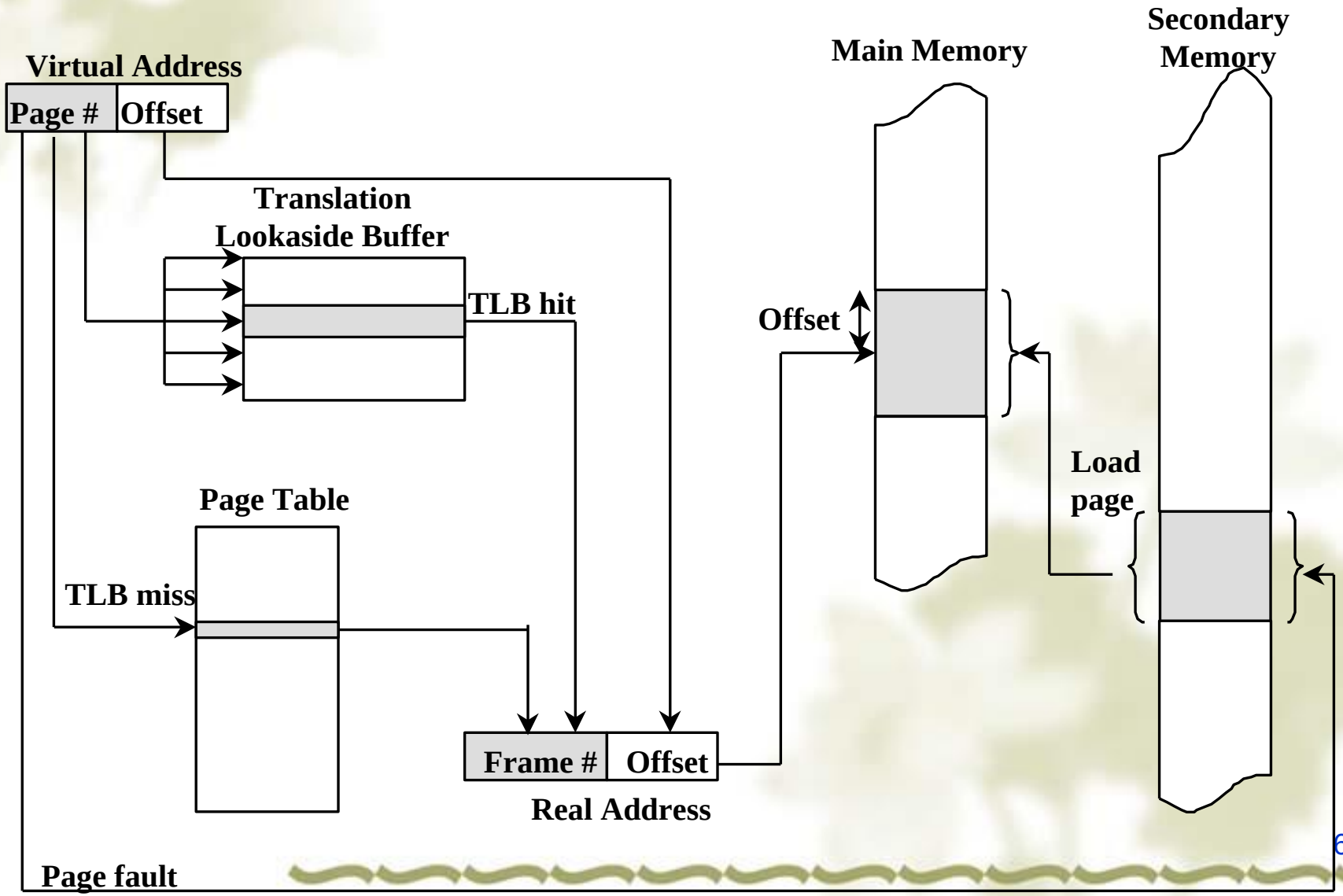
- Programs only know virtual addresses
- Each virtual address must be translated
  - May involve walking hierarchical page table
  - Page table stored in memory
  - So, each program memory access requires several actual memory accesses
- Solution: cache “active” part of page table
  - **Translation Look-aside Buffers *(TLBs)***
  - TLB also called “associative memory”

Bits in a TLB Entry

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - Physical page number: translated address
  - Valid
  - Access bits: kernel and user (nil, read, write)
- Optional (useful) bits
  - Process tag
  - Reference
  - Modify
  - Cacheable

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Translation Look-aside Buffer (TLB)



TLB Function

- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page table entry is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does an ordinary page table lookup.
  - It then evicts one page table entry out of TLB and replaces it with the new entry, so that next time that page table entry is found in TLB.
- TLB hit ratio (Page address cache hit ratio)
  - Percentage of time page table entry found in associative memory

Hardware-Controlled TLB

- On a TLB miss
  - Hardware loads the PTE (Page Table Entry) into the TLB
    - Need to write back if there is no free entry
  - Generate a fault if the page containing the PTE is invalid
  - VM software performs fault handling
  - Restart the CPU
- On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction

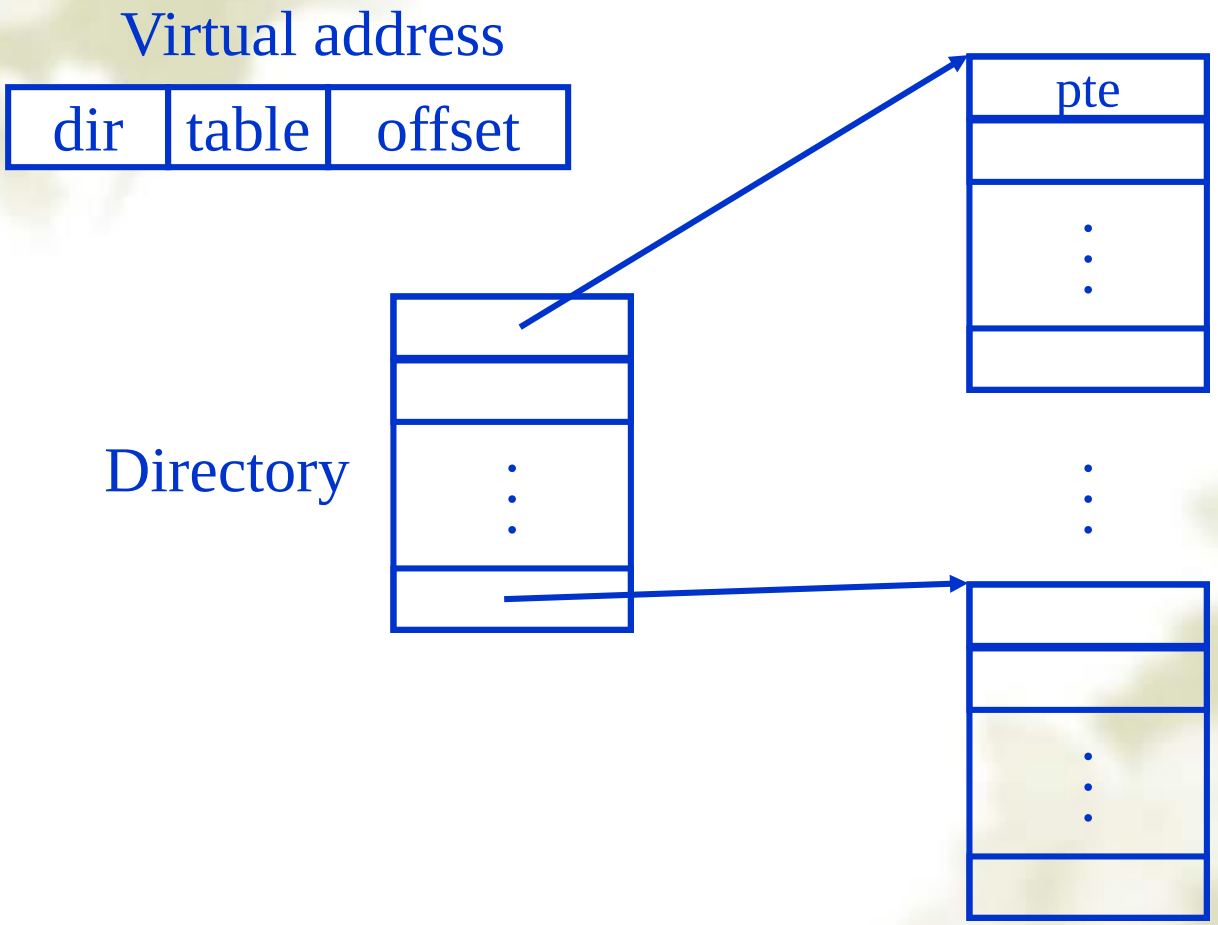
Software-Controlled TLB

- On a miss in TLB, generate a TLB fault, then trap to OS (software)
  - Check if the page containing the PTE is in memory
  - If no, perform page fault handling
  - Write back if there is no free entry, then load the PTE into the TLB
  - Restart the faulting instruction
- On a hit in TLB, the hardware checks valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction

Issue 2 (Page table may be large)

- Multilevel Page Tables
  - Since the page table can be very large, one solution is to page the page table
  - Divide the page number into
    - An index into a page table of second level page tables
    - A page within a second level page table
  - Advantage
    - reduce the table size.
    - don't keep page tables in memory that are not needed.

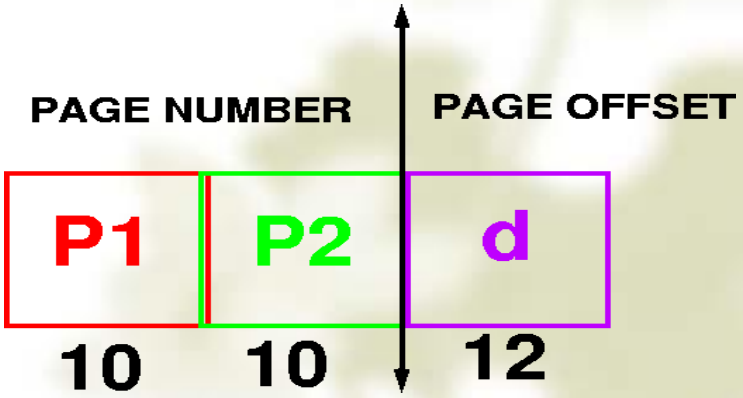
Multiple-Level Page Tables

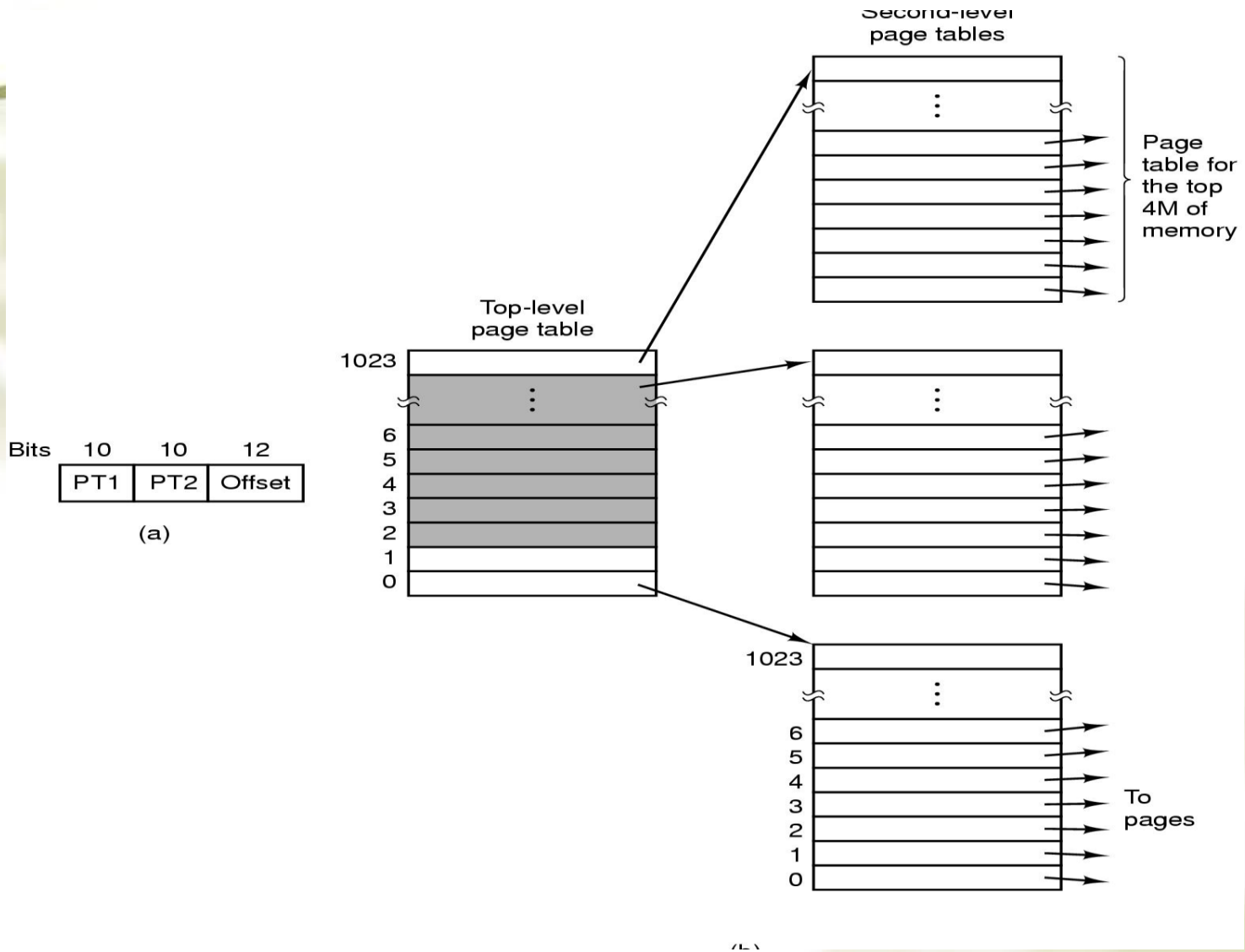


What does this by us? Sparse address spaces and easier paging

Example Addressing on a Multilevel Page Table System

- A logical address (on 32 bit machine with 4k page size) is divided into
  - A page number consisting of 20 bits
  - A page offset consisting of 12 bits
- Divide the page number into
  - A 10-bit page number
  - A 10-bit page offset





Performance: Since each level is stored as a separate table in memory, converting a logical address to a physical one in a four-level paging may take five memory accesses.

Inverted Page Tables

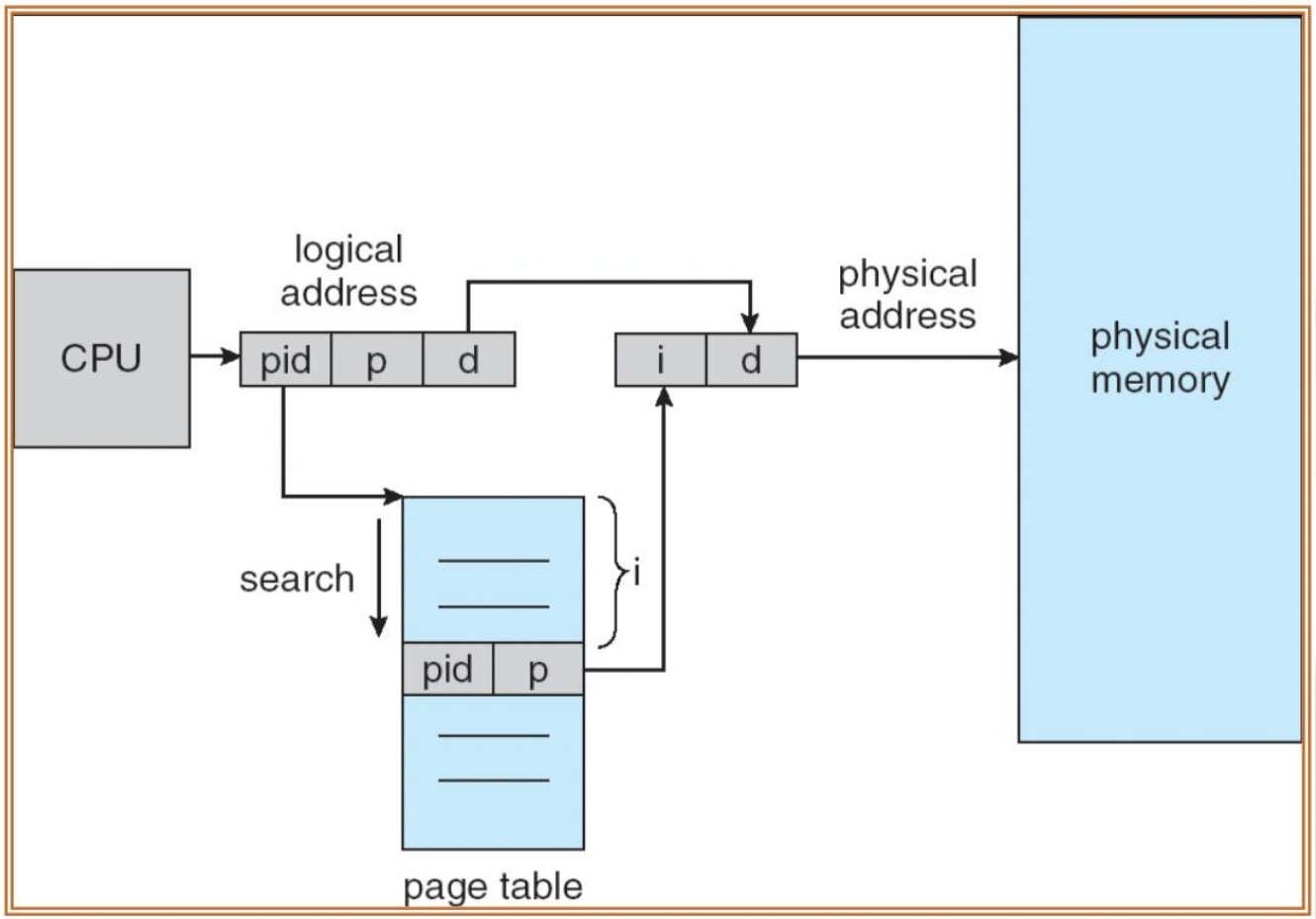
- Main idea
  - One PTE for each physical page frame
  - The physical page number is used as an index into the table
  - Hash (Vpage, pid) to Ppage#
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
  - Overhead of managing hash chains, etc

Linear Inverted Page Tables

- One global page table
  - One entry for each physical page frame
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. (process, virtual page)
  - The physical page number is used as an index into the table

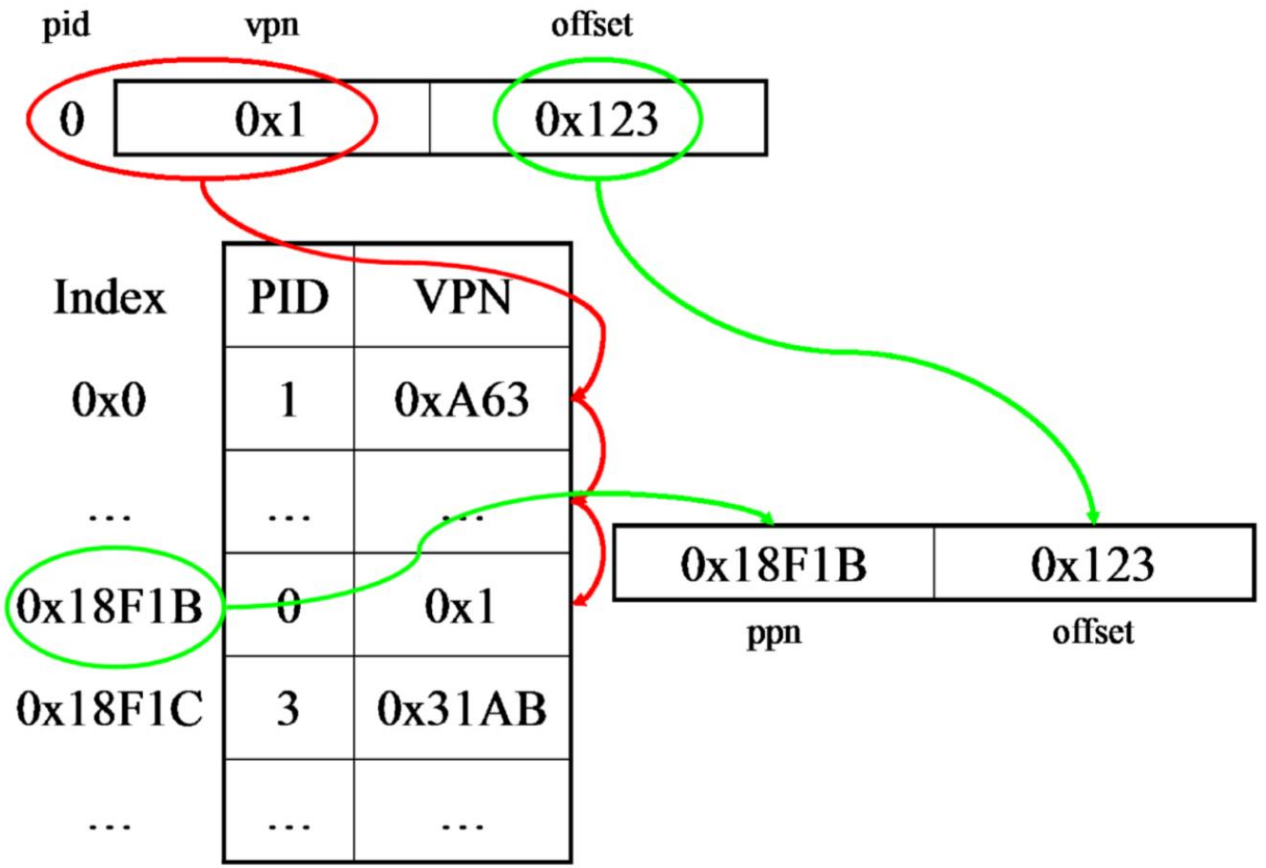
Linear Inverted Page Tables

- Address Translation Scheme



Linear Inverted Page Tables

- Example



Linear Inverted Page Tables

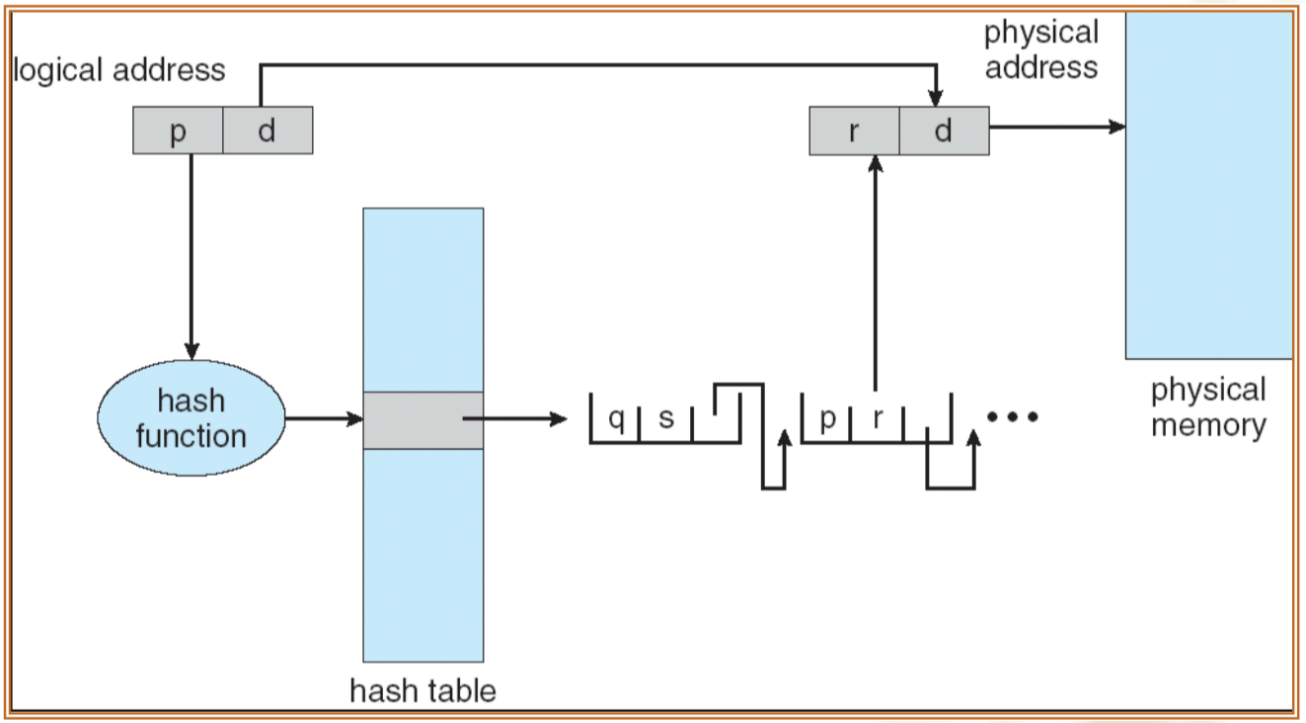
- Pros
  - Small page table for large address space
- Cons
  - Lookup is difficult
    - E.g. In the previous example, the worst scenario, will search for 2^{17} entries of the inverted page table.

Hashed Inverted Page Tables

- Add an extra level before the actual page table, called a hash table.
  - The process ID and virtual page number are hashed to get an entry in the hash table
  - When hashing with hash table, there may be conflicts, which can be solved by using chain address method.
  - Add the next field in the inverted page table items to form a linked list (the index of the header is in the hash table)

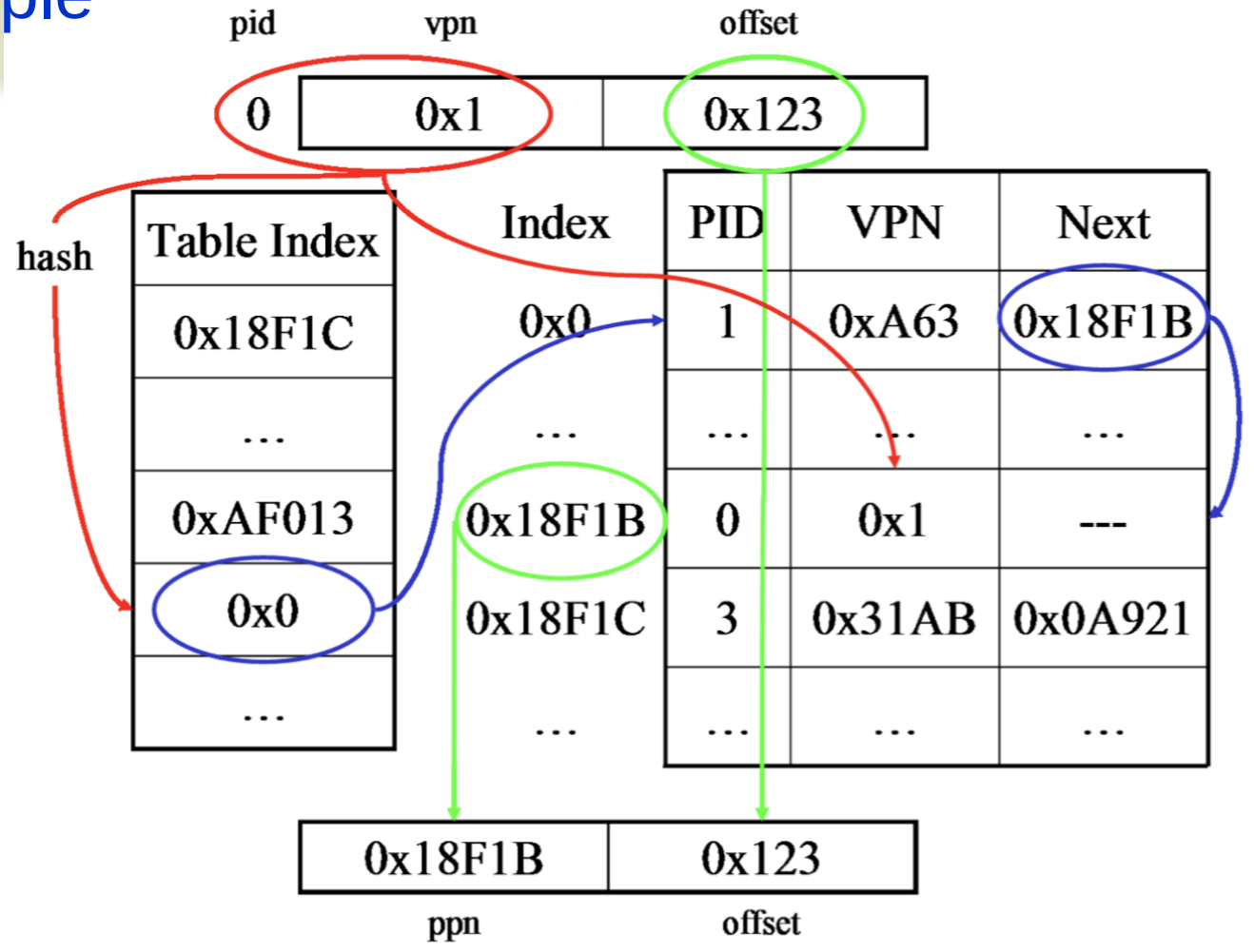
Hashed Inverted Page Tables

- Lookup in hash table for page table entry
  - Compare process ID and virtual page number
    - If match, then found
    - If not match, check the next pointer for another page table entry and check again

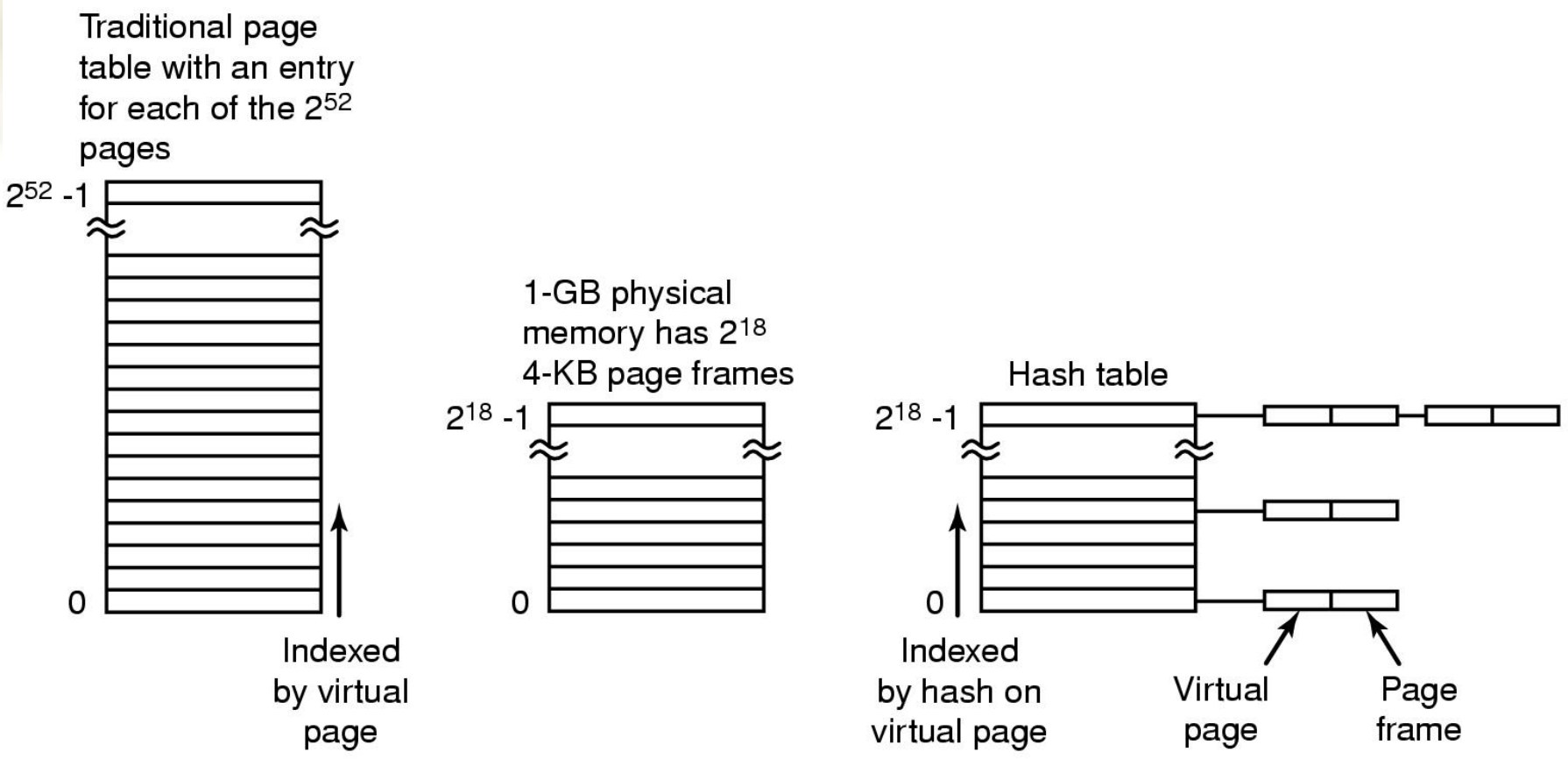


Hashed Inverted Page Tables

- Example



Hashed Inverted Page Tables



Comparison of traditional page table with an inverted page table

Hashed Inverted Page Tables

- Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread.
- Pros
  - With a good hashing scheme and a hashmap proportional to the size of physical memory, O(1) time. Very efficient!
- Cons
  - Overhead of managing hash chains, etc