Memory Management

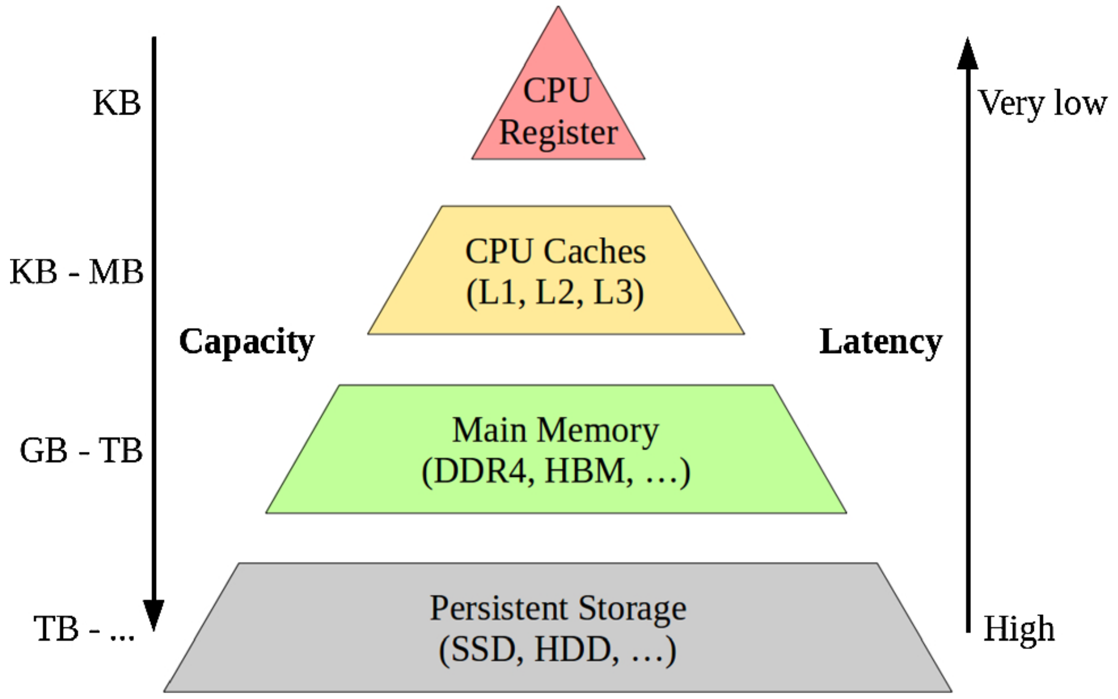
Operating Systems

Why Memory Management?

- *‘‘Programs expand to fill the memory available to hold them.’’*
- To provide a convenient abstraction for programming
- To allocate scarce memory resources among competing processes to maximize performance with minimal overhead
- To utilise concepts of physical and virtual memory to increase available memory

Memory Management

- **Memory hierarchy**
  - a few megabytes of very fast, expensive, volatile cache memory
  - a few gigabytes of medium-speed, medium-priced, volatile main memory
  - a few terabytes of slow, cheap, non-volatile magnetic or solid-state disk storage
- **Memory manager**
  - part of the operating system that manages the memory hierarchy

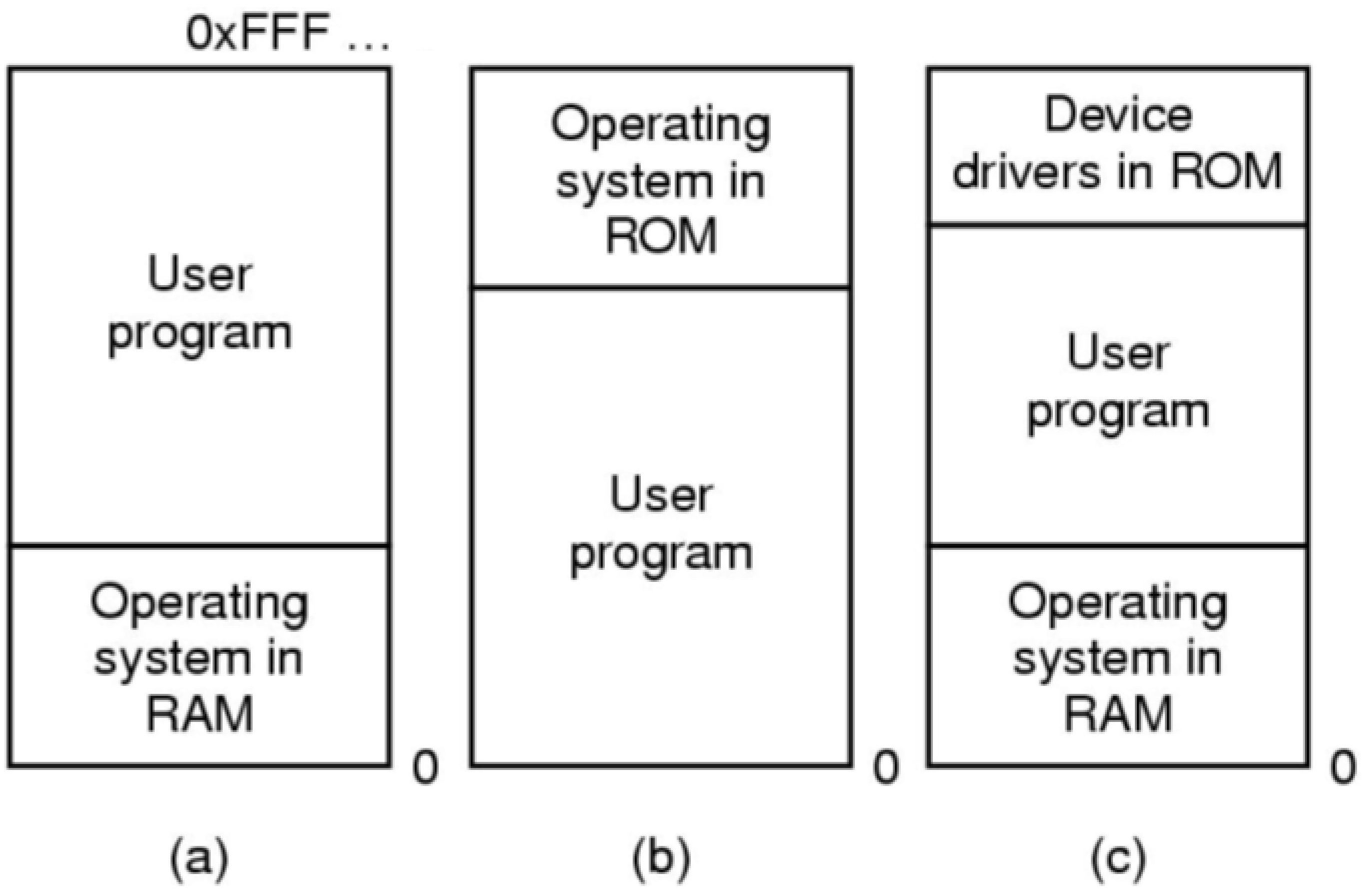


No Memory Abstraction

- Only physical memory is used
- Only one process running at a time
  a) Operating system at the bottom of memory in RAM
     - Early mainframes and minicomputers

  b) Operating system in ROM at the top of memory
     - Some handheld computers and embedded systems

  c) Device drivers at the top of memory in a ROM and the rest of the system in RAM down below
     - Early personal computers, with BIOS on ROM

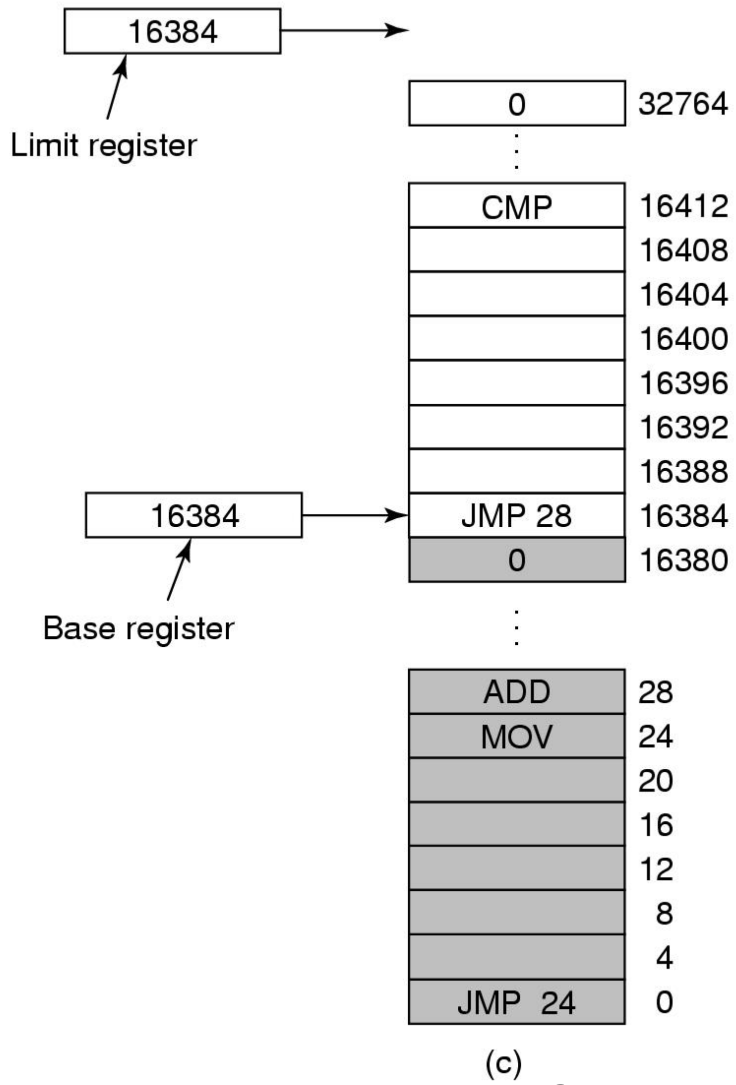


Memory Abstraction – Address Spaces

- Solves two problems to allow multiple applications to be in memory at the same time without interfering with each other
  - Protection
  - Relocation
- An **address space** is the 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.
- E.g. Telephone numbers with extensions

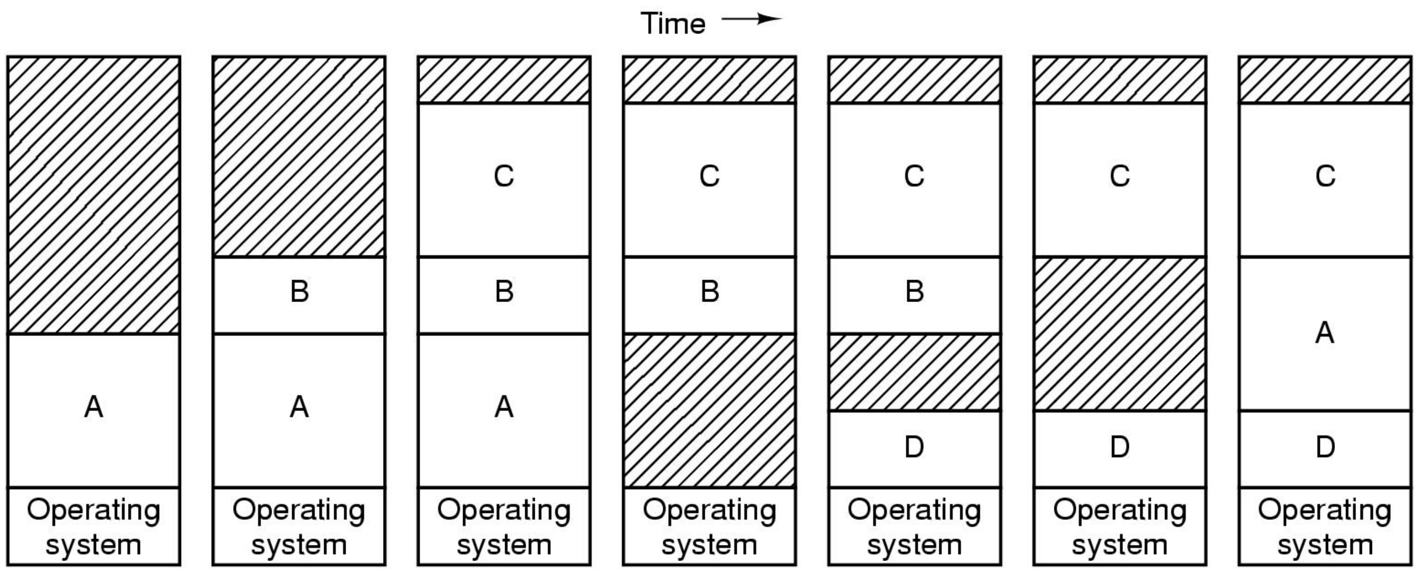
Base and Limit Registers

- Map each process' address space onto a different part of physical memory
- CPU with special hardware registers:
  - Base register – loaded with the physical address where its program begins in memory
  - Limit register – loaded with the length of the program
- A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference.



Swapping

- The total amount of RAM needed by all the processes is often much more than can fit in memory.
- **Swapping** consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.



Virtual Memory

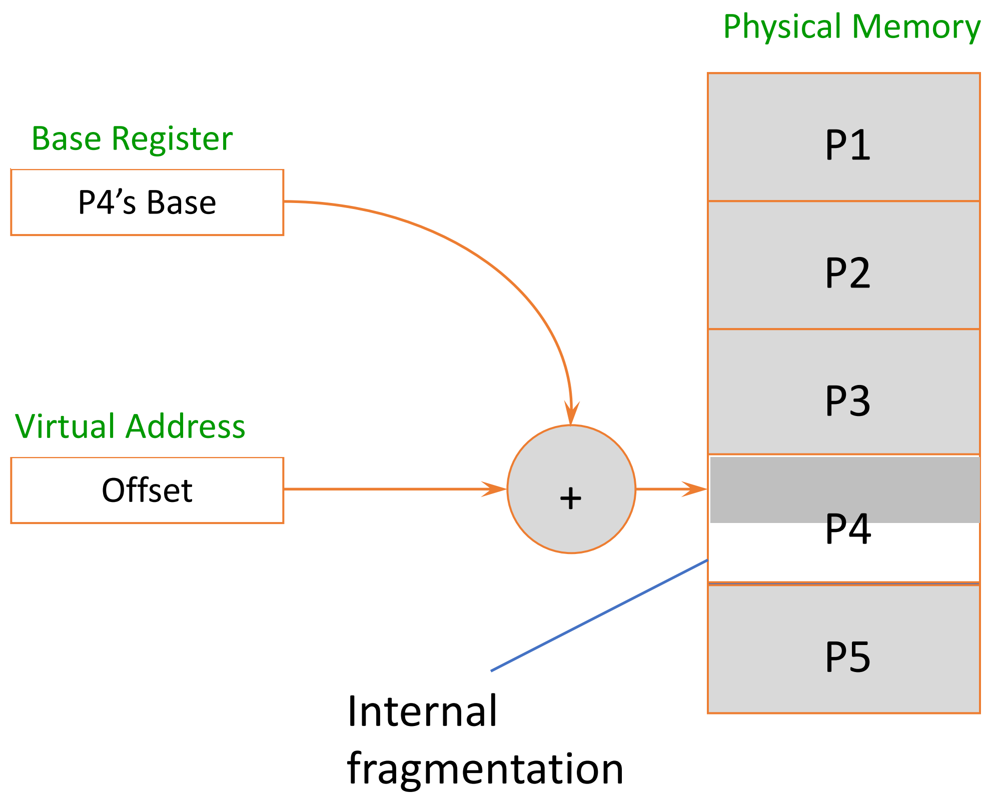
- Need to run programs that are too large to fit in memory
- Need to have systems that can support multiple programs running
- Swapping is not an attractive option because of speed
- Virtual memory (VM) is abstraction that the OS will provide for managing memory
  - Enables a program to execute with less than its complete data in physical memory
  - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
- OS will adjust amount of memory allocated to a process based upon its behaviour
- VM requires hardware support and OS management algorithms to pull it off

Virtual Addresses

- Virtual addresses are independent of the actual physical location of the data referenced
- OS determines location of data in physical memory
- Instructions executed by the CPU issue virtual addresses
- Virtual addresses are translated by hardware into physical addresses (with help from OS)
- The set of virtual addresses that can be used by a process comprises its virtual address space

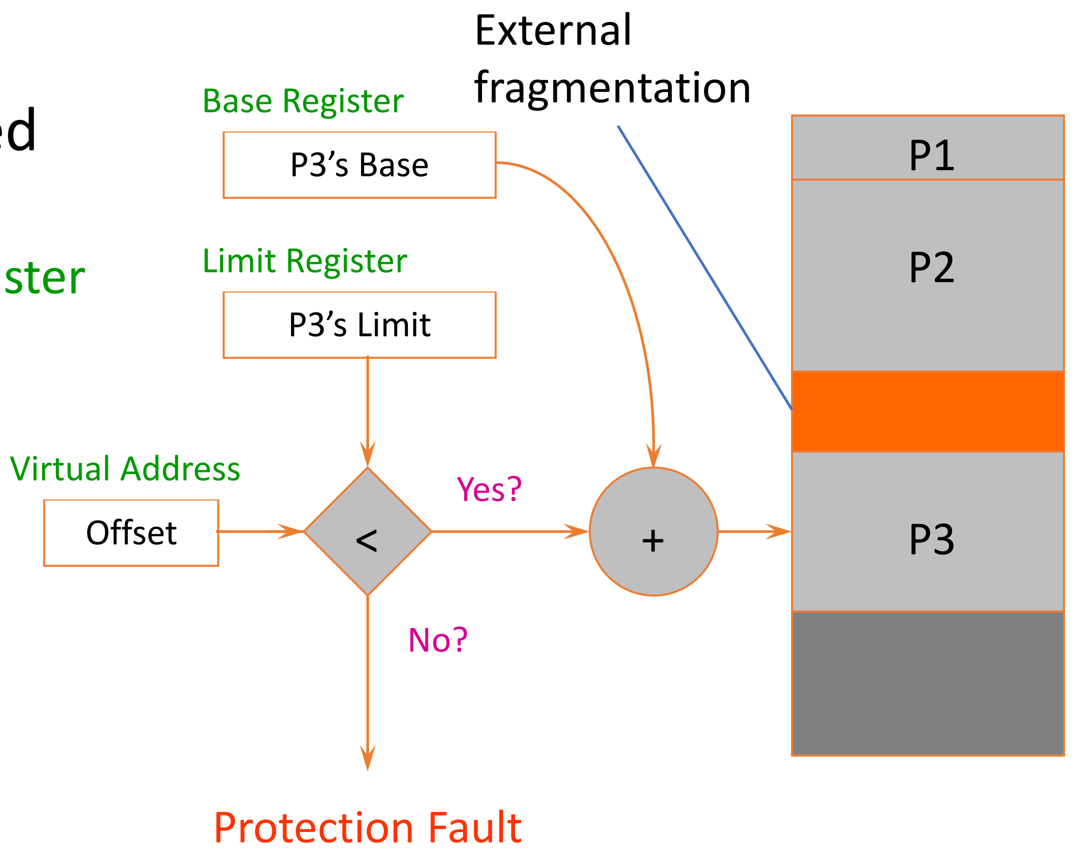
Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process
  - Size of each partition is the same and fixed
- Advantages
  - Easy to implement, fast context switch
- Problems
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)



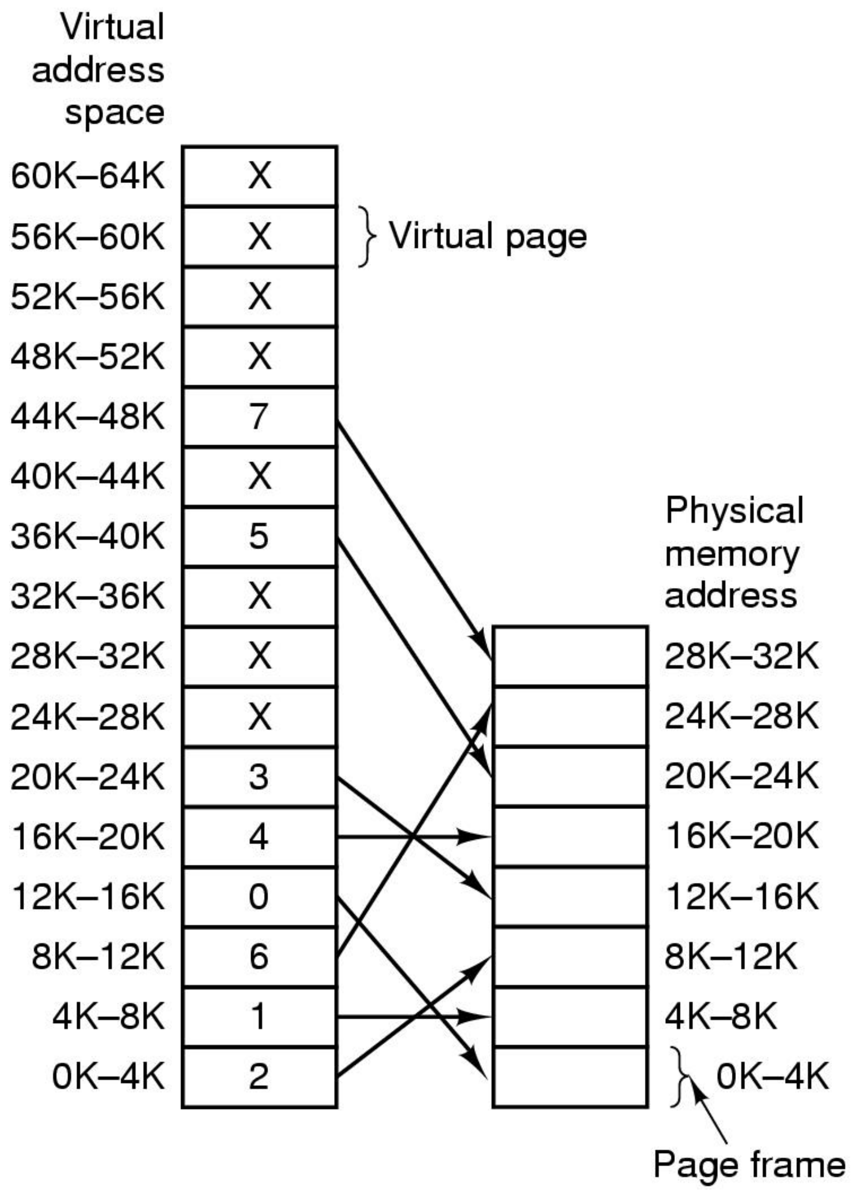
Variable Partitions

- Physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register
  - Why do we need the limit register? Protection
    - If (physical address > base + limit) then exception fault
- Advantages
  - No internal fragmentation
    - allocate just enough for process
- Problems
  - External fragmentation
    - job loading and unloading produces empty holes scattered throughout memory



Paging

- The virtual address space consists of fixed-size units called pages.
- The corresponding units in the physical memory are called **page frames**.
- The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page.
- 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.

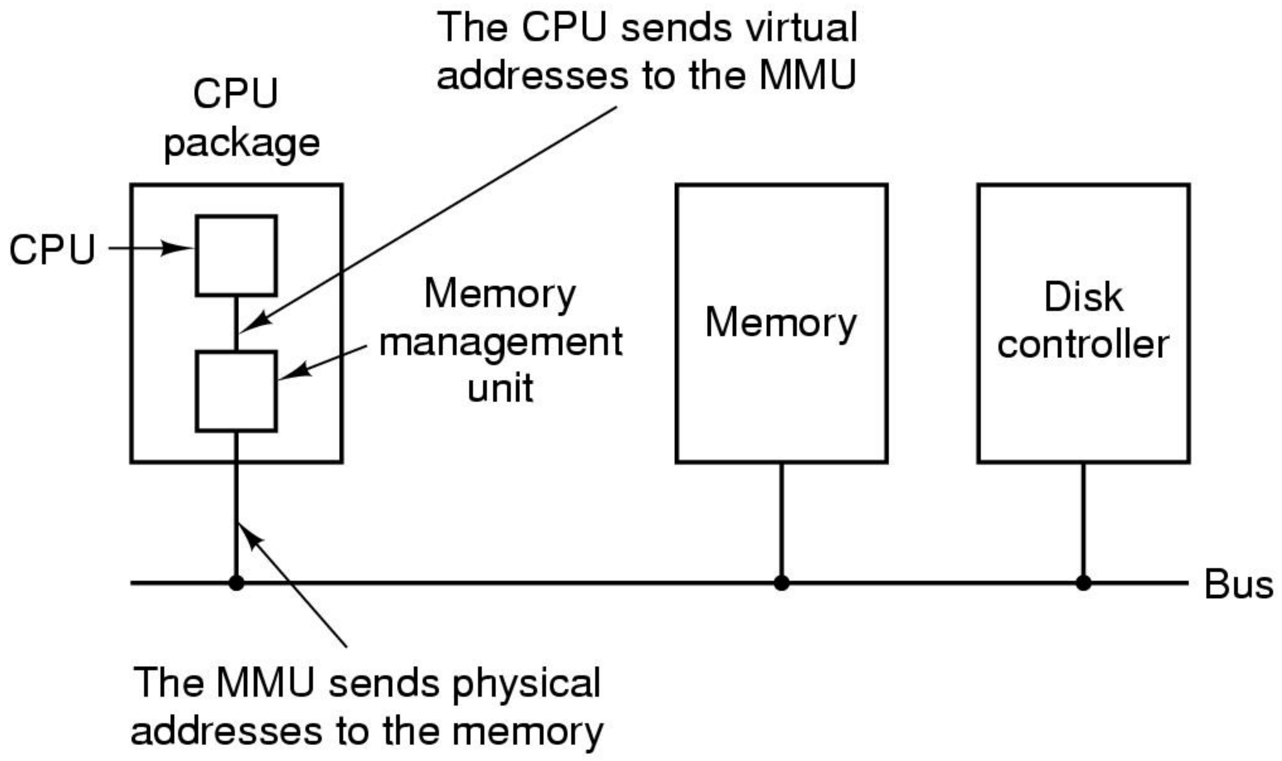


Paging

- How paging can solve fragmentation problems?
  - External fragmentation: can be solved by re-mapping between VA and PA
  - Internal fragmentation: can be solved if the page size is relatively small

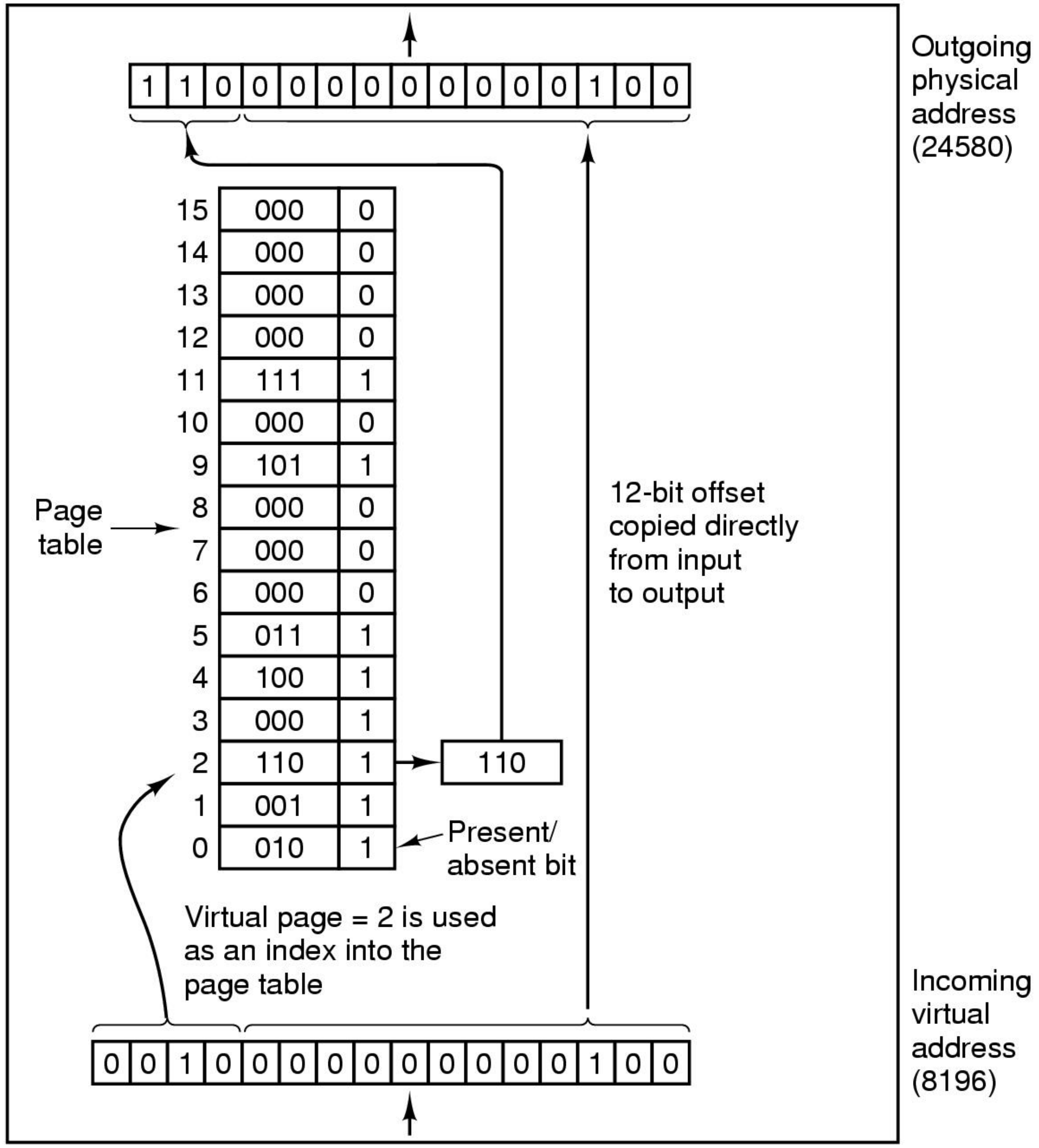
Memory Management Unit (MMU)

- When the program tries to access address 0, virtual address 0 is sent to the MMU.
- The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287).
- It thus transforms the address to 8192 and outputs address 8192 onto the bus.


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

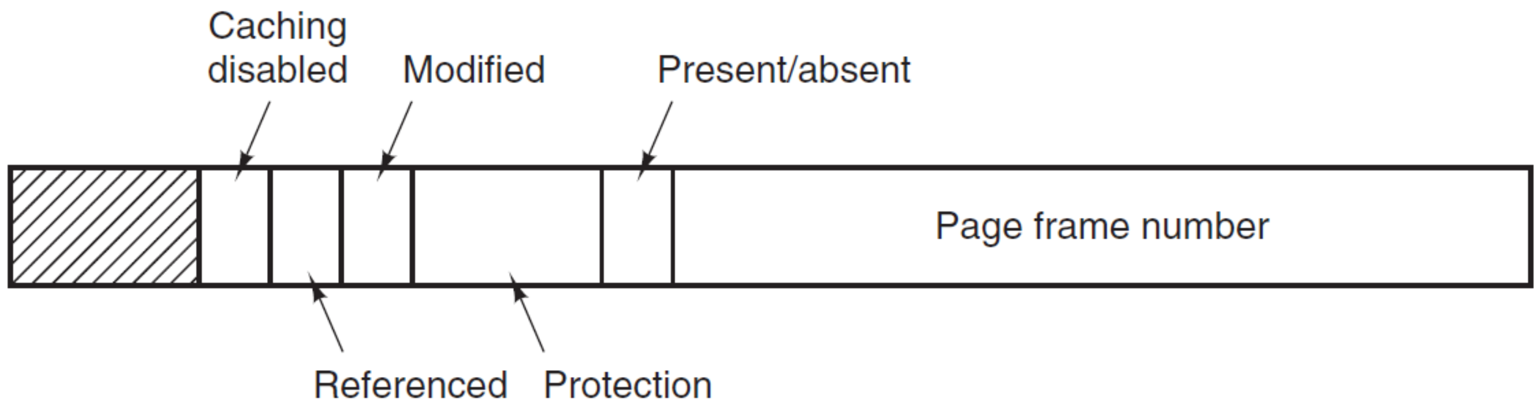
Page address mapping

- E.g. virtual address 8196 (0010000000000100 in binary), being mapped using the MMU
- The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset.
- With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.
- **Present/absent bit** keeps track of which pages are physically present in memory.
- The high-order 3 bits of the output register, along with the 12-bit offset, form a 15-bit physical address.



Page Table Entry (PTE)

- 32 bits (common size)
- *Page frame number*
- *Present/absent* – which pages are physically present in memory
- *Protection* – what kinds of access are permitted (read/write/exec)
- *Modified* – whether or not the page has been written
- *Reference* – whether the page has been accessed
- *Caching disabled* – allows caching to be disabled for the page



Paging Advantages

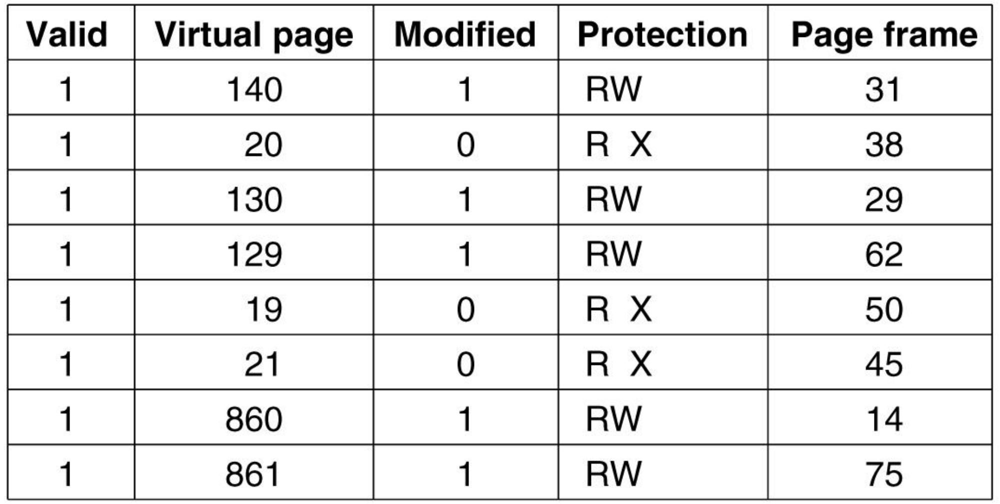
- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
- Easy to swap out chunks of a program
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size

Paging Limitations

- Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
    - Even more for two-level page tables!
  - Solution – use a hardware cache of lookups
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
    - Remember: each process has its own page table!
  - Solution – 2-level page tables

TLB (Translation Lookaside Buffer)

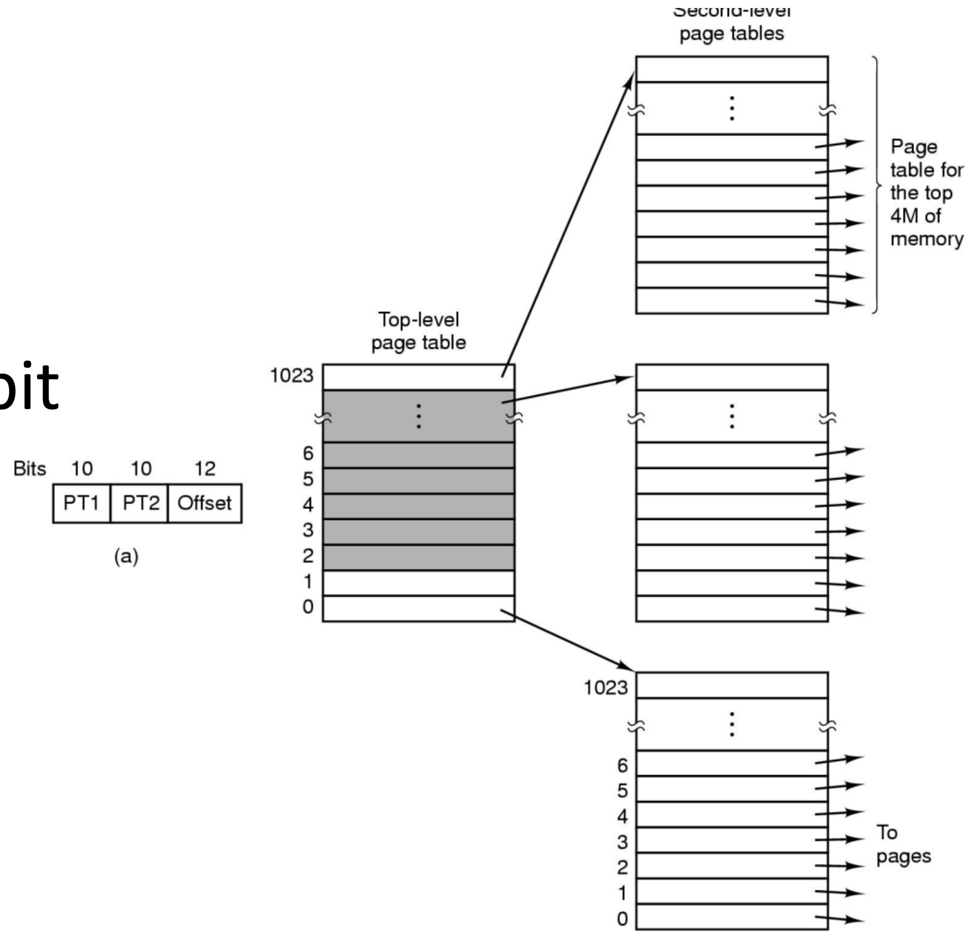
- A small hardware device for mapping virtual addresses to physical addresses without going through the page table.
- When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB.
- When the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup.



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

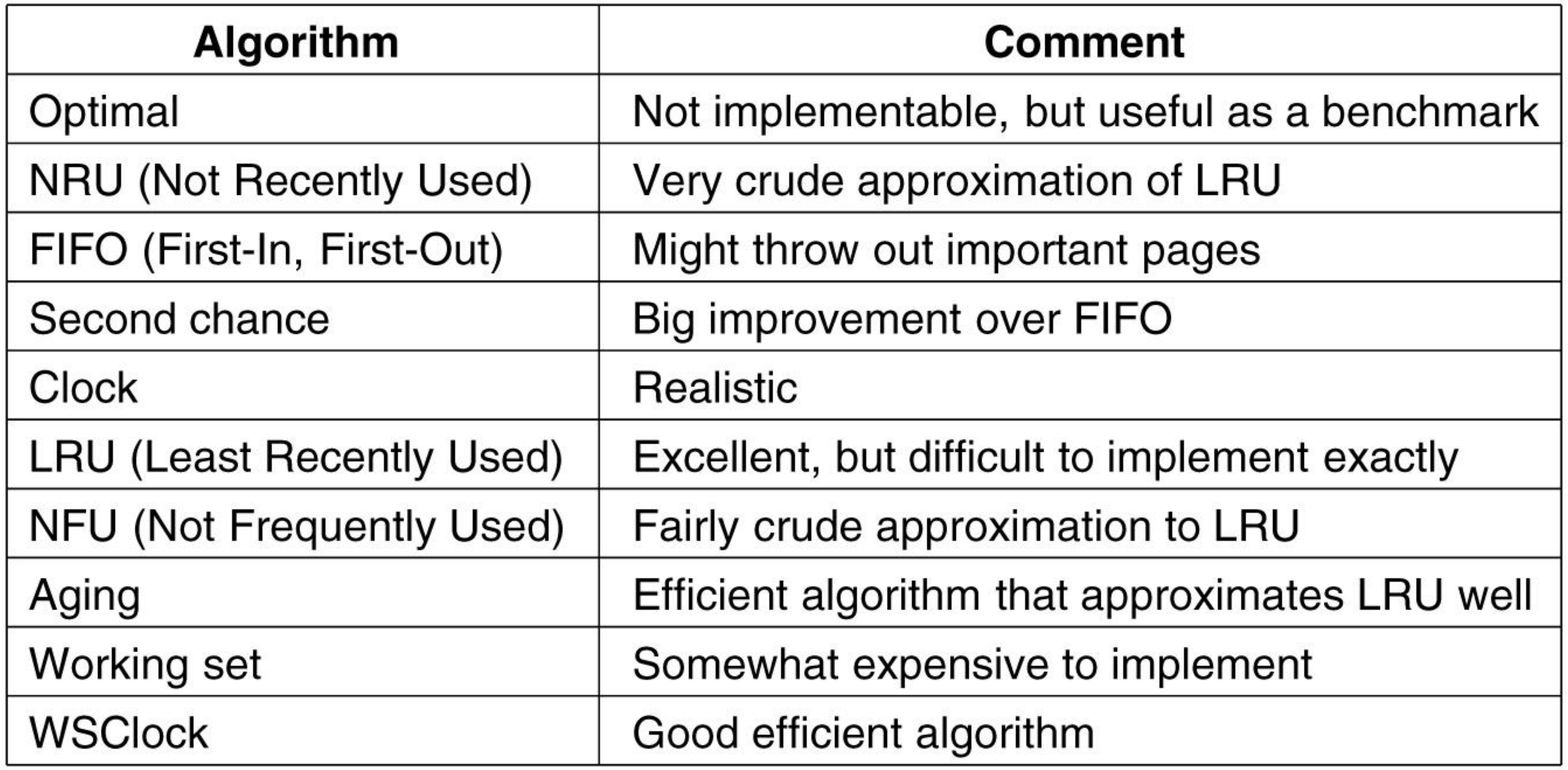
Multilevel Page Tables

- Single level page table size is too large
  - E.g. 4KB page, 32 bit virtual address, 1M entries per page table!
- A 32-bit virtual address is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field.
- Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.



Page Replacement Algorithms

- When a page fault occurs, the operating system has to choose a page to evict (remove from memory) to make room for the incoming page.



Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Page Replacement Algorithms

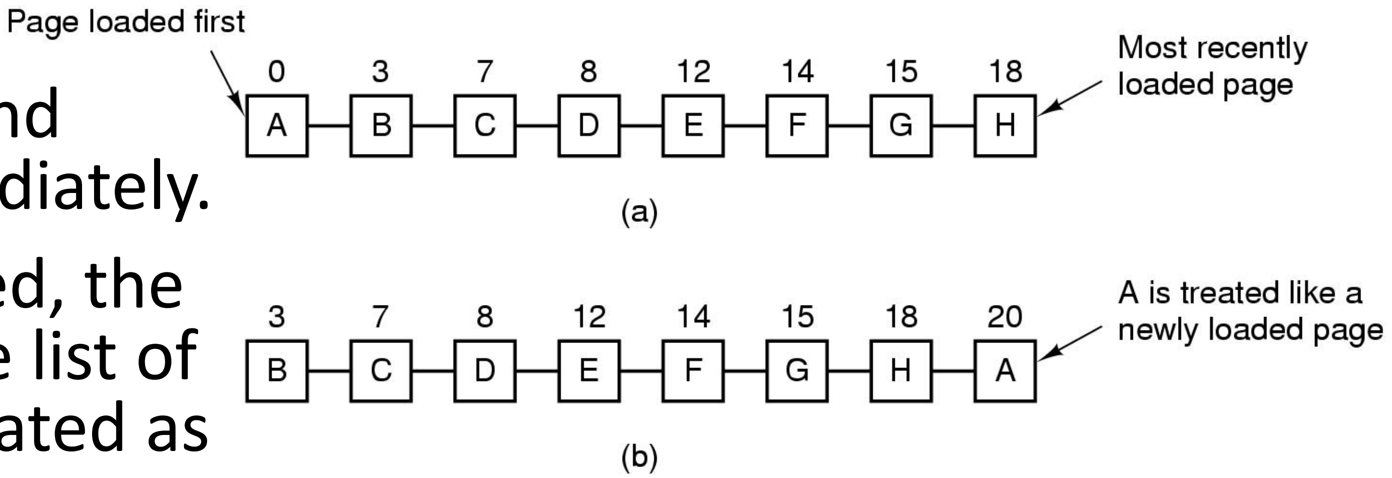
- The **optimal** algorithm evicts the page that will be referenced furthest in the future. Unfortunately, there is no way to determine which page this is, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.
- The **NRU (not recently used)** algorithm divides pages into four classes depending on the state of the *R (reference)* and *M (modify)* bits. A random page from the lowest-numbered class is chosen. This algorithm is easy to implement, but it is very crude.
- **FIFO (first in first out)** keeps track of the order in which pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.
- **Second chance** is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance.

Page Replacement Algorithms

- **Clock** is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.
- **LRU** is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used.
- **NFU** is a crude attempt to approximate LRU. It is not very good.
- **Aging** is a much better approximation to LRU and can be implemented efficiently.
- The **working set** algorithm gives reasonable performance, but it is somewhat expensive to implement.
- **WSClock** is a variant that not only gives good performance but is also efficient to implement.

Second Chance Algorithm

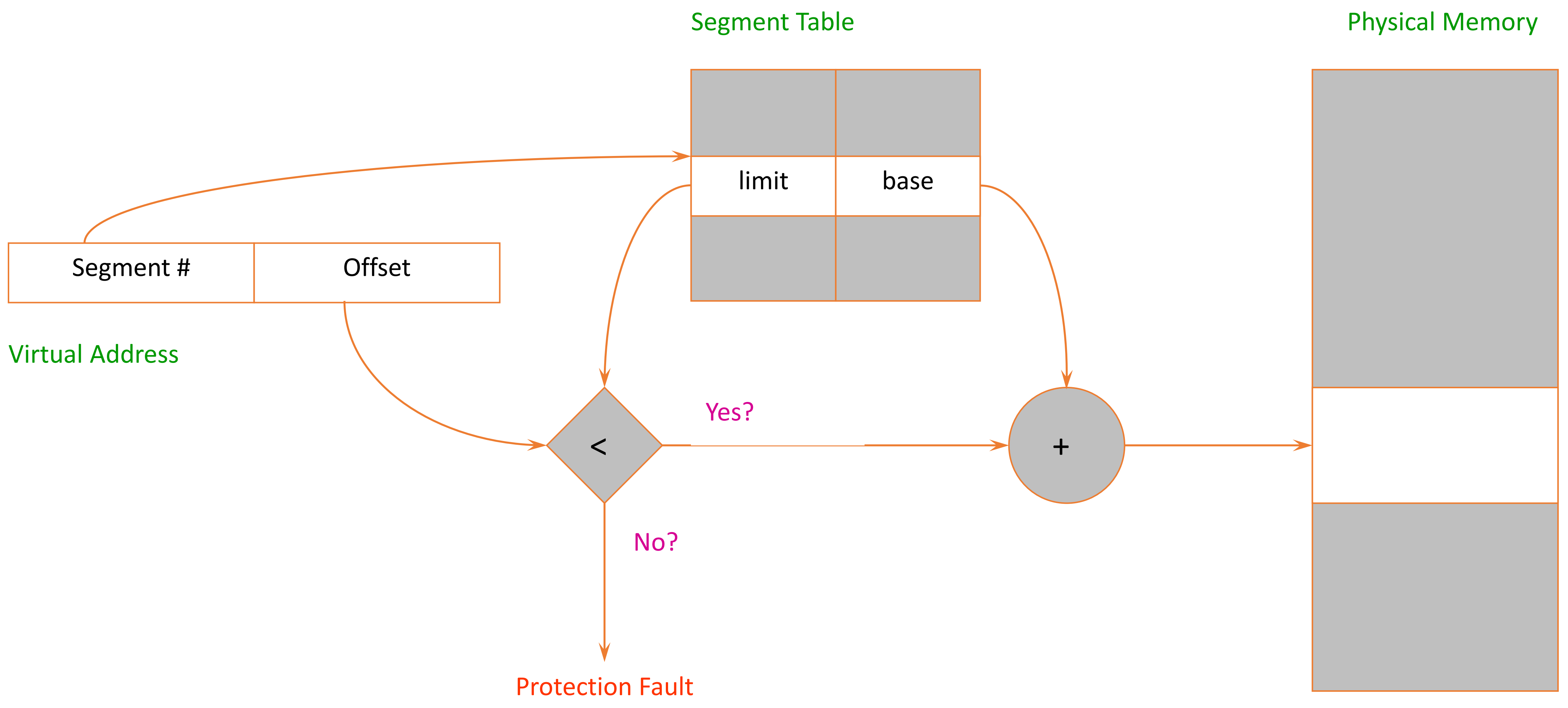
- A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page.
- If it is 0, the page is both old and unused, so it is replaced immediately.
- If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory.
- Then the search continues.



Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - Units of memory from user's perspective
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table

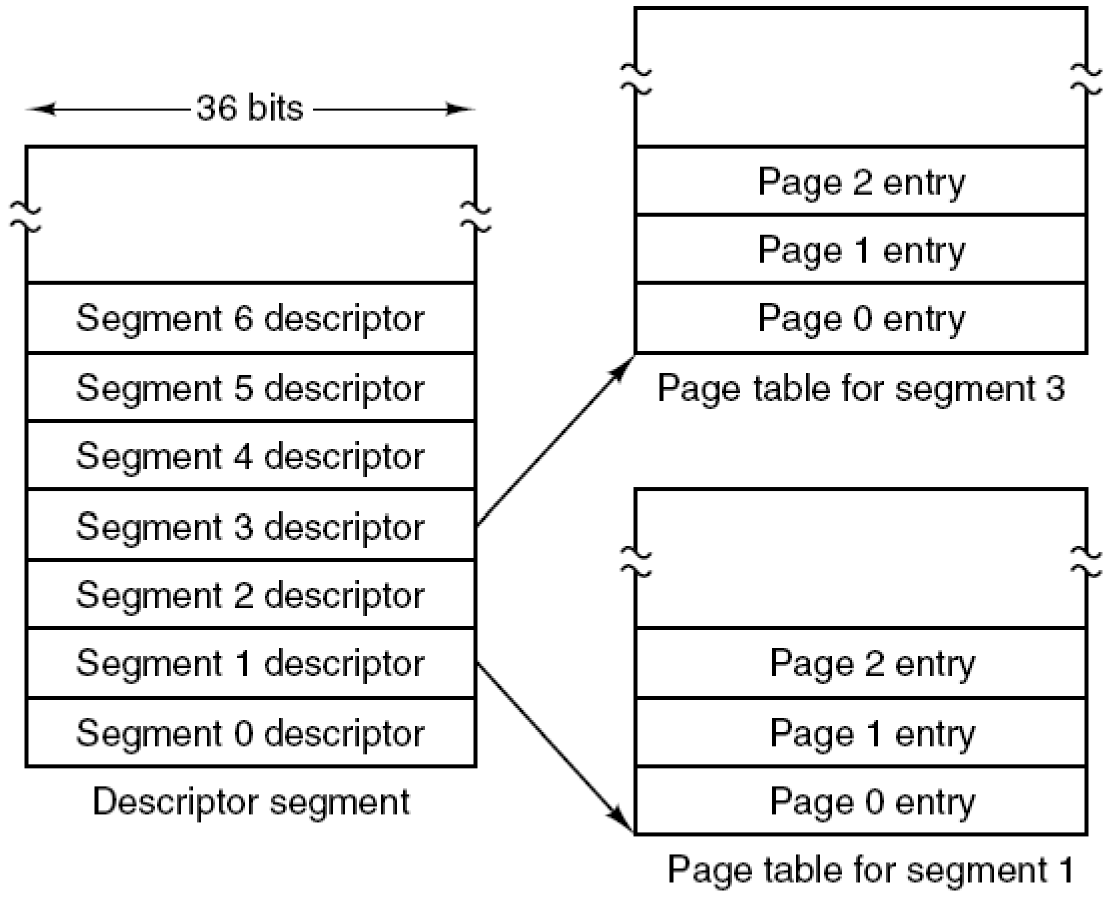
Segmentation



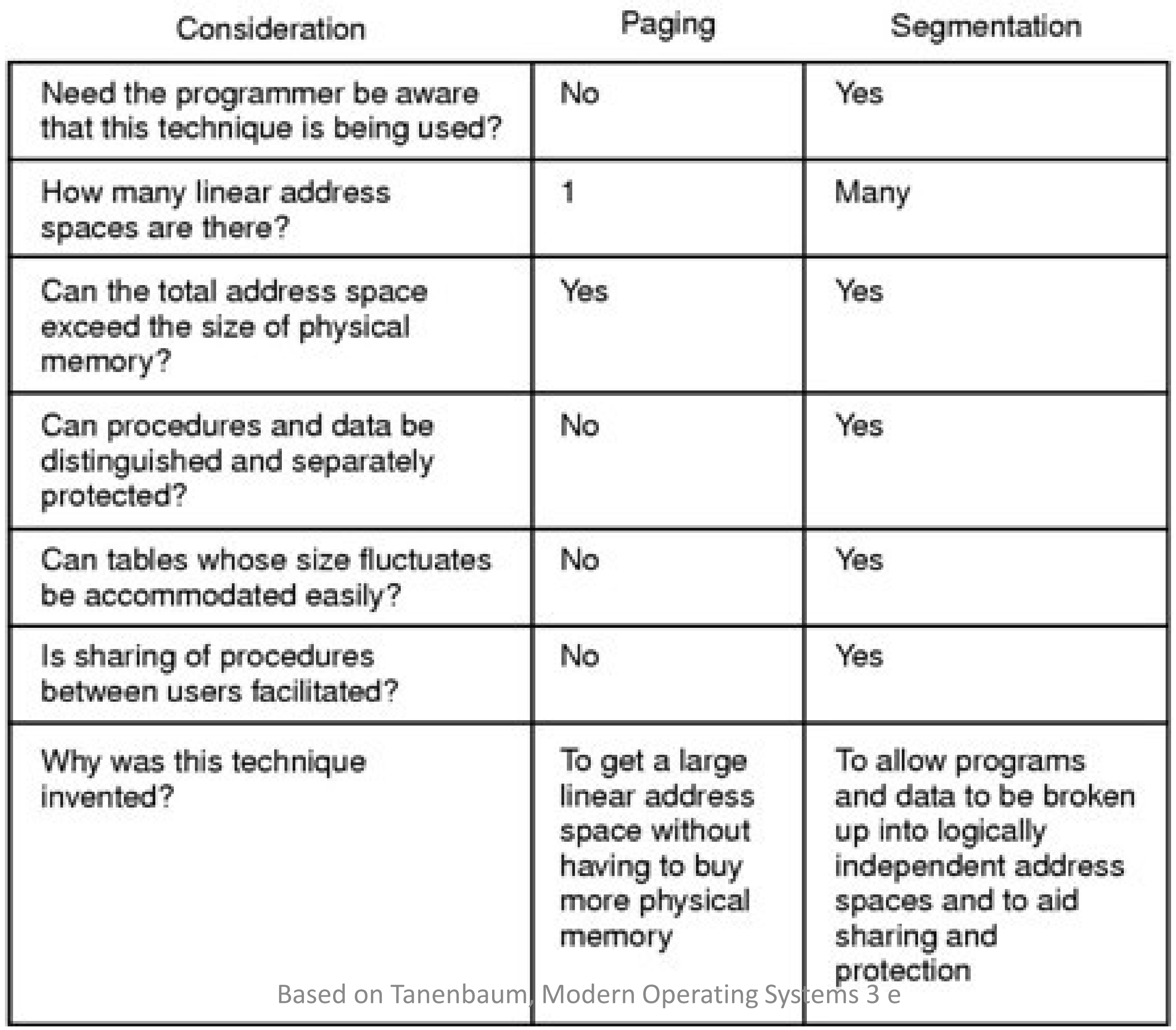
Segmentation and Paging

- Can combine segmentation and paging
  - The x86 supports segments and paging
- Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed size chunks
  - Makes segments easier to manage within physical memory
  - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex…

Segmentation and Paging



Segmentation and Paging



Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Summary

- Virtual memory
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses
- Various techniques
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user's perspective
  - Combine paging and segmentation to get benefits of both