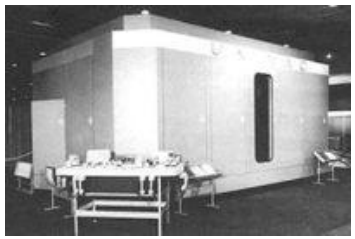
Memory management: outline

- Concepts
- Swapping
- Paging
  - Multi-level paging
  - TLB & inverted page tables

Memory size/requirements are growing…



1951: the UNIVAC computer:
1000 72-bit words!

1971: the Cray 1 supercomputer:
About 200K memory gates!

1983: the IBM XT: 640KB
"should be enough for everybody…"

2012: today's laptops: 4GB-8GB

Our requirements from memory

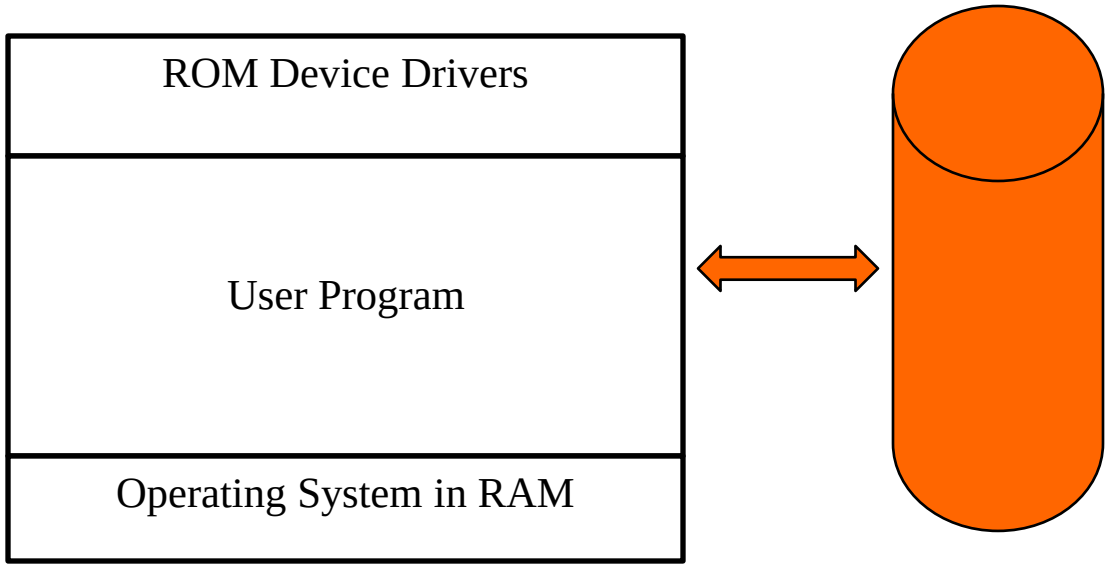
- An indispensible resource
- Variation on Parkinson's law: "Programs expand to fill the memory available to hold them"
- Ideally programmers want memory that is
  - fast
  - non volatile
  - large
  - cheap

The Memory Hierarchy

- Memory hierarchy
  - Hardware registers: very small amount of very fast volatile memory
  - Cache: small amount of fast, expensive, volatile memory
  - Main memory: medium amount of medium-speed, medium price, volatile memory
  - Disk: large amount of slow, cheap, non-volatile memory
- The memory manager is the part of the OS that handles main memory and transfers between it and secondary storage (disk)

Mono-programming memory management

- Mono-programming systems require a simple memory manager
  - User types a command
  - System loads program to main memory and executes it
  - System displays prompt, waits for new command

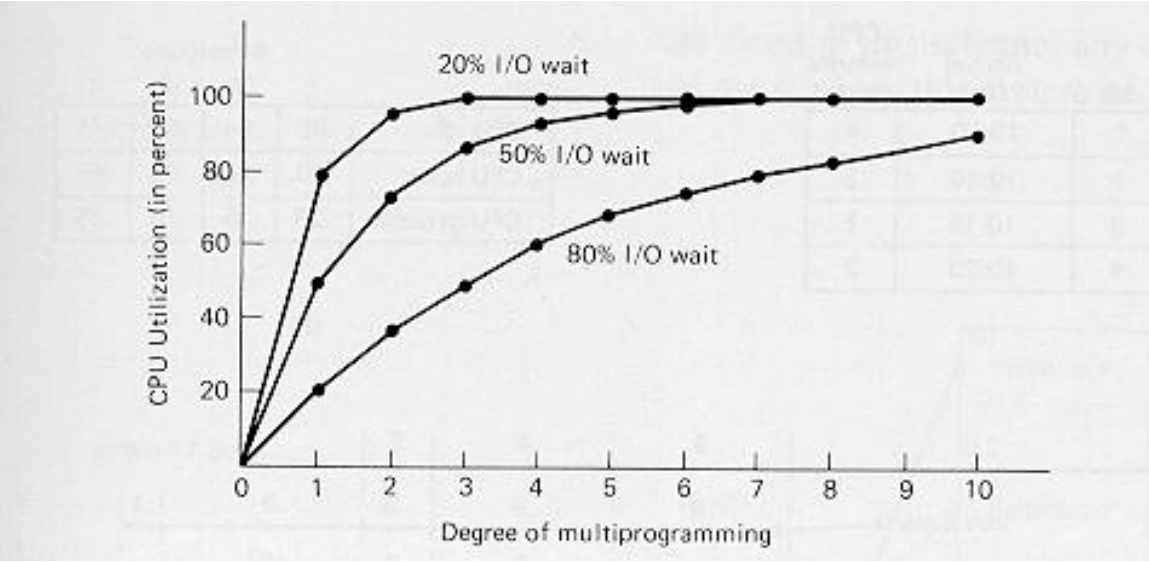


MS DOS memory organization

Multi-programming Motivation

n processes, each spending a fraction ***p*** of their time waiting for I/O, gives a probability p^n of all processes waiting for I/O simultaneously

CPU utilization = 1 - p^n



This calculation is simplistic

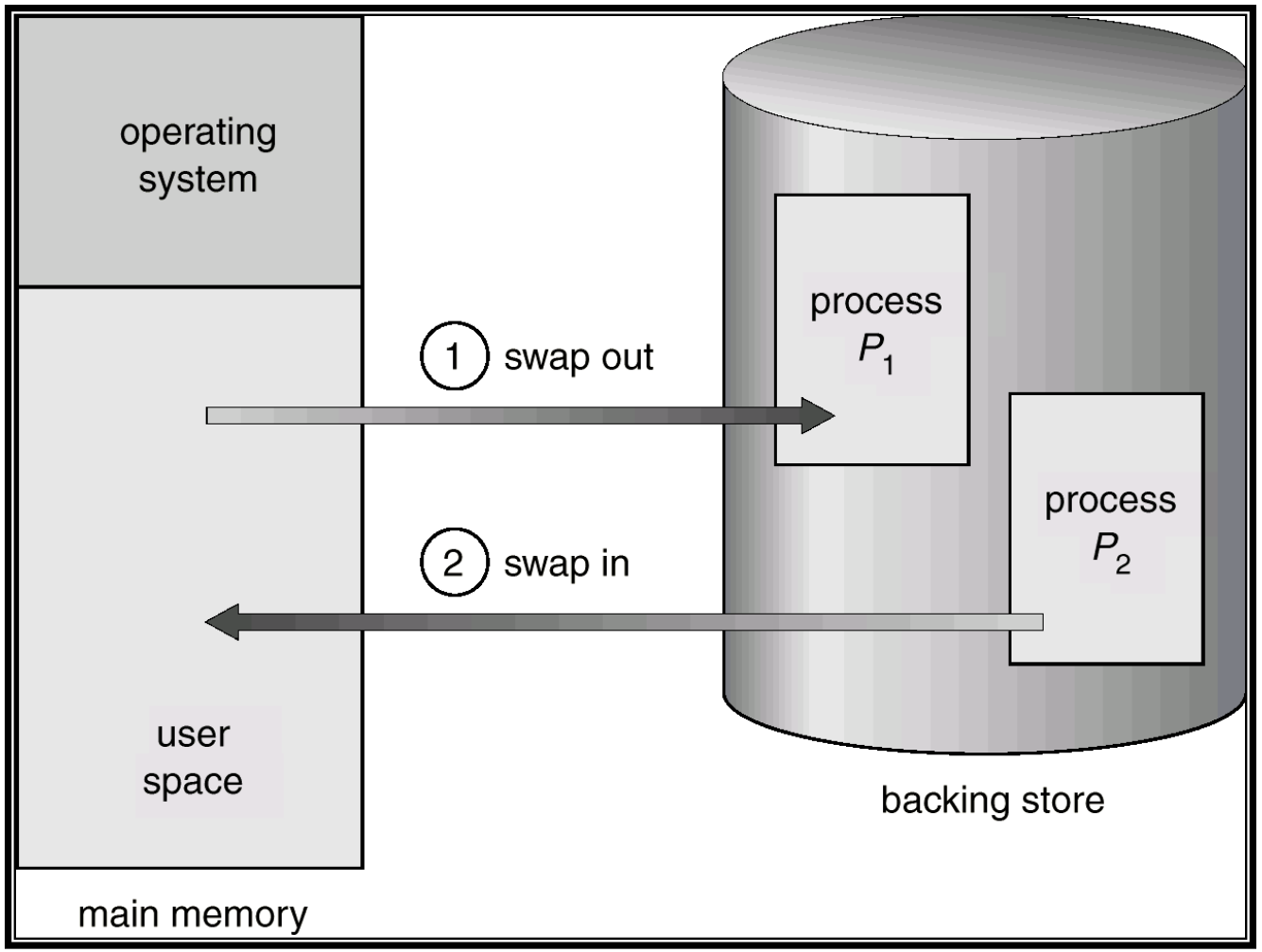
Memory/efficiency tradeoff

- Assume each process takes 200k and so does the operating system
- Assume there is 1Mb of memory available and that p=0.8
- space for 4 processes $\Rightarrow$ 60% cpu utilization
- Another 1Mb enables 9 processes
  $\Rightarrow$ 87% cpu utilization

Memory management: outline

- Concepts
- Swapping
- Paging
  - Multi-level paging
  - TLB & inverted page tables

Swapping: schematic view



Swapping

- Bring a process in its entirety, run it, and then write back to backing store (if required)
- Backing store – fast disk large enough to accommodate copies of all memory images for all processes; must provide direct access to these memory images.
- Major part of swap time is transfer time; total transfer time is proportional to the *amount* of memory swapped. This time can be used to run another process
- Creates holes in memory (fragmentation), memory compaction may be required
- No need to allocate swap space for memory-resident processes (e.g. Daemons)
- Not used much anymore (but still interesting…)

Multiprogramming with Fixed Partitions (OS/360 MFT)

- *How to organize main memory?*
- *How to assign processes to partitions?*
- *Separate queues **vs.** single queue*

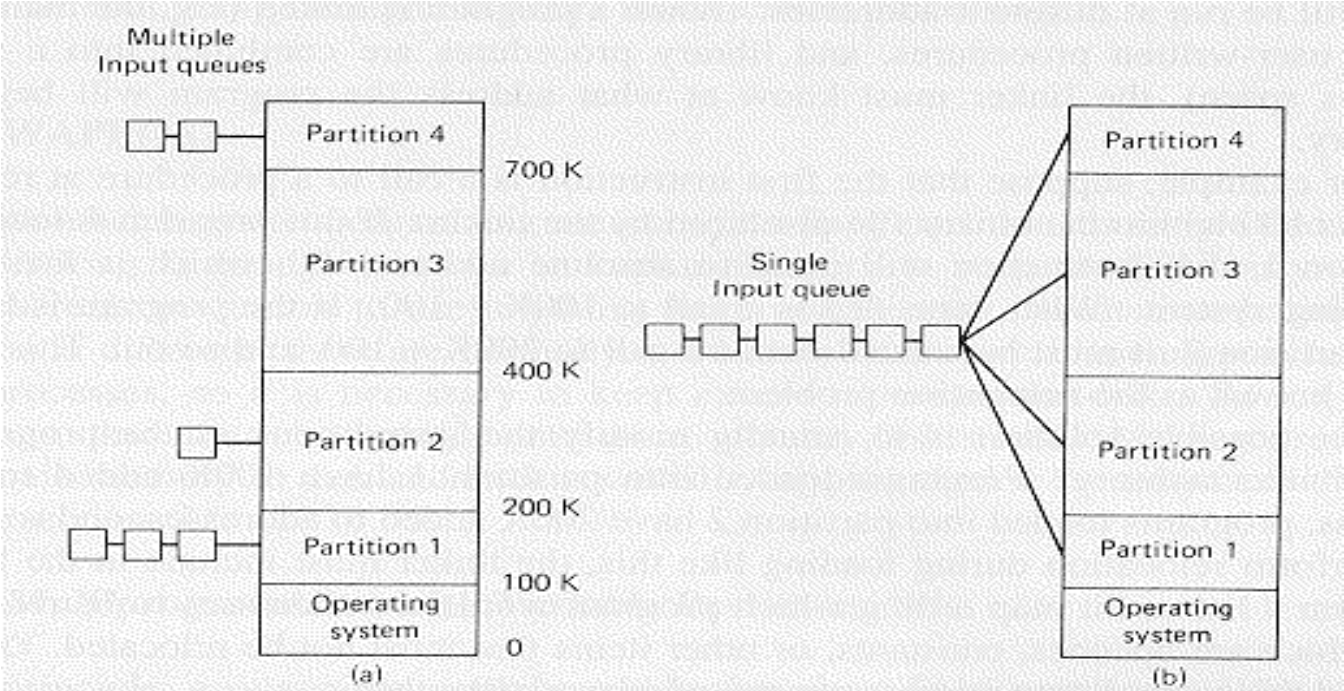


Fig. 3-4. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

Allocating memory - growing segments

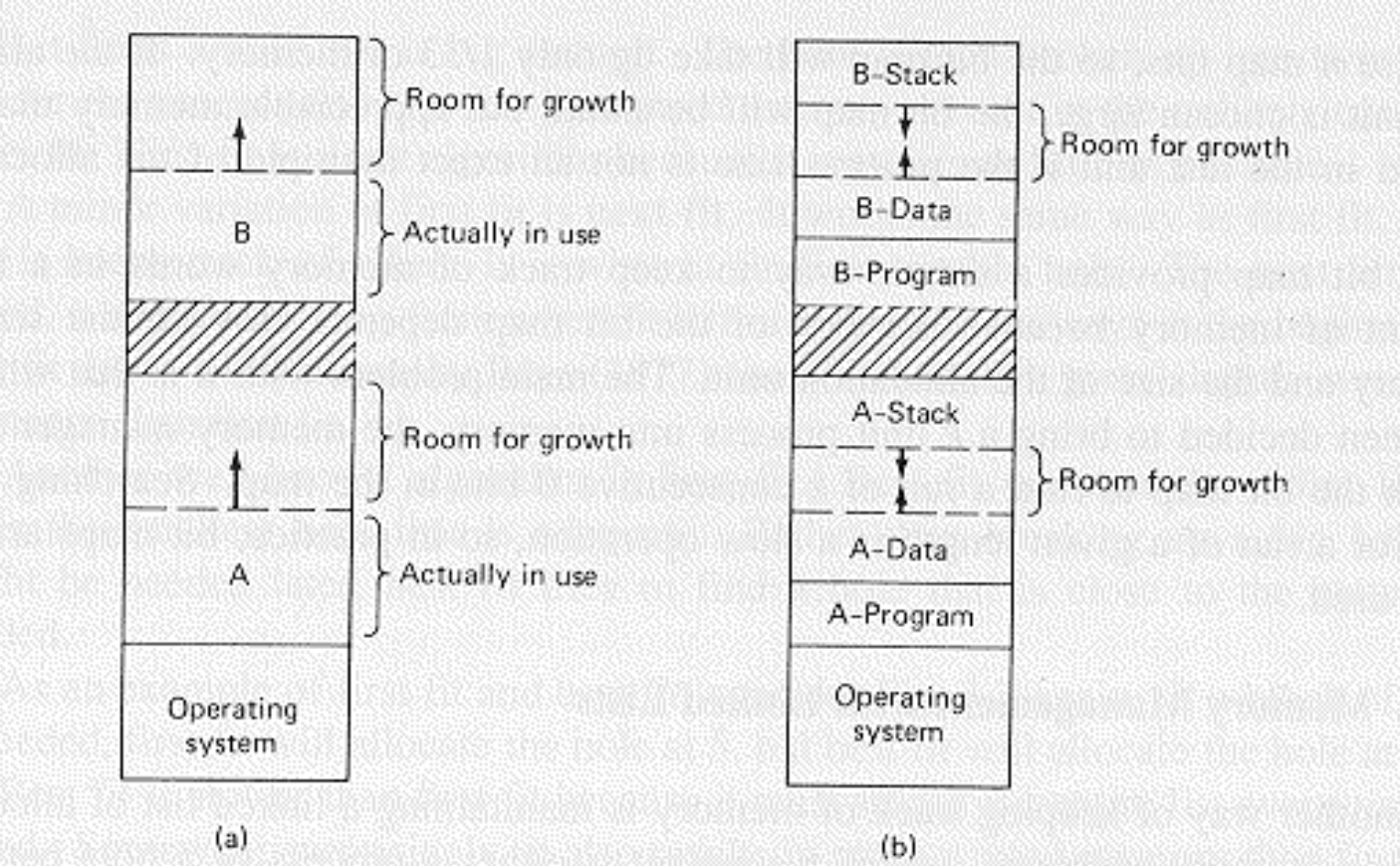


Fig. 3-6. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Memory Allocation - Keeping Track (bitmaps; linked lists)

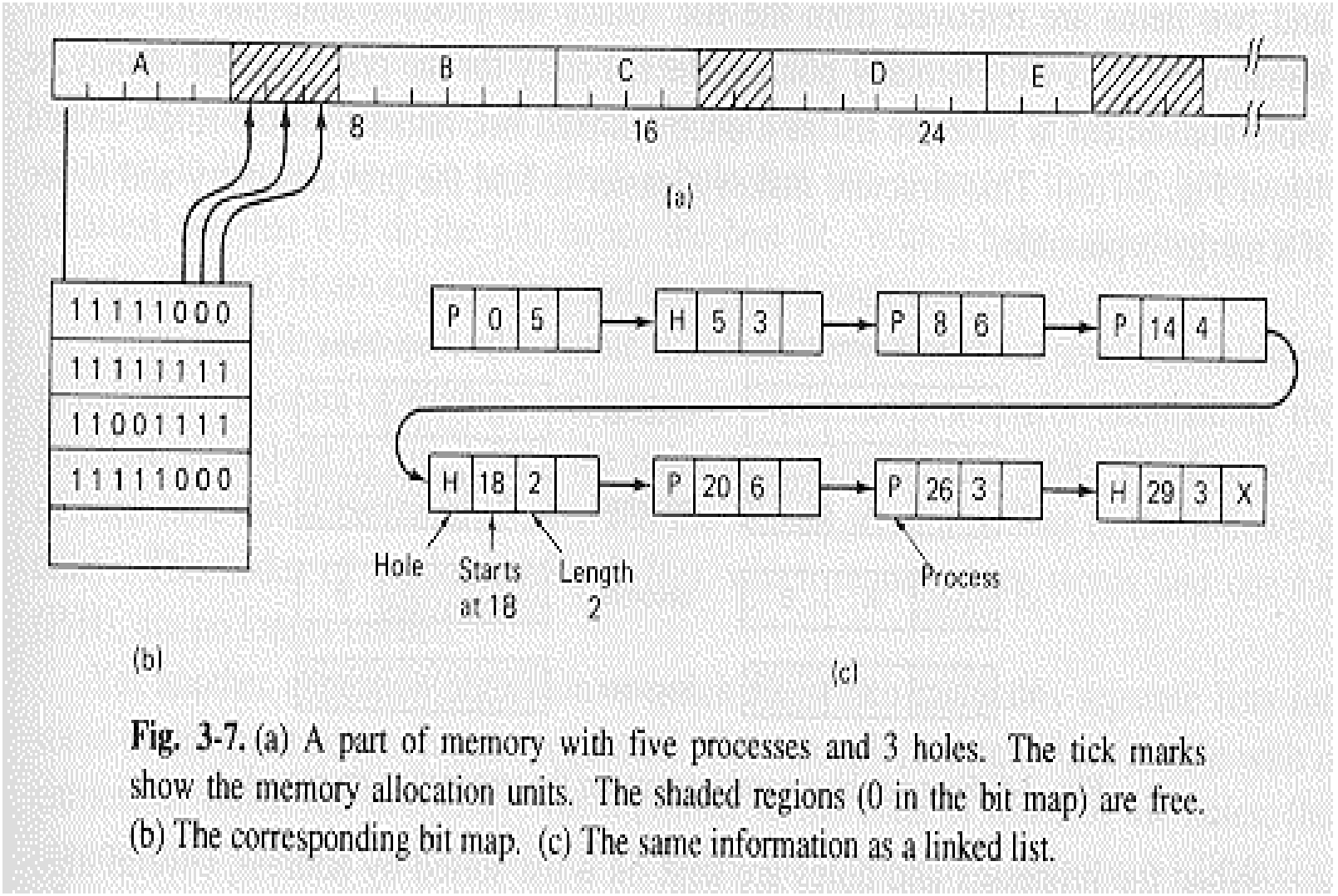


Fig. 3-7. (a) A part of memory with five processes and 3 holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a linked list.

Swapping in Unix (prior to 3BSD)

- When is swapping done?
  - Kernel runs out of memory
  - a fork system call – no space for child process
  - a brk system call to expand a data segment
  - a stack becomes too large
  - A swapped-out process becomes ready
- Who is swapped?
  - a suspended process with “highest” priority (in)
  - a process which consumed much CPU (out)
- How much space is swapped? use holes and first-fit (more on this later)

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes (e.g., MS/DOS .com programs)
- **Load time**: Must generate *relocatable* code if memory location is not known at compile time
- **Execution time**: Binding delayed until run-time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers or virtual memory support*)

Which of these binding-types dictates that a process be swapped back from disk to same location?

Dynamic Linking

- Linking postponed until execution time
- A small piece of code, *stub*, is used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and calls the routine
- Operating system makes sure the routine is mapped to processes' memory address
- Dynamic linking is particularly useful for libraries (e.g., Windows DLLs)

Do DLLs save space in main memory or in disk?

Strategies for Memory Allocation

- *First fit* – do not search too much..
- *Next fit* - start search from last location
- *Best fit* - a drawback: generates small holes
- *Worst fit* - solves the above problem, badly
- *Quick fit* - several queues of different sizes

Main problem of such memory allocation – Fragmentation

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **Reduce external fragmentation by compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

Memory Compaction

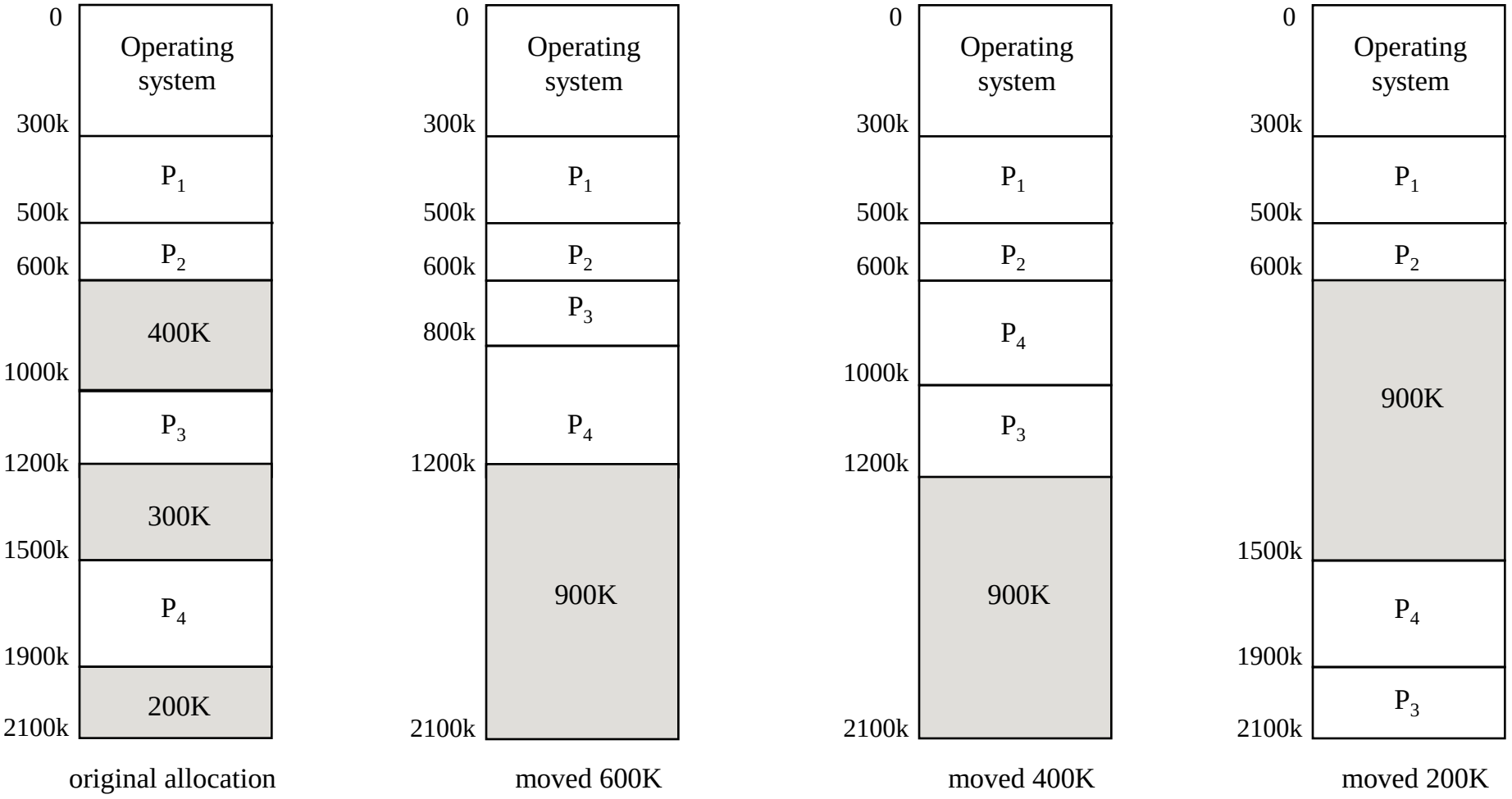


Figure 8.11 Comparison of some different ways to compact memory

The Buddy Algorithm

- An example scheme – the Buddy algorithm (Knuth 1973):
  - Separate lists of free holes of sizes of powers of two
  - For any request, pick the 1st large enough hole and halve it recursively
  - Relatively little external fragmentation (as compared with other simple algorithms)
  - Freed blocks can only be merged with neighbors of their own size. This is done recursively

The Buddy Algorithm

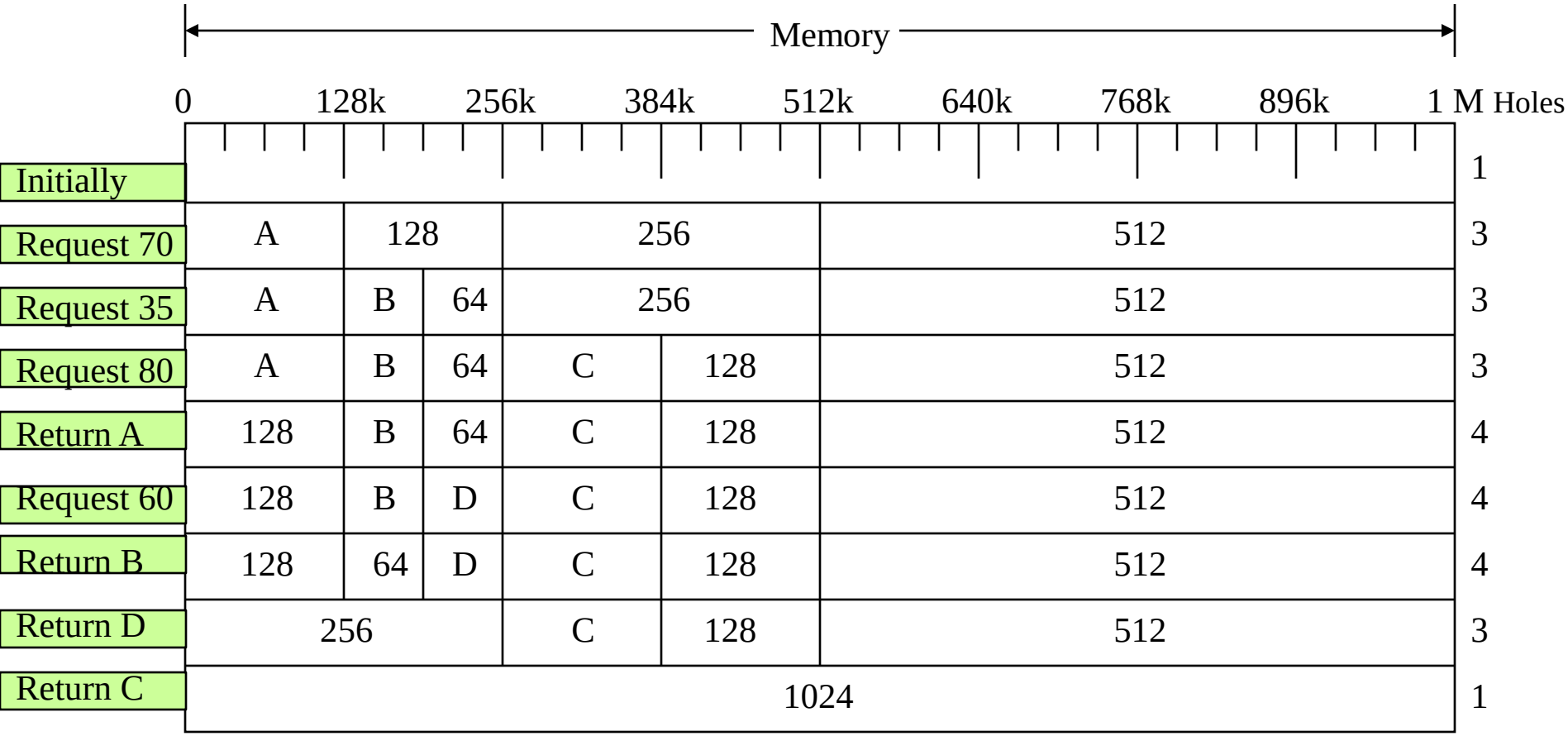


Fig. 3-9. The buddy algorithm. The horizontal axis represents memory addresses. The numbers are the sizes of unallocated blocks of memory in K. The letters represent allocated blocks of memory.

Logical vs. Physical Address Space

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to modern memory management
  - *Logical address* – generated by the CPU; also referred to as *virtual address*
  - *Physical address* – address seen by the memory unit

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding schemes

Memory management: outline

- Concepts
- Swapping
- Paging
  - Multi-level paging
  - TLB & inverted page tables

Paging and Virtual Memory

- Support an address space that is independent of physical memory
- Only part of a program may be in memory: program size may be larger than physical memory
- 2^{32} addresses for a 32 bit (address bus) machine - *virtual addresses*
- can be achieved by segmenting the executable (using segment registers), or by dividing memory using another method
- *Paging* - Divide physical memory into *fixed-size* blocks (*page-frames*)
- Allocate to processes *non-contiguous* memory chunks – *disregarding holes*

Memory Management Unit

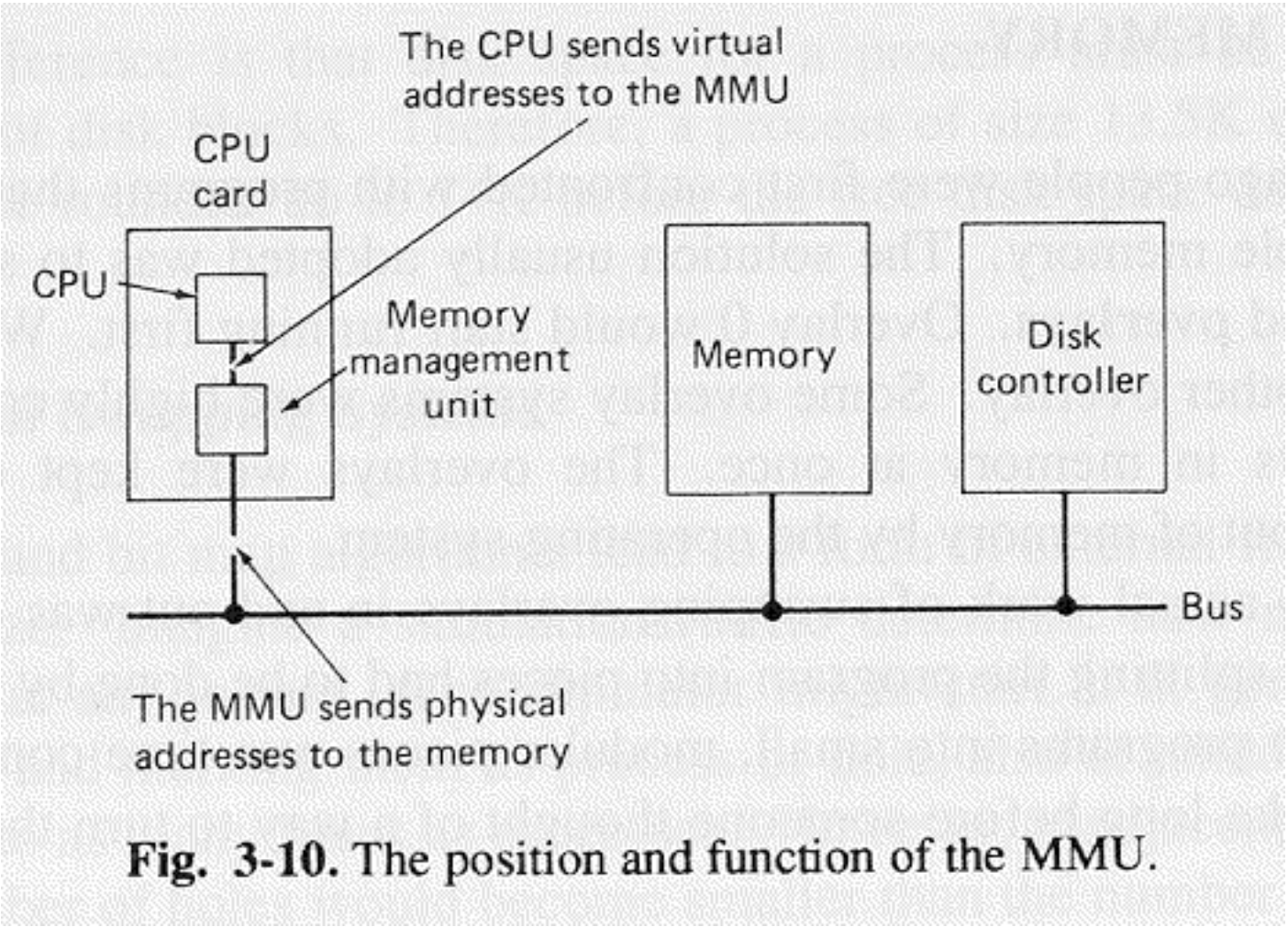


Fig. 3-10. The position and function of the MMU.

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical addresses (among other things). Typically part of CPU
- The MMU translates the virtual address generated by a user process before it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

```
mov A, 1000   # virtual (logical) address
is sent to the MMU – which maps this to:
mov A, 9192   # physical (real) address
```

Paging

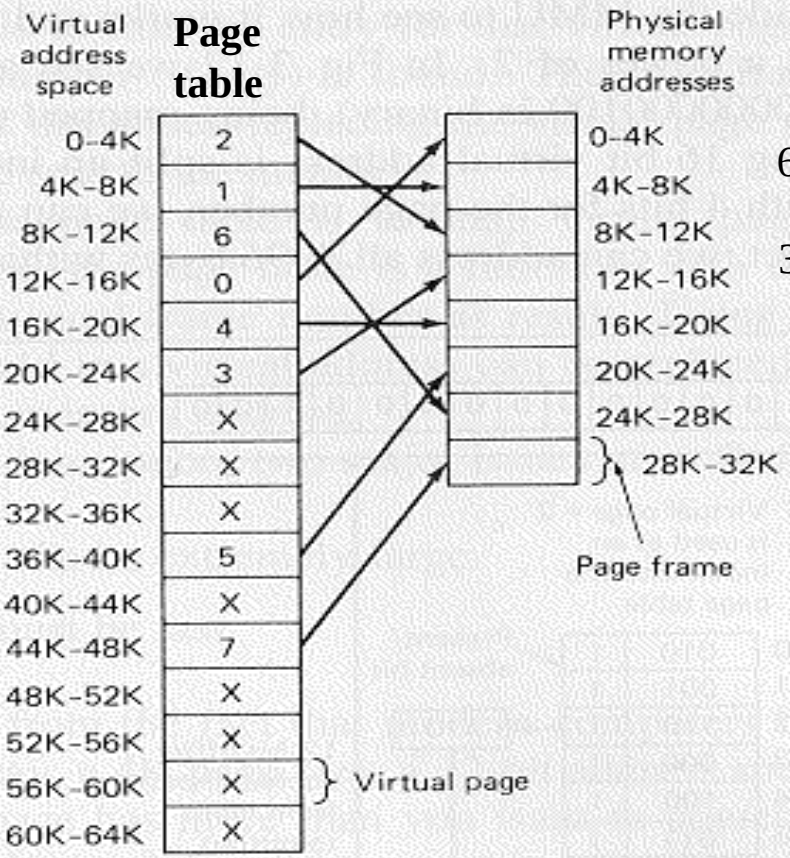


Fig. 3-11. The relation between virtual addresses and physical memory addresses is given by the page table.

64K: virtual space

32K: physical space

Must choose the size of the pages: determines size of the page table

Operation of the MMU

Map incoming virtual address to physical address.

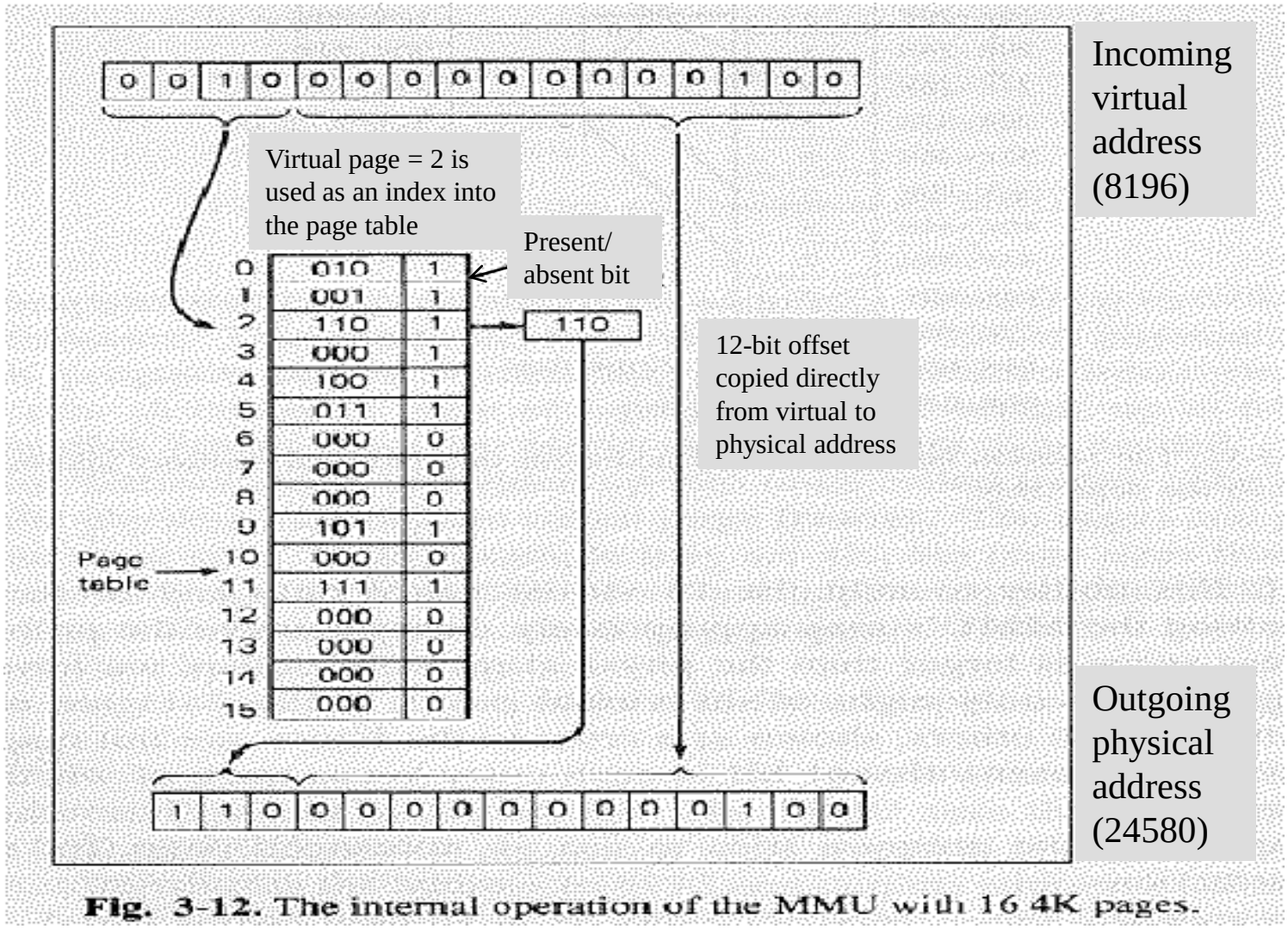


Fig. 3-12. The internal operation of the MMU with 16 4K pages.

Page Faults: Pages In/Out

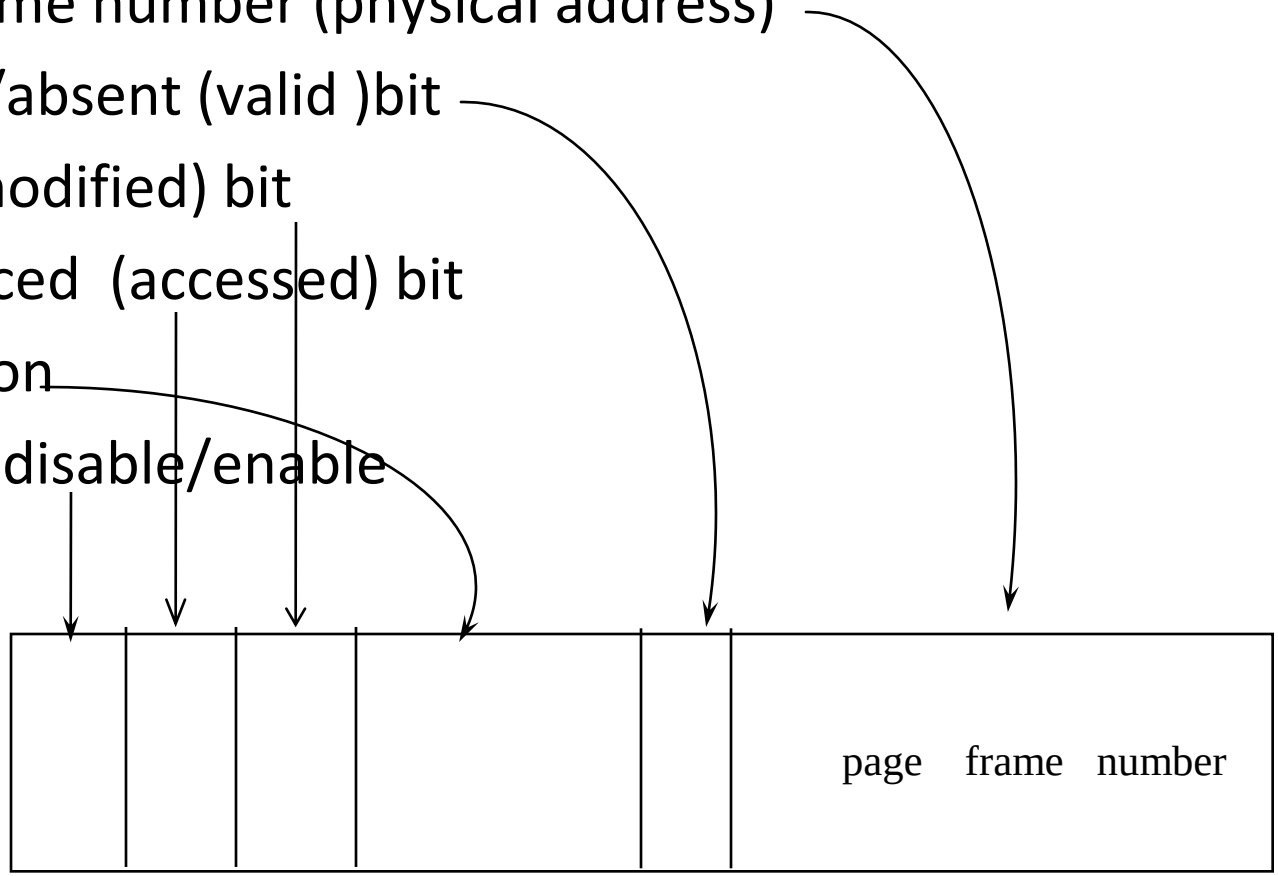
- When MMU tries to access a page which is not currently loaded in RAM – a hardware trap is raised: PAGE FAULT.
- This allows the kernel to handle the swapping of one currently loaded page with the requested page.
- The kernel maintains copies of the pages in disk. Up to full process virtual memory for each process in swap disk.
- Issues:
  - Select page to be evicted (Eviction strategy)
  - Should evicted page be written to disk to save modifications?

Pages: blocks of virtual addresses

Page frames: refer to physical memory segments

Page Table Entries (PTE) contain (per page):

- Page frame number (physical address)
- Present/absent (valid)bit
- Dirty (modified) bit
- Referenced (accessed) bit
- Protection
- *Caching* disable/enable



Page size vs. Page-table size Tradeoffs

- A logical address of 32-bits (4GB) can be divided into:
  - 1K pages and 4M entries table
  - 4K pages and 1M entries table
- Large pages – a smaller number of pages, but higher internal fragmentation
- Smaller pages – larger tables (also waste of space)

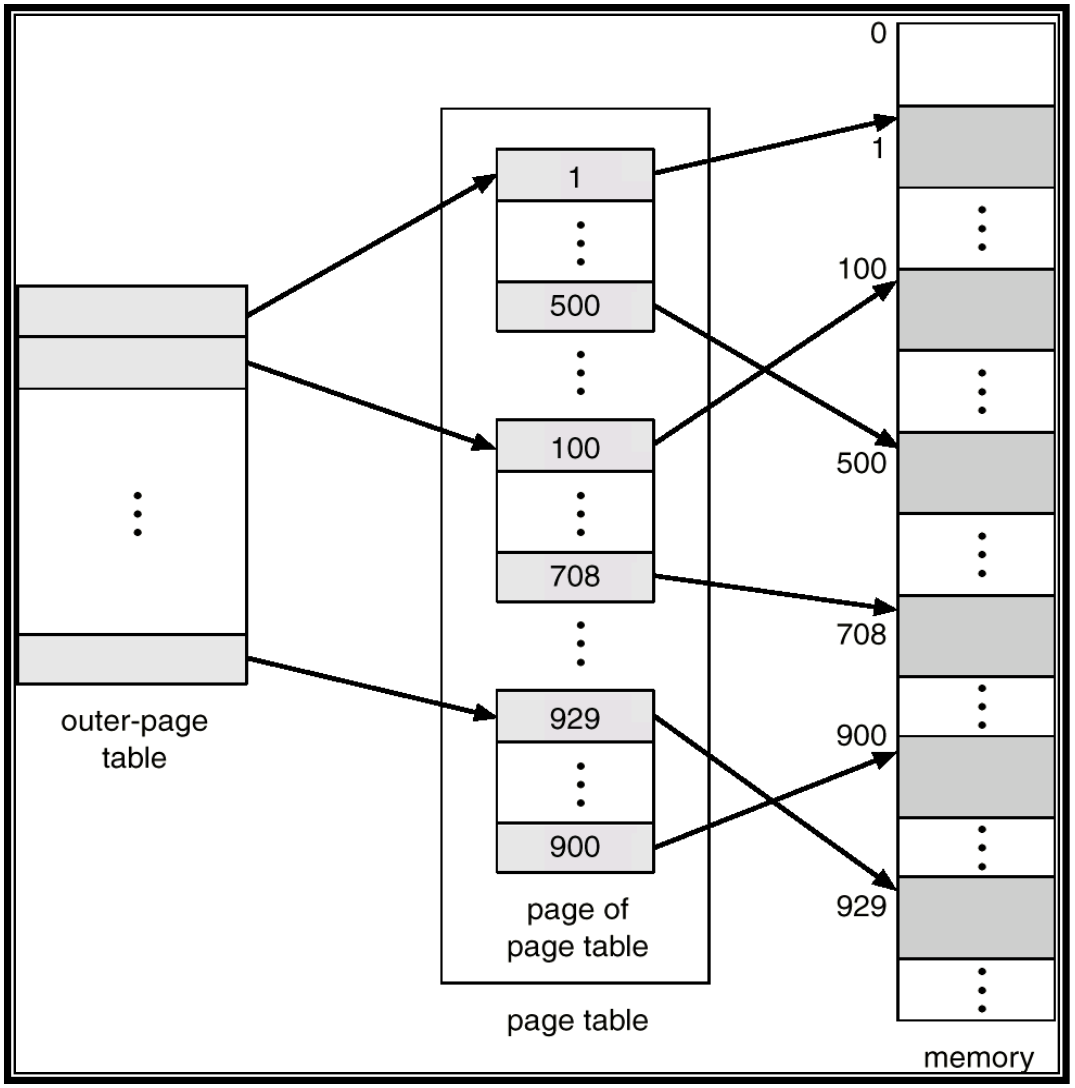
Large tables, and we need ONE PER PROCESS!

Page table considerations

- Can be very large (1M pages for 32 bits, 4K page size)
- Must be fast (every instruction needs it)
- One extreme will have it all in hardware - fast registers that hold the page table and are loaded with each process - too expensive for the above size
- The other extreme has it all in main memory (using a page table base register – *ptbr* - to point to it) - each memory reference during instruction translation is doubled...
- Possible solution: to avoid keeping complete page tables in memory - make them multilevel, and avoid making multiple memory references per instruction by caching

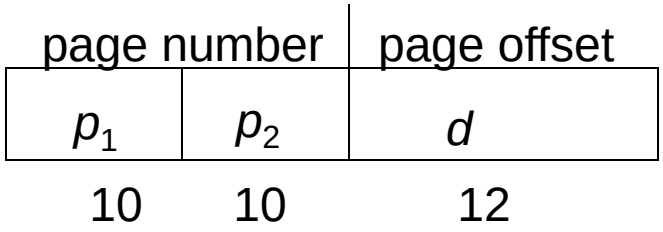
We do paging on the page-table itself!

Two-Level Page-Table Scheme



Two-Level Paging 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.
- Since the page table itself is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address has the following structure:

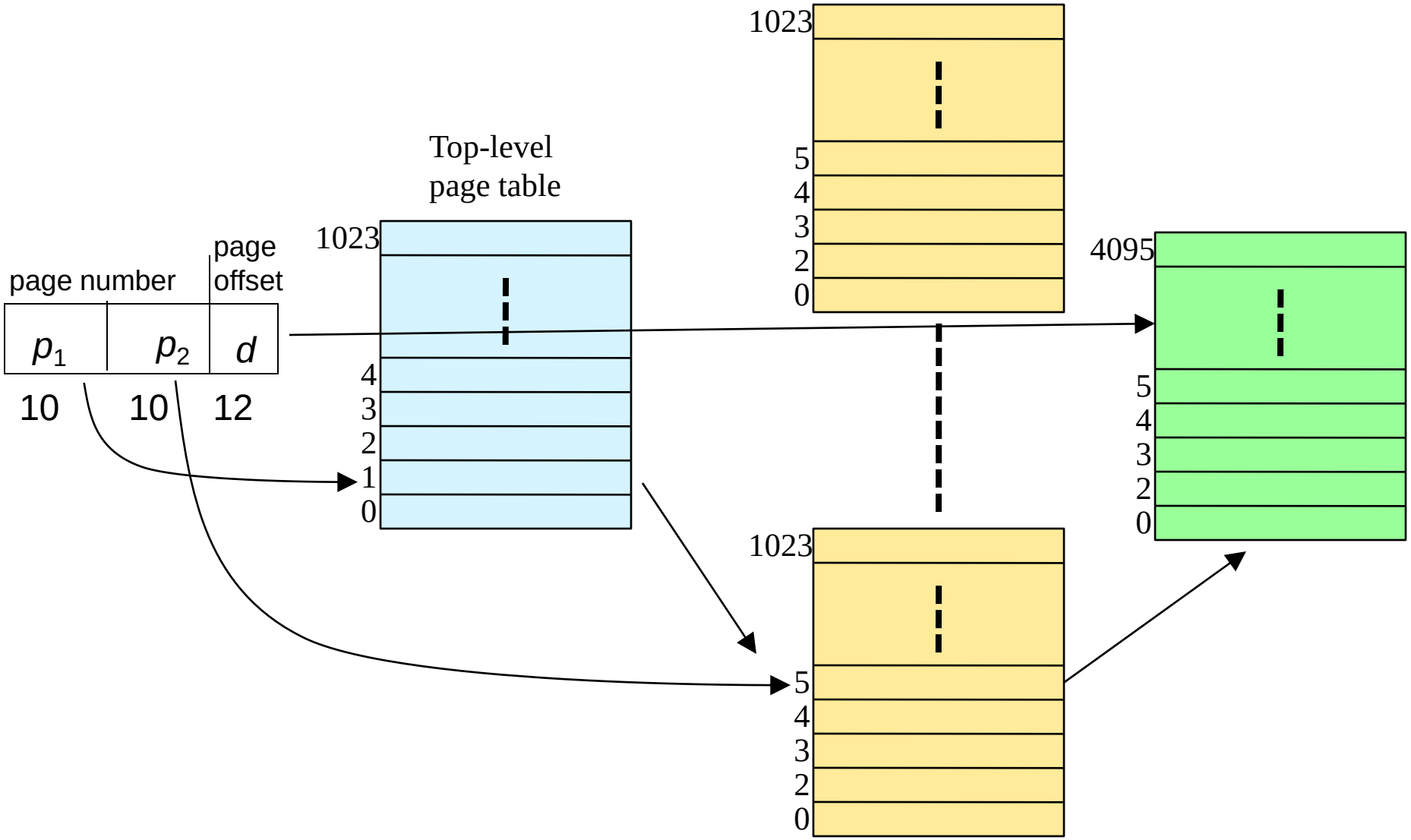


Where p_1 is an index into the top-level (outer) page table, and p_2 is an index into the selected second-level page table

Two-Level Paging: Motivation

- Two-level paging helps because most of the time a process does not need ALL of its virtual memory space.
- Example: A process in a 32bit machine uses
  - 4MB of stack
  - 4MB of code segment
  - 4MB of heap
- Only 12MB effectively used out of 4GB – only 3 pages of pages needed (out of 1024)

Two-Level Paging Example (cont'd)



Translation Lookaside Buffer (TLB):
Associative memory for minimizing redundant memory accesses

- TLB resides in MMU
- Most accesses are to a small set of pages ➔ high hit rate

  Locality of reference

Valid entry	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

Fig. 3-20. An associative memory to speed up paging.

Notes about TLB

- TLB is an associative memory
- Typically, inside the MMU
- With a large enough *hit-ratio,* the extra accesses to page tables are rare
- Only a complete virtual address (all levels) can be counted as a hit
- with multi-processing, TLB must be cleared on context switch - wasteful..
  - Possible solution: add a field to the associative memory to hold process ID and change in context switch.
- TLB management may be done by hardware or OS

Inverted page tables

- Regular page tables impractical for 64-bit address space
  4K page size / 2^{52} pages x 8 bytes → 30M GB page tables!
- *Inverted page table* – sorted by (physical) page frames and not by virtual pages
  1 GB of RAM & 4K page size / 256K entries → 2 MB table
- A single inverted page table used for all processes currently in memory
- Each entry stores which process/virtual-page maps to it
- A hash table is used to avoid linear search for every virtual page
- In addition to the hash table, TLB registers are used to store recently used page table entries

Inverted Page Table Architecture

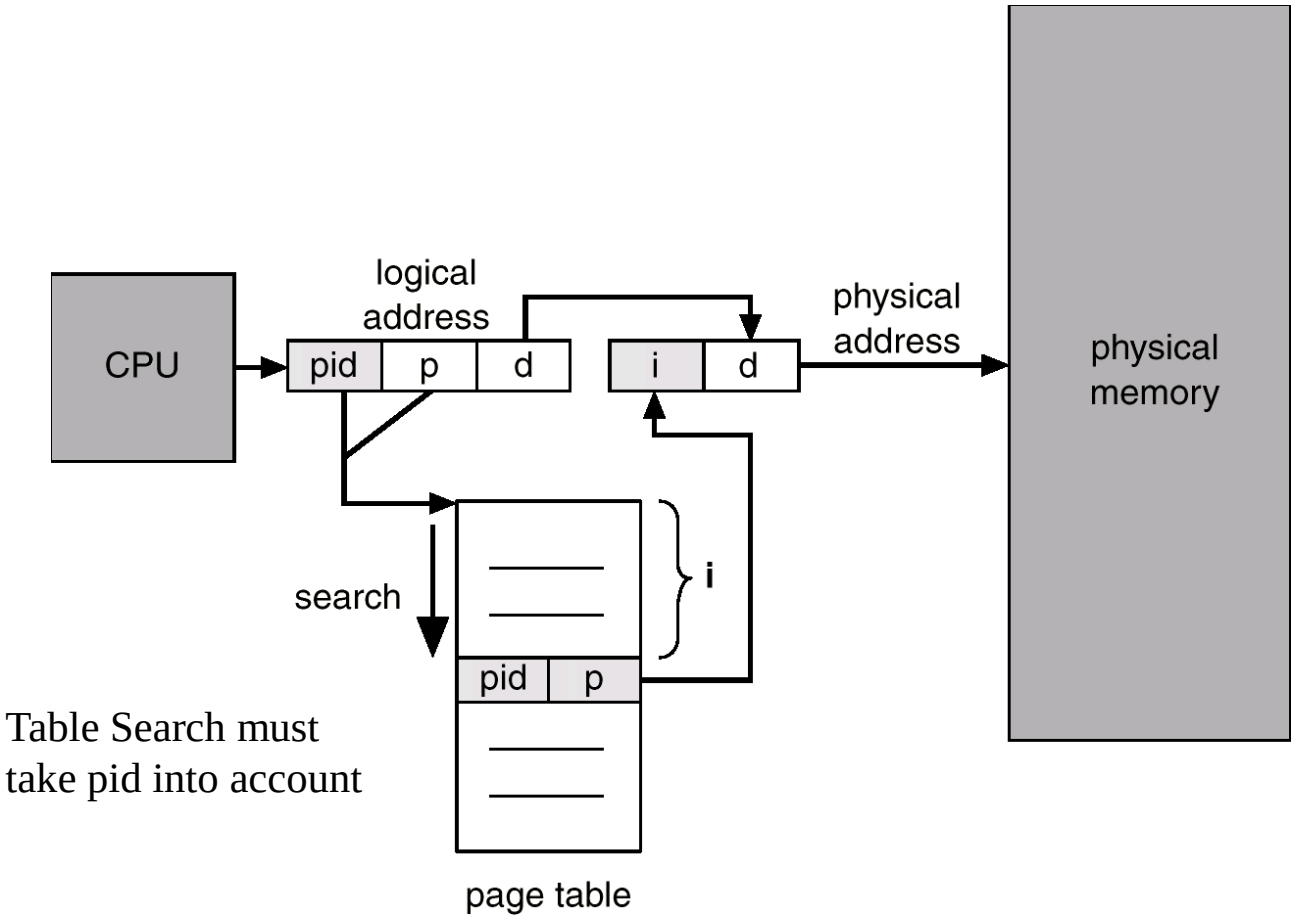
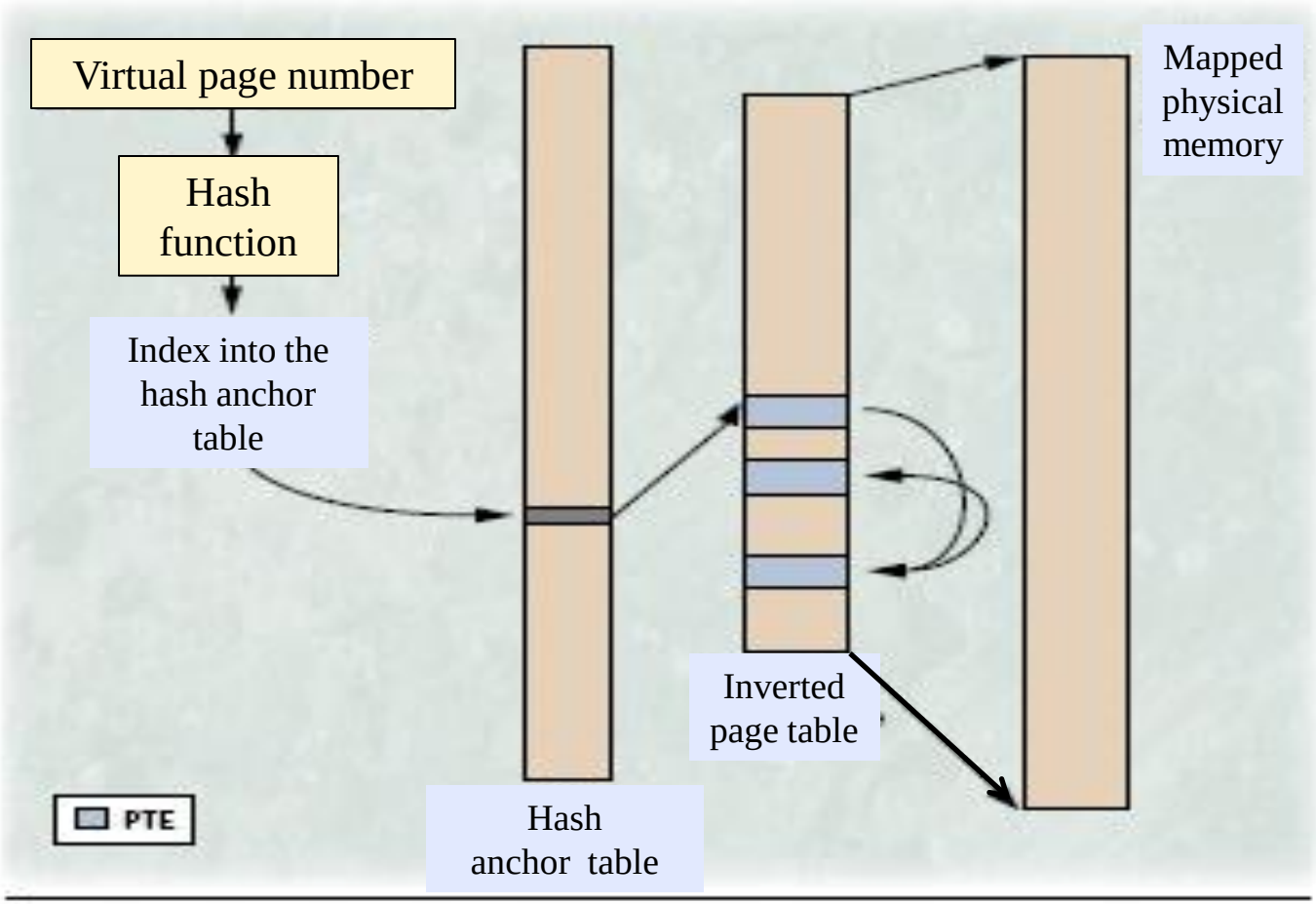


Table Search must take pid into account

Inverted Table with Hashing



The inverted page table contains one PTE for every page frame in memory, making it densely packed compared to the hierarchical page table. It is indexed by a hash of the virtual page number.

Inverted Table with Hashing

- The hash function points into the *anchor hash table*.
- Each entry in the anchor table is the first link in a list of pointers to the inverted table.
- Each list ends with a *Nil* pointer.
- On every memory call the page is looked up in the relevant list.
- TLB still used to prevent search in most cases

Shared Pages