Memory Management

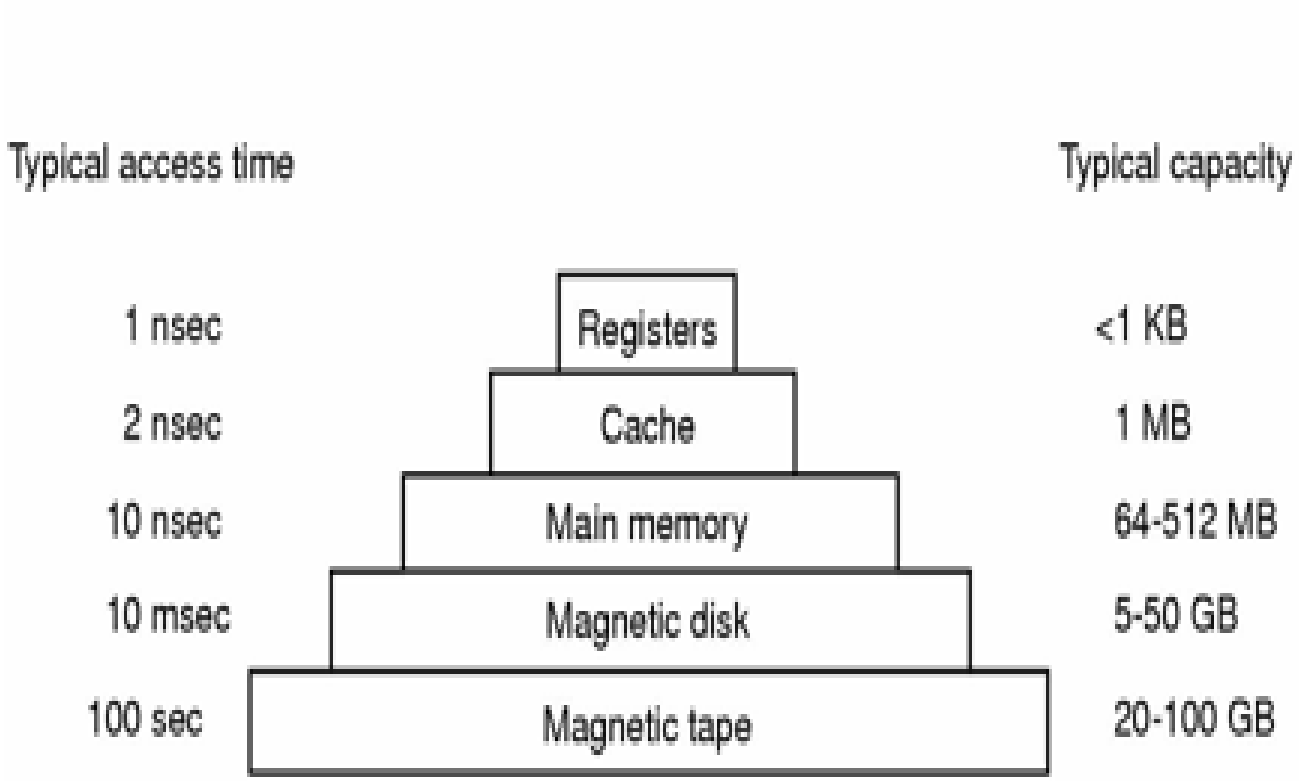
Contents

- Introduction, Monoprogramming vs. Multi-programming, Modelling Multiprogramming, Multiprogramming with fixed and variable partitions, Relocation and Protection.
- Memory management (Bitmaps & Linked-list), Memory Allocation Strategies
- Virtual memory: Paging, Page Table, Page Table Structure, Handling Page Faults, TLB's
- Page Replacement Algorithms: FIFO, Second Chance, LRU, Optimal, LFU, Clock, WS-Clock, Concept of Locality of Reference, Belady's Anomaly
- Segmentation: Need of Segmentation, its Drawbacks, Segmentation with Paging (MULTICS)

Memory

- Main memory (RAM) is an important resource that must be very carefully managed.
- Basically every programmer or even normal computer programmer or user wants infinitely large and fast memory. And the memory is also non-volatile memory.
- The part of the OS that manages the memory hierarchy is called the memory manager.
  - Its job is to keep track of which parts of memory are in use and which parts are not in use.
  - To allocate memory to processes when they need it.
  - To deallocate it when they’re done.
  - To manage swapping between main memory and disk when main memory is too small to hold all the processes.

Computer Hardware Review



- Typical memory hierarchy
  - numbers shown are rough approximations

- Memory management systems can be divided into two basic classes:
  - Those that move processes back and forth between main memory and disk during execution (swapping and paging) and
  - Those that don’t.

Monoprogramming without Swapping or Paging

- The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the OS.
- Three simple ways of organizing memory

-An operating system with one user process,

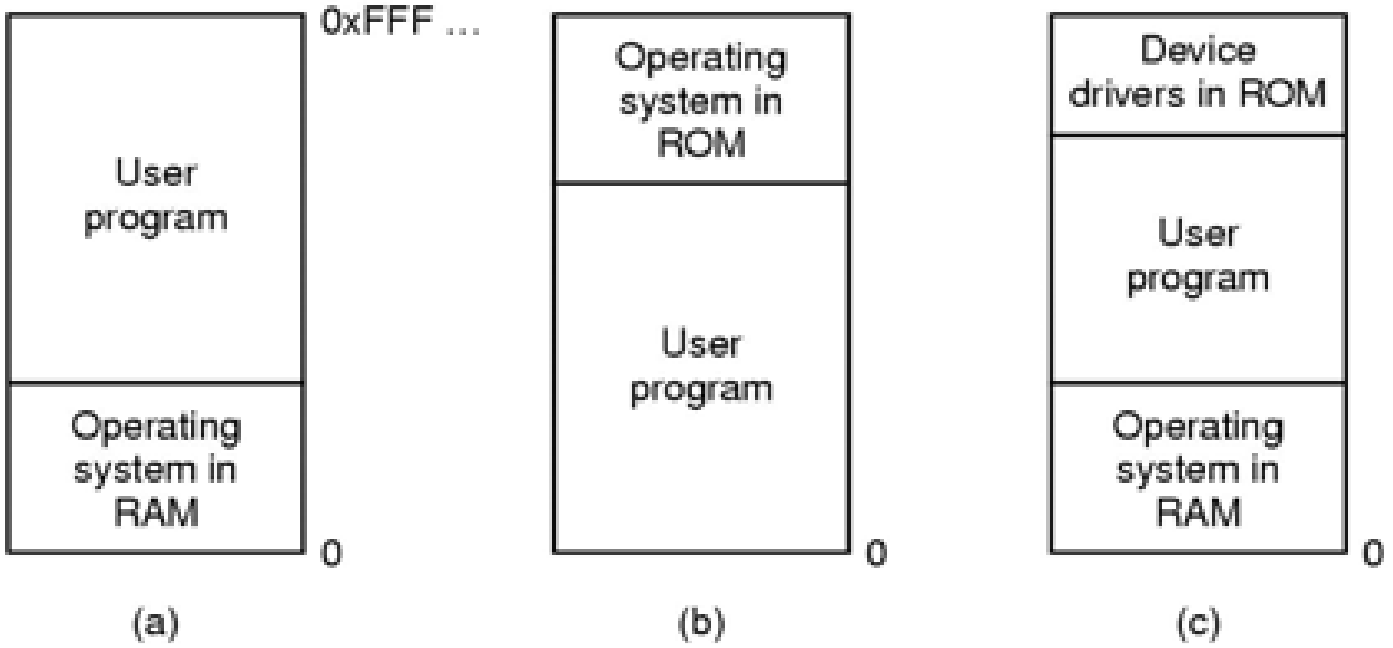


Figure 4-1: Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- The OS may be at the bottom of memory in RAM (a). Or it may be in ROM at the top of memory (b) or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below (c).
- The first model was formerly used on mainframes and minicomputers but is rarely used any more.
- The second model is used on some palmtop computers and embedded systems.
- The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the BIOS.
- When the system is organized in this way, **only one process at a time** can be running.
- As soon as the user types a command, the OS copies the requested program from disk to memory and executes it.
- When the process finishes, the OS displays a prompt character and waits for a new command.
- When it receives the command, it loads a new program into memory, overwriting the first one.

Multiprogramming

- Today, almost all computer systems allow multiple processes to run at the same time.
- When multiple processes running at the same time, means that when one process is blocked waiting for input/output to finish, another one can use the Central Processing Unit.
- Therefore, multiprogramming increases the central processing unit utilization.
- Now-a-day, both network server and client machines have the ability to run multiple processes at the same time.
- To achieve multiprogramming, the simplest way is just to divide memory up into n partitions (normally unequal partitions).
- Multiprogramming lets the processes use the Central Processing Unit when it would be. The Central Processing Unit (CPU) utilization can be improved when multiprogramming is used.

Logical and Physical Address

Logical Address

- Address generated by **CPU** while a program is running is referred as **Logical Address**.
- The logical address is virtual as it does not exist physically. Hence, it is also called as **Virtual Address**. This address is used as a reference to access the physical memory location.
- The set of all logical addresses generated by a programs perspective is called **Logical Address Space**.
- The logical address is mapped to its corresponding physical address by a hardware device called **Memory-Management Unit**. The address-binding methods used by MMU generates **identical** logical and physical address during **compile time** and **load time**.

Physical Address

- **Physical Address** identifies a physical location in a memory.
- MMU (**Memory-Management Unit)** computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address.
- The logical address is mapped to the physical address using a hardware called **Memory-Management Unit**.
- The set of all physical addresses corresponding to the logical addresses in a Logical address space is called **Physical Address Space**.

Contiguous memory allocation	Noncontiguous memory allocation
• Program execution takes place without the overhead.	• Address translation is overhead.
• Swapped-in processes are placed in the original area.	• Swapped-in processes can be placed anywhere in memory.
• Suffer from external fragmentation.	• Only paging suffers from internal fragmentation.
• Allocates a single area of memory for the process.	• Allocates more than one block of memory for the process.
• Wastage of memory	• No wastage of memory.
• Fixed-sized partition, variable partition	• Paging, multi-level paging, inverted paging and segmentation

Memory Management Techniques

- Before details about moving processes out

  -Assign memory to processes

- Memory partitioning
  - Fixed partitioning
  - Dynamic partitioning
  - Simple paging
  - Simple segmentation
  - Virtual memory paging
  - Virtual memory segmentation

Memory Allocation

Multiprogramming with Fixed Partitions

- Except on very simple embedded systems, monoprogramming is hardly used any more.
- Most modern systems allow multiple processes to run at the same time.
- Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU.
  - –Multiprogramming increases the CPU utilization.
- The easiest way to achieve multiprogramming is simply to divide memory up into *n* (possibly unequal) partitions.
- This partitioning can, for example, be done manually when the system is started up.

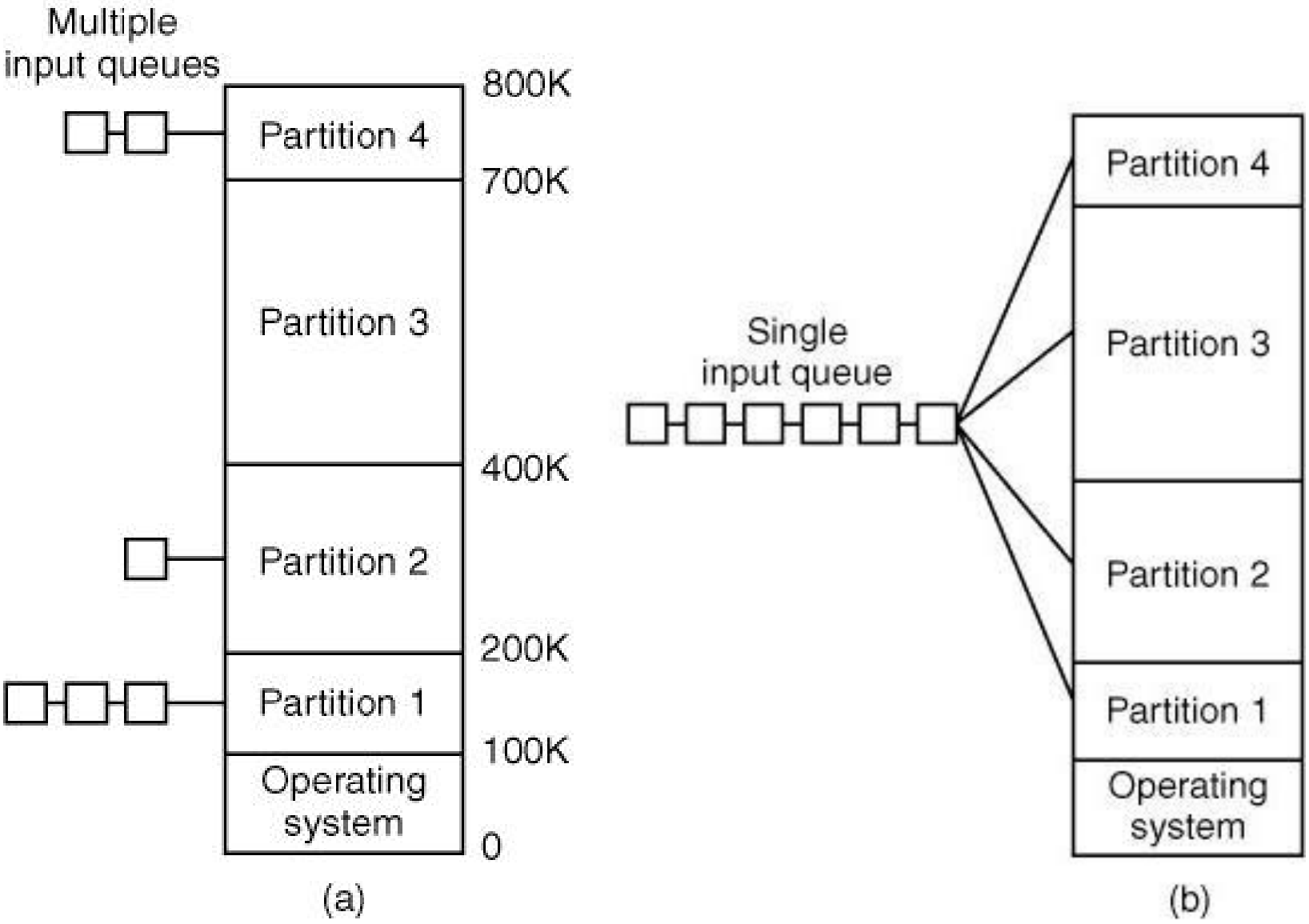


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition.(b) Fixed memory partitions with a single input queue.

- In figure (a) we see how this system of fixed partitions and separate input queues look.

–The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 & 4 in (a).

- An alternative organization is to maintain a single queue as in (b).

–Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

–Since it's undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that

- Two fixed partitioning schemes

  -Equal-size partitions

  -Unequal-size partitions

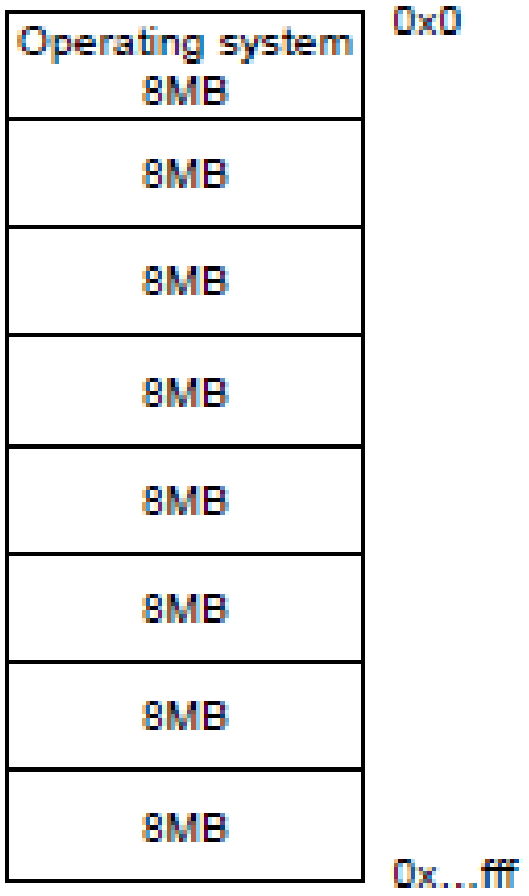
Equal-size partitions

- Big programs can not be executed

  -Unless program parts are loaded from disk

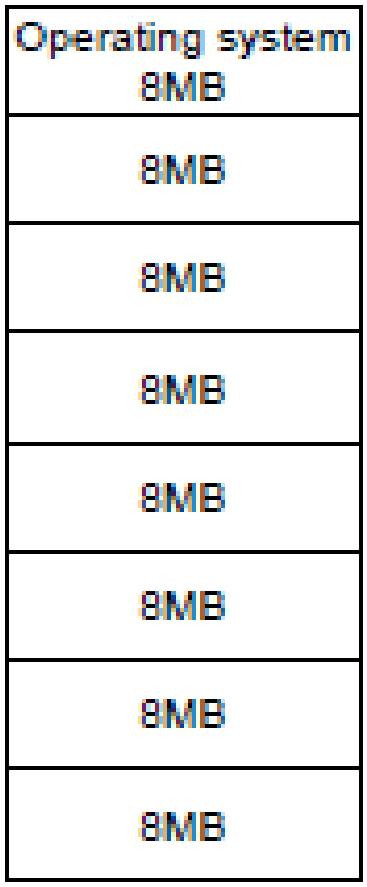
- Small programs use entire partition

  -A problem called “internal fragmentation”



Unequal-size partitions

- Bigger programs can be loaded at once
- Smaller programs can lead to less internal fragmentation
- Advantages require assignment of jobs to partitions



Fragmentation

- Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.
- **Internal fragmentation** is when a process is allocated more memory than required, and therefore, few space is left unused.

  For example: When process 1 with the size of 10KB is allocated a block of 12KB, 2KB space is left unused. Now, when process 2, with size 10 KB is allocated a 0KB block, no space is left unused. When process 3 with size 12KB is allocated a 13 KB block, 1KB is left unused.
- From this example, 2 processes are allocated space more than required and this unused space is so small to store a new process and is wasted.
- Systems with fixed-sized allocation units, such as the single partitions scheme and paging suffer from internal fragmentation.
- First-fit and best-fit memory allocation does not suffer from internal fragmentation.

- **External Fragmentation happens** when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used.
- If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request but it is not contiguous.
- Systems with variable-sized allocation units, such as the multiple partitions scheme and segmentation suffer from external fragmentation.
- First-fit and best-fit memory allocation suffers from external fragmentation.

Requirements of Memory management

Relocation and Protection

- Multiprogramming introduces two essential problems that must be solved:
  - –Relocation and protection.
- From the previous two figures it is clear that different jobs will be run at different addresses.
  - –When a program is linked (i.e., the main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.
  - –For example, suppose that the first instruction is a call to a procedure at absolute address 100 within the binary fire produced by the linker.
  - –If this program is loaded in partition 1 (at address 100K), that instruction will jump to absolute address 100, which is inside the OS.
  - –What is needed is a call to 100K + 100.
  - –If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. -> this is the relocation problem.

- The classical solution, which was used on machines ranging from the CDC 6600 (the world's first supercomputer) to the Intel 8088 (the heart of the original IBM PC), is to equip each CPU with two special hardware registers, usually called the **base and limit registers.**
  - –When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition.
  - –Every memory address generated automatically has the base register contents added to it before being sent to memory.
  - –Thus if the base register contains the value 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 instruction, without the instruction itself being modified.

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

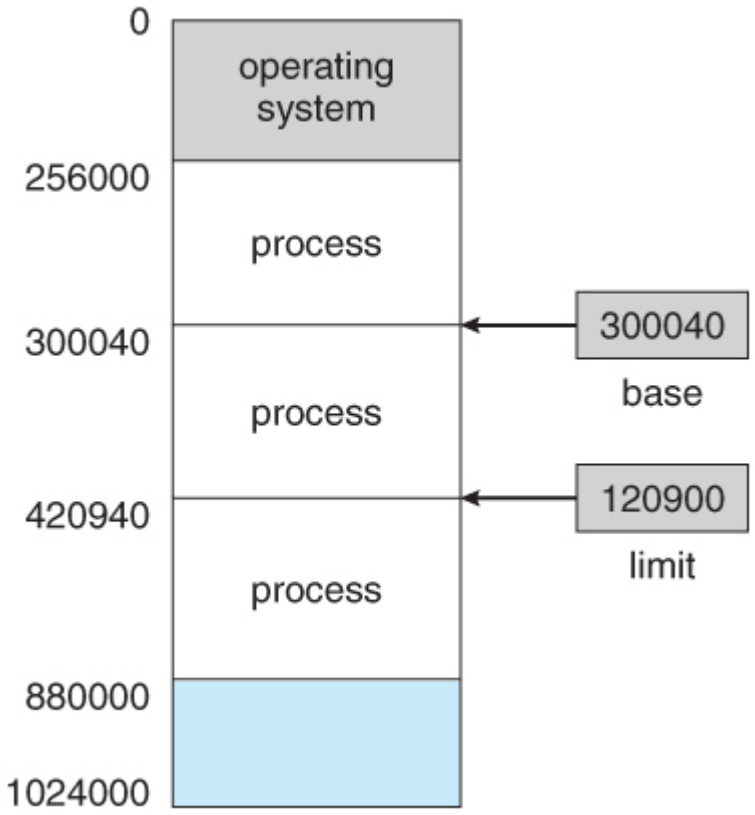


Figure 4-3 : A base and a limit register define a logical address space

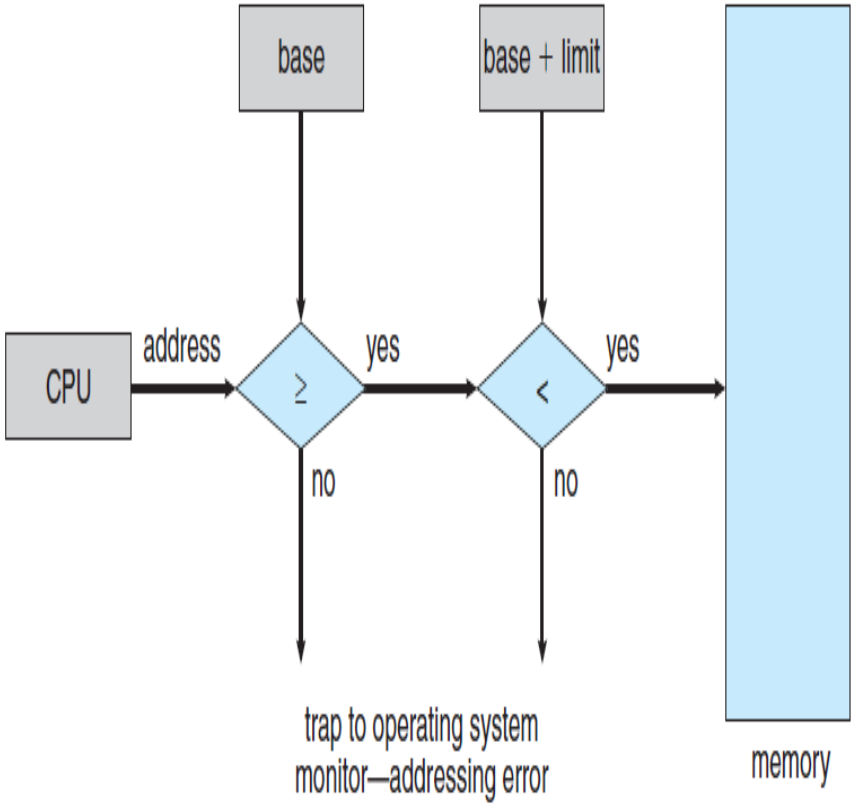


Figure 4-4 : Hardware Address Protection

–Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition.

–The hardware protects the base and limit registers to prevent user programs from modifying them.

- A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry-propagation time unless special addition circuits are used.
- Few computers use it now.

Memory Protection

- Memory protection implemented

– by associating protection bit with each frame

– to indicate if read-only or read-write access is allowed

– Can also add more bits to indicate page execute-only, and so on

- **Valid-invalid** bit attached to each entry in the page table:
- When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to "invalid," the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid -invalid bit.

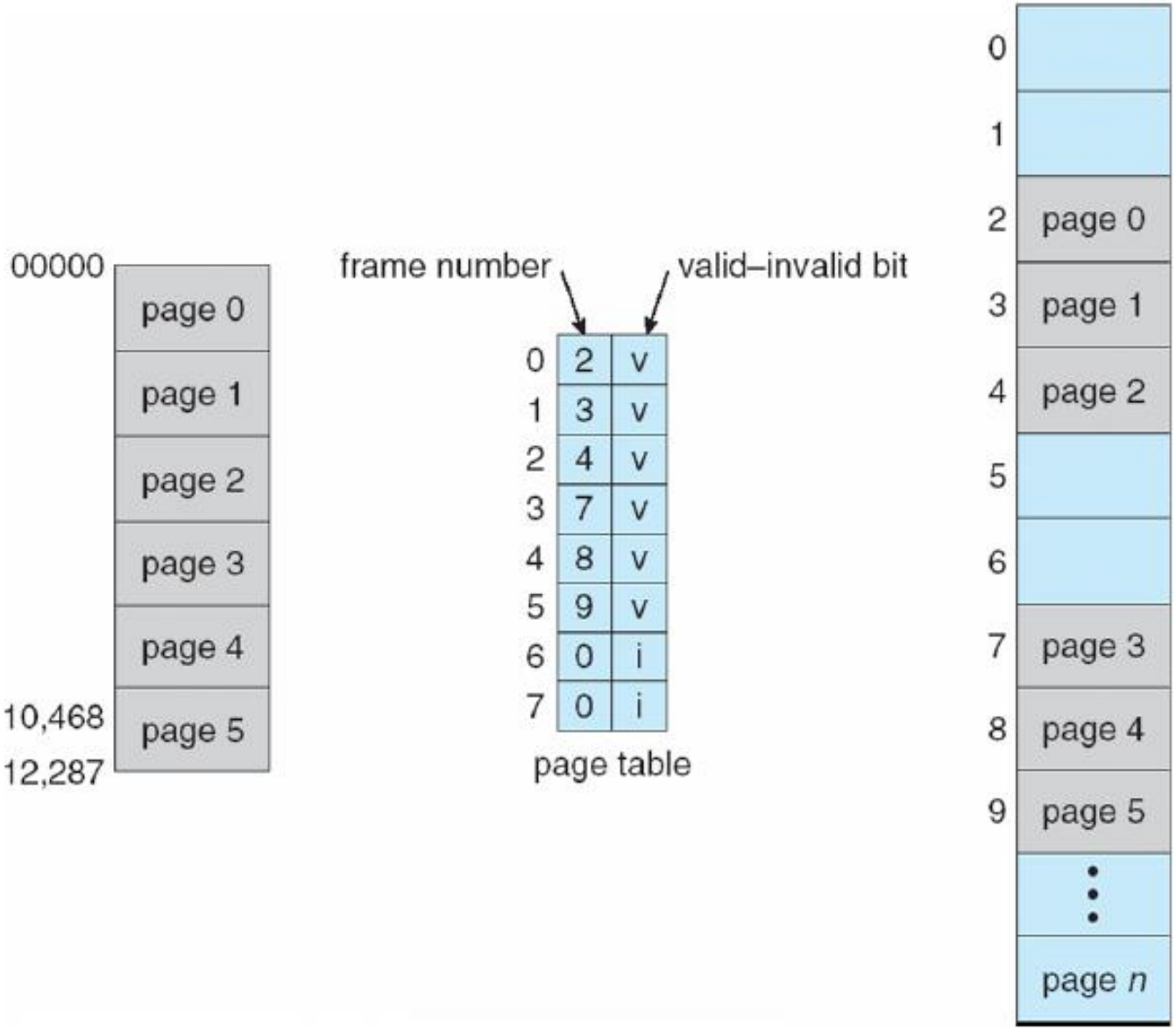


Figure 4-5 : Valid (v) or Invalid (i) Bit In A Page Table

Shared Pages

- **Shared code**

– One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)

– Similar to multiple threads sharing the same process space

– Also useful for inter-process communication if sharing of read-write pages is allowed

- **Private code and data**

– Each process keeps a separate copy of the code and data

– The pages for the private code and data can appear anywhere in the logical address space

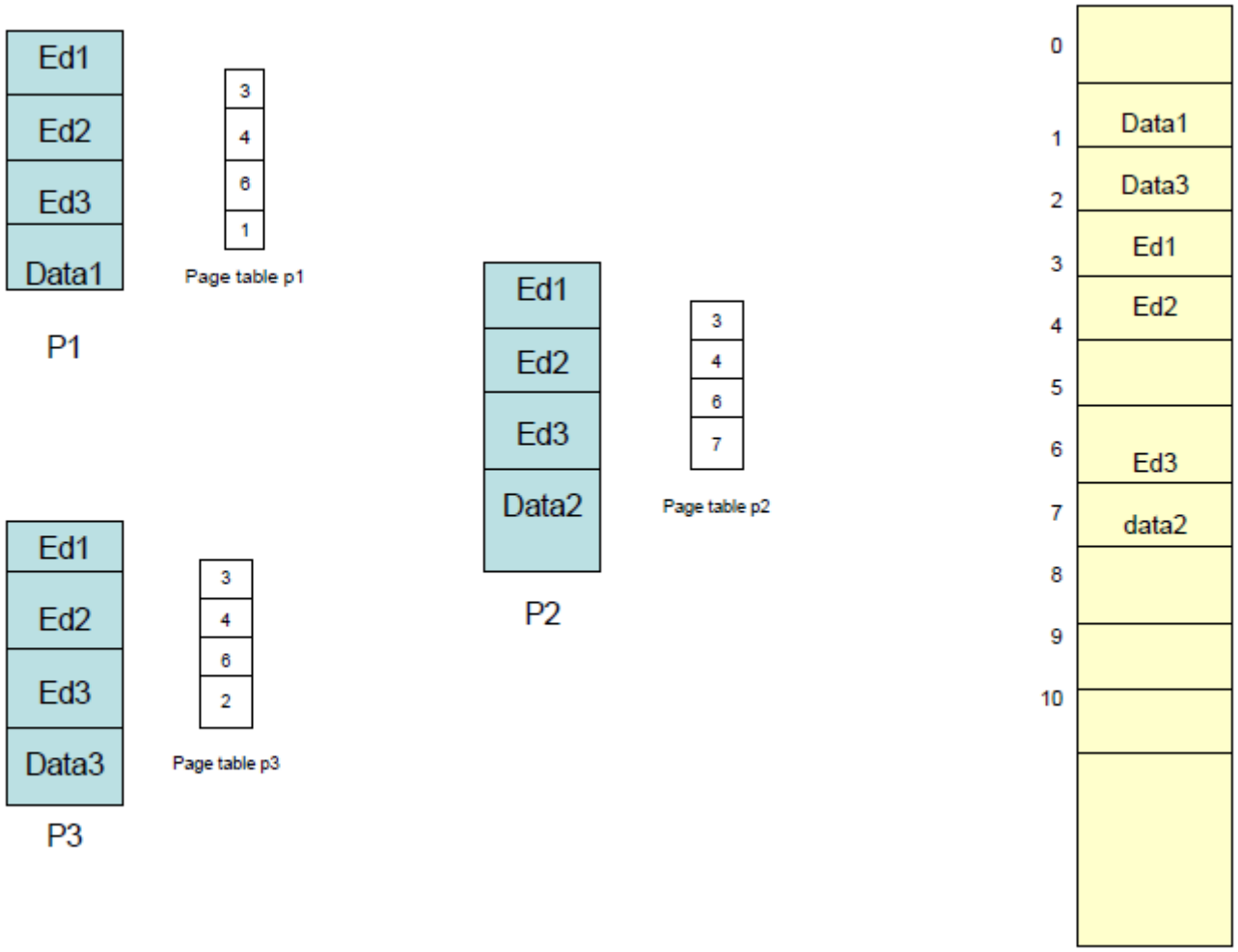


Figure 4-6 : Shared Pages Example

- Two general approaches to memory management can be used, depending on the available hardware:
  - –**Swapping** (the simplest strategy that consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk) and
  - –**Virtual memory** (which allows programs to run even when they are only partially in main memory).
- **Swapping** is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes.
- It is used to improve main memory utilization.
- At some later time, the system swaps back the process from the secondary storage to main memory.

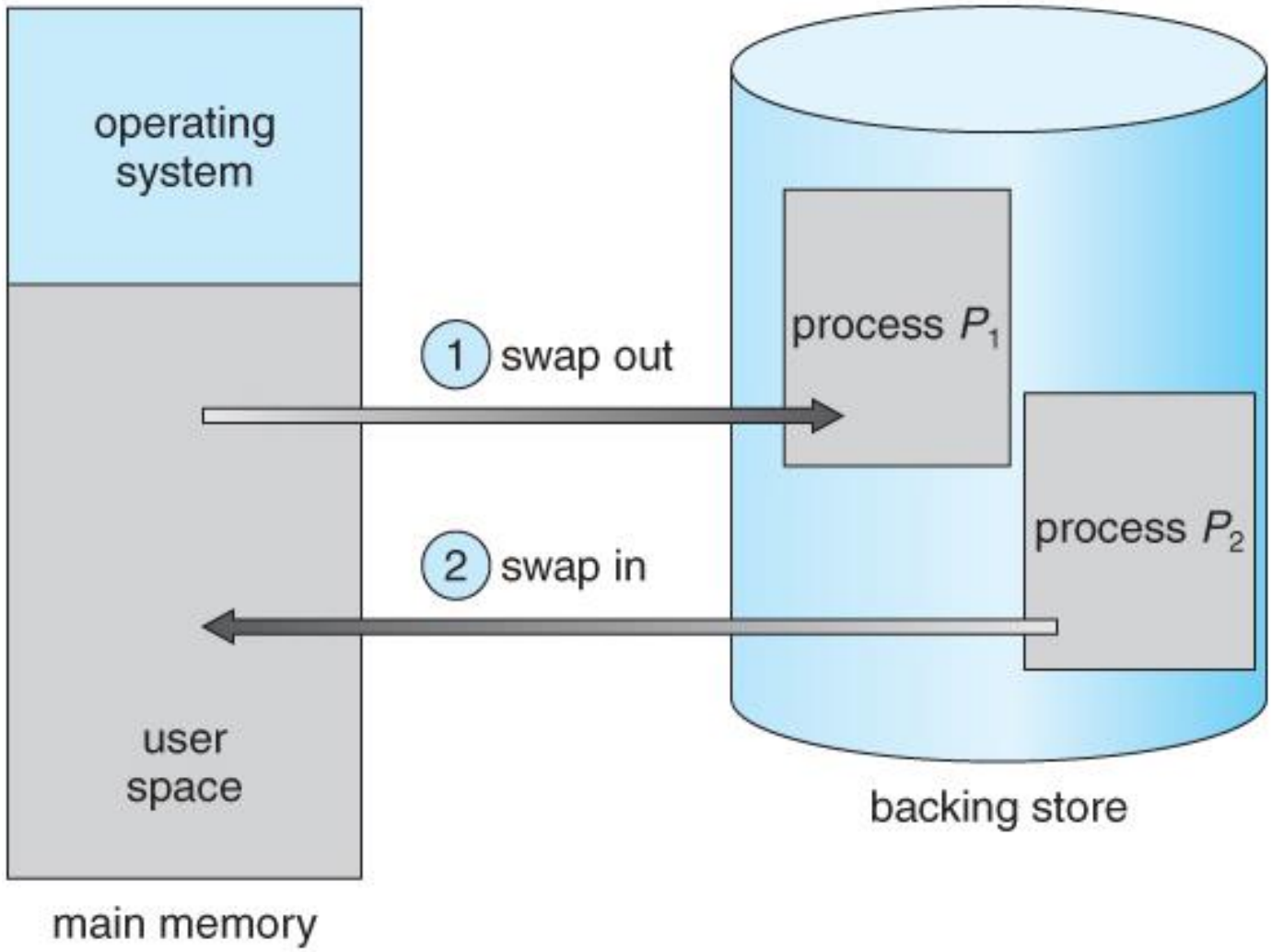


Figure 4-7 : Swapping

- The operation of a swapping system is shown below:

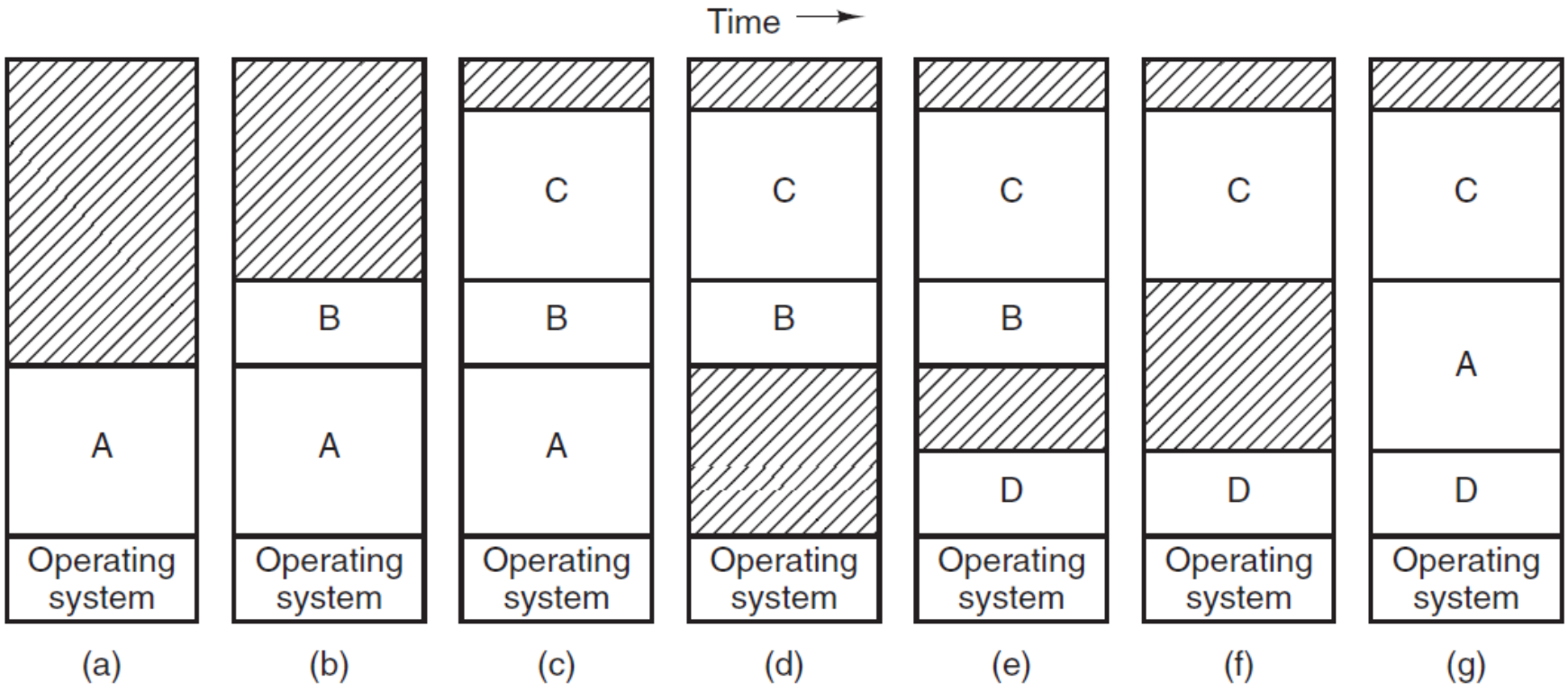


Figure 4.8 : Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

- Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk.
- In Fig.4-8(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again.
- Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.
- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time.
- when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well.

- In Fig 4-9 (a) we see a memory configuration in which space for growth has been allocated to two processes.

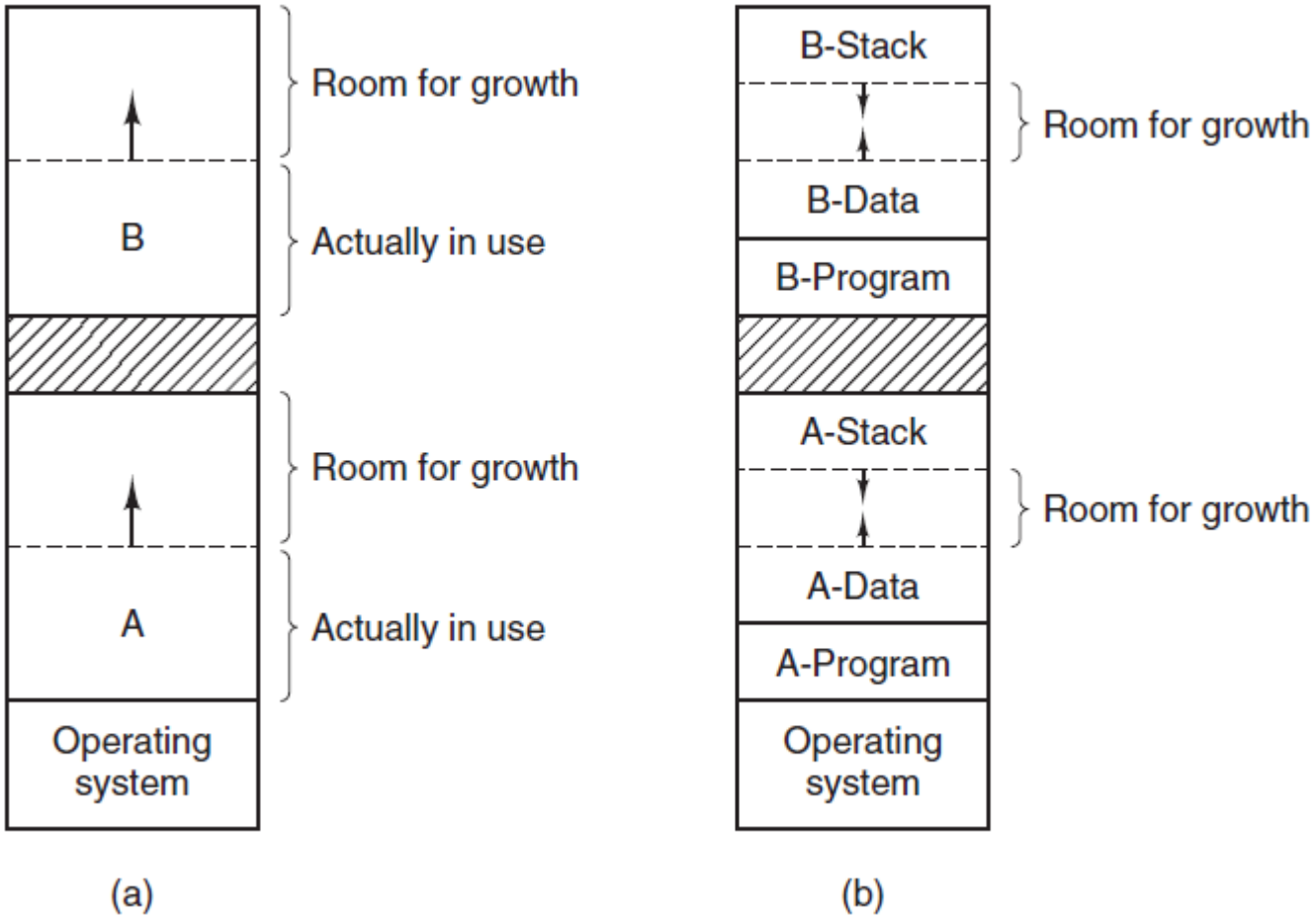


Figure 4-9 : (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

- If processes can have two growing segments—for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses
- In (b) we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward.
- The memory between them can be used for either segment. If it runs out, the process will either have to be moved to a hole with sufficient space, swapped out of memory until a large enough hole can be created, or killed.

Managing Free Memory

Memory management with Bitmaps

- When memory is assigned dynamically, the operating system must manage it.
- In general terms, there are two ways to keep track of memory usage: bitmaps and free lists.
- With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes.
- Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).
- The next figure shows part of memory and the corresponding bitmap.

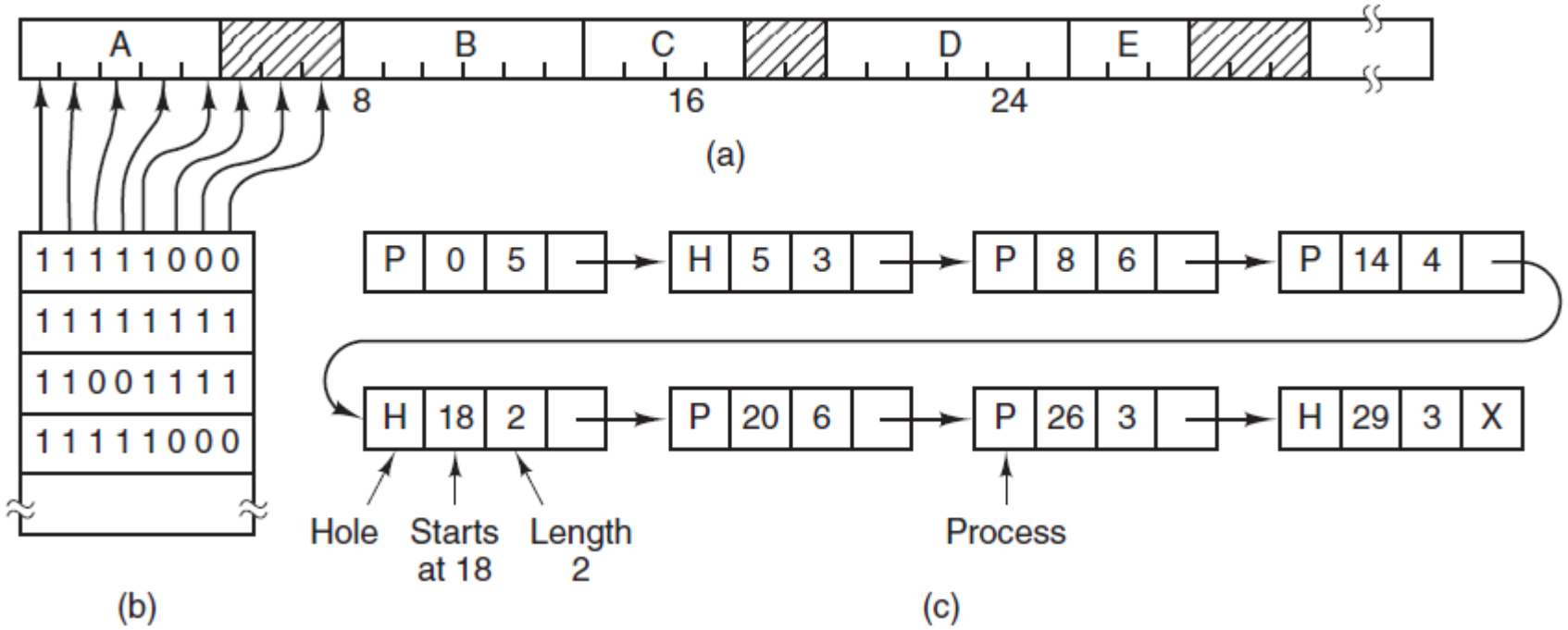


Figure 4-10 : (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

- The size of the allocation unit is an important design issue.
- The smaller the allocation unit, the larger the bitmap.
- A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit.
- The main problem with it is that when it has been decided to bring a k unit process into memory, the mem manager must search the bitmap to find a run of k consecutive 0 bits in the map
- searching a bitmap for a run of a given length is a slow operation

Memory Management with Linked Lists

- Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.
- When processes and holes are kept on a list that is sorted by address, several algorithms can be used to allocate memory for a newly created process or an existing process being swapped in from the disk.
- There are different types of algorithms like, first fit, best fit, next fit, worst fit, quick fit.
- The memory of Fig. 4-11(a) is represented in Fig. 4-11(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

Figure 4-11 : Four neighbor combinations for the terminating process, X.

- A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory).
- These may be either processes or holes, leading to the four combinations shown below.
  - –In (a) updating the list requires replacing a P by an H.
  - –In (b) and also in (c), two entries are coalesced into one, and the list becomes one entry shorted.
  - –In (d), three entries are merged and two items are removed from the list.
- The process of merging two adjacent holes to form a single larger hole is called **coalescing** .
- It is possible to combine all the holes into one big one by moving all the processes downward as far as possible; this technique is called memory **compaction**.
- When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk).

Memory Allocation Strategies

First fit

- The process manager scans along the list of segments until it finds a hole that is big enough.
- The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit.
- First fit is a fast algorithm because it searches as little as possible.

Next fit

- It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.
- The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always beginning, as first fit does.
- Simulations by bays (1977) show that next fit gives slightly worse performance than first fit.

Best fit

- Best fit searches the entire list and takes the smallest hole that is adequate.
- Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.
- Best fit is slower that first fit because it must search the entire list every time it is called.
- Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes (first fit creates larger holes on average).

worst fit

- To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about worst fit, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful.
- Simulation has shown that worst fit is not a very good idea either.

Quick fit

- Quick fit maintains separate lists for some of the more common sizes requested.
- For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-kb holes, the second entry is a pointer to a list of 8-kb holes, the third entry a pointer to 12-kb holes, and so on.
- Holes of say, 21-kb, could either be put on the 20-kb list or on a special list of odd-sized holes.
- With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all the other scheme that sort by hole size.

Example:

Given five memory partitions of 100KB, 500KB, 200KB, 300KB, 600KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

Solution:

First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K - 212K)

426K must wait

Best-fit:

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Worst-fit:

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, best-fit turns out to be the best.

Virtual Memory

- Many years ago people were first confronted with programs that were too big to fit in the available memory.
- A solution adopted in the 1960s was to split programs into little pieces, called **overlays**.
- Overlay 0 would start running first, When it was done, it would call another overlay.
- Some overlay systems were highly complex, allowing many overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the overlay manager.
- Although the actual work of swapping overlays in and out was done by the operating system, the work of splitting the program into pieces had to be done manually by the programmer.
- Splitting large programs up into small, modular pieces was time consuming, boring, and error prone.

- The method that was devised has come to be known as **virtual memory**. The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called **pages**.
- Virtual memory, or virtual storage is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" which "creates the illusion to users of a very large memory".
- These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program.

Paging

- **Paging** is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages.

- Memory Management Unit generates physical address from virtual/logical address provided by the program.

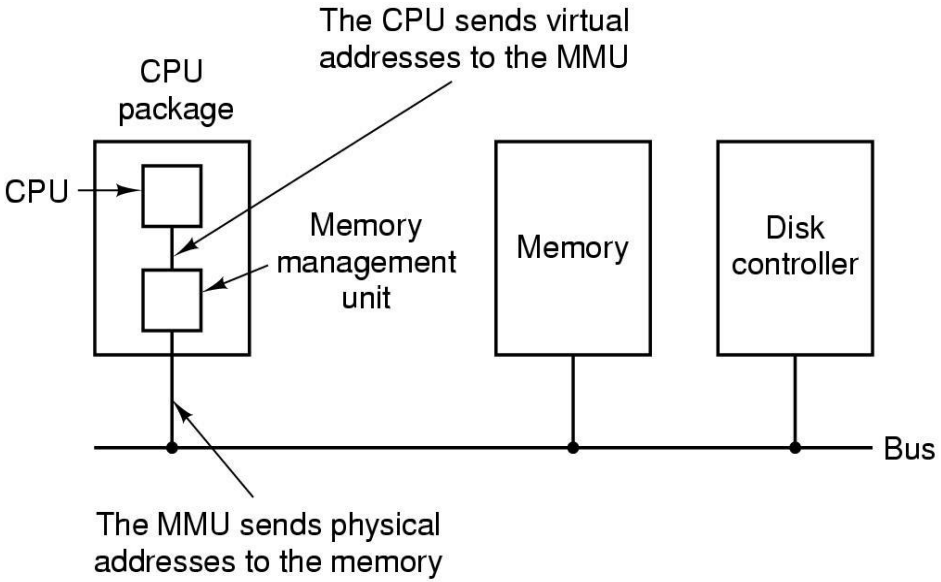


Figure 4-12 : MMU maps virtual addresses to physical addresses and puts them on memory bus

- When a program uses an instruction like:
  - –MOV REG, 1000
- It does this to copy the contents of memory address 1000 to REG.
- Addresses can be generated using indexing, base registers, segment registers, etc.
- These program-generated addresses are called virtual addresses and form the **virtual address space**.
- When virtual memory is used, the virtual addresses do not directly go to the memory bus.
- Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses

Pages and Page Frames

- The virtual address space consists of fixed-size units called **pages**. The corresponding units in the physical memory are called **page frames**.
- The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation.
- Virtual addresses divided into pages
  - 512 bytes-64 KB range
  - Transfer between RAM and disk is in whole pages
  - Example on next slide

Mapping of pages to page frames

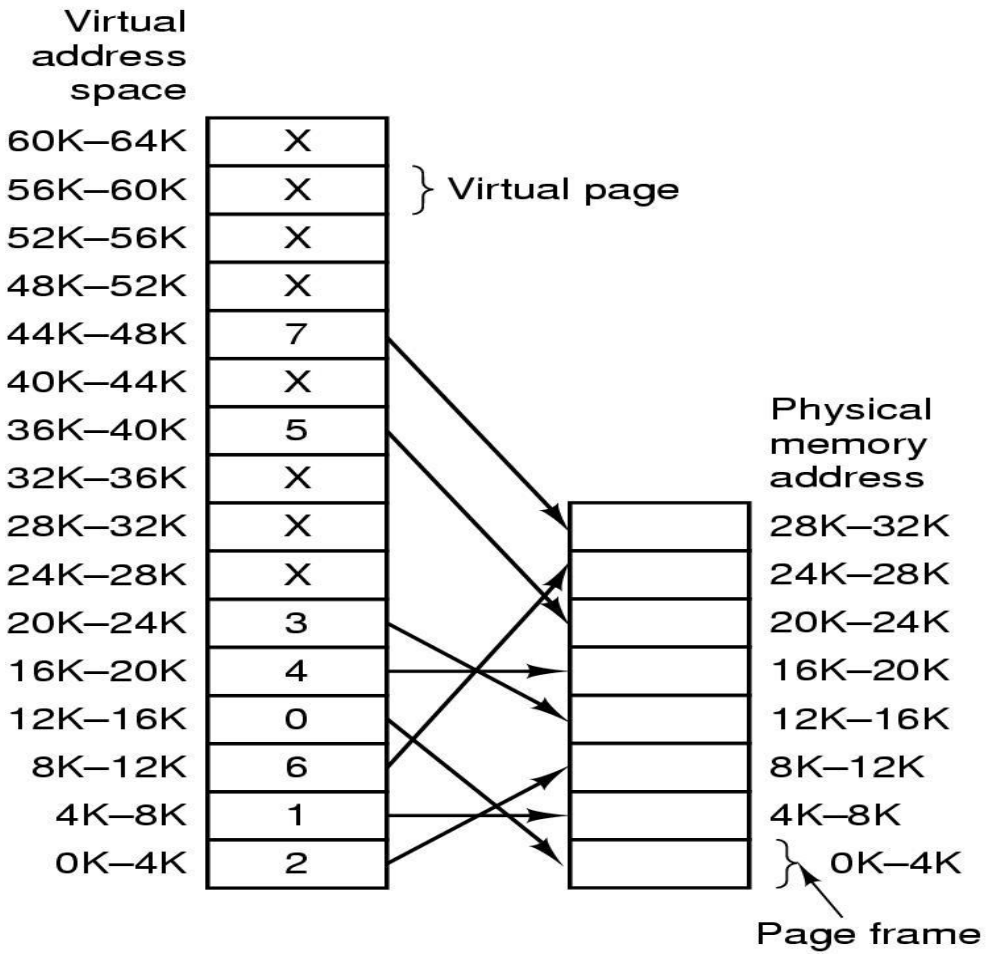


Figure 4-13 : The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

Page Fault Processing

- Present/absent bit tells whether page is in memory
- What happens If address is not in memory?
- Trap to the OS
  - OS picks page to write to disk
  - Brings page with (needed) address into memory
  - Re-starts instruction

Address Translation Scheme

- The addresses generated by the machine while executing in user mode are logical addresses. The paging hardware translates these addresses to physical addresses.
- Address generated by CPU 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 MMU.

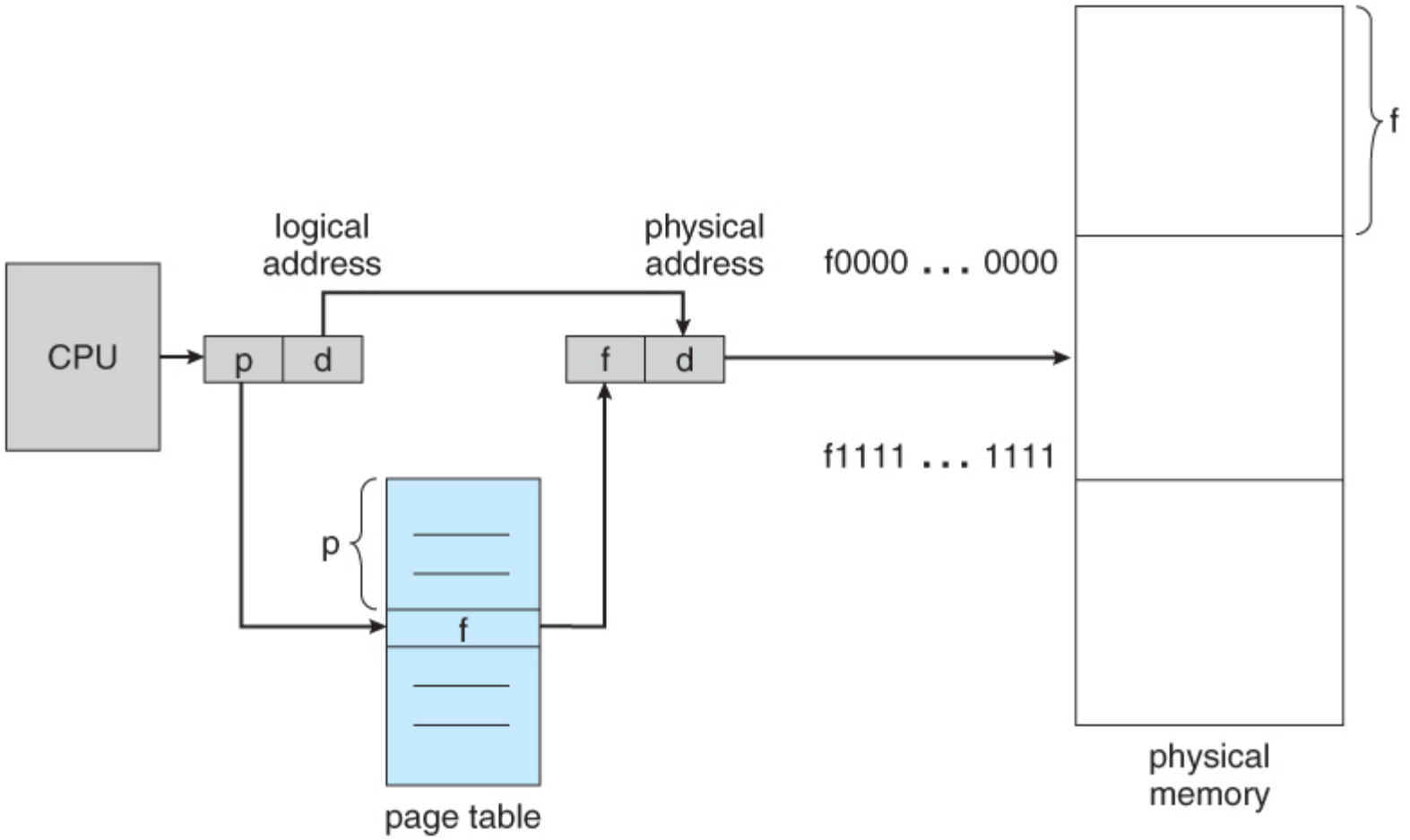


Figure 4-14 : Address Translation (logical to physical address)

Page Table

- The **page table** is used to look up what frame a particular page is stored in at the moment. or
- A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses
- Virtual address={virtual page number, offset}
- Virtual page number used to index into page table to find page frame number
- If present/absent bit is set to 1, attach page frame number to the front of the offset, creating the physical address
- which is sent on the memory bus

Address translation Example-MMU operation

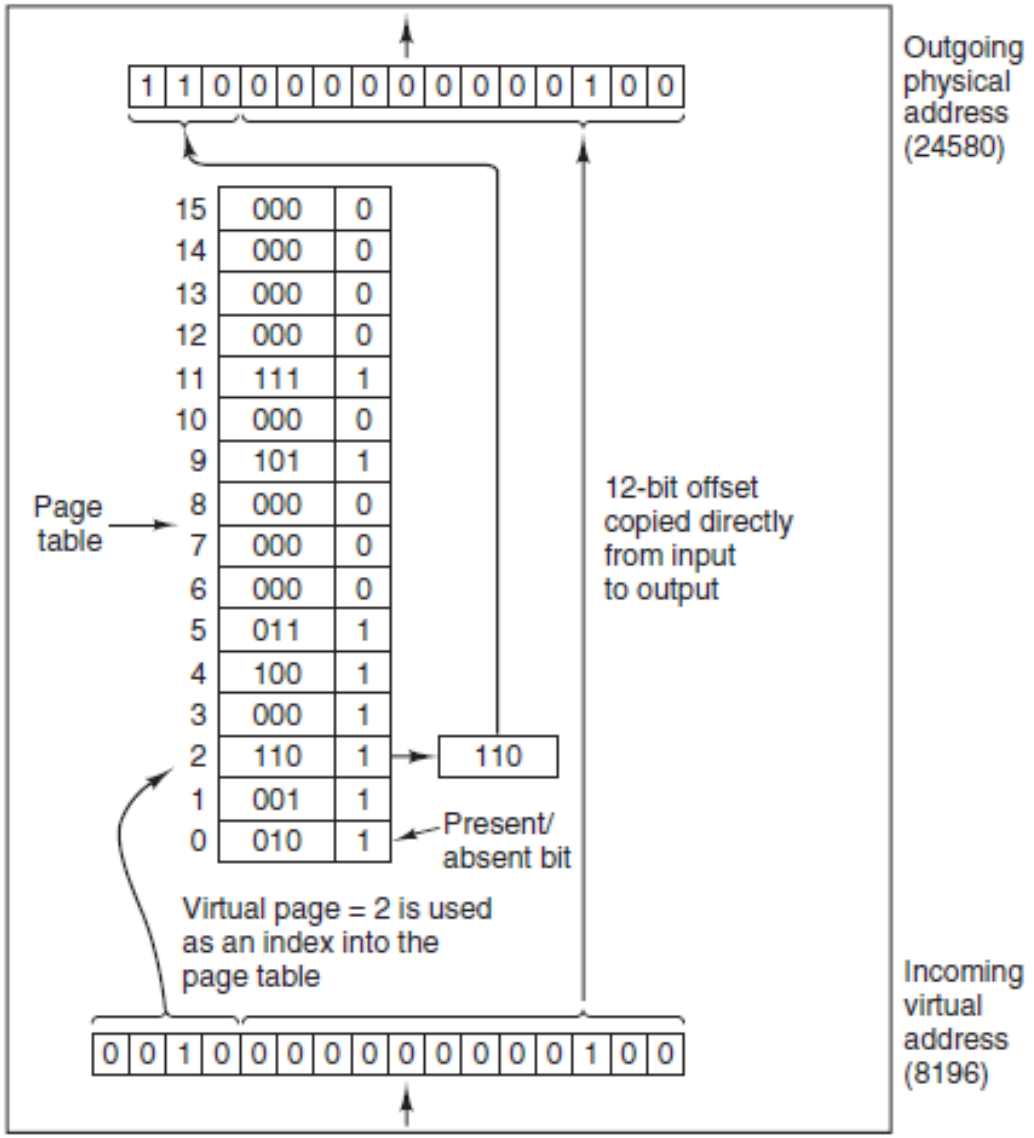


Figure 4-15 : The internal operation of the MMU with 164-KB pages.

Page Tables for Large Memories

Multi-level page tables

- Want to avoid keeping the entire page table in memory because it is too big
- Hierarchy of page tables does this
- The hierarchy is a page table of page tables

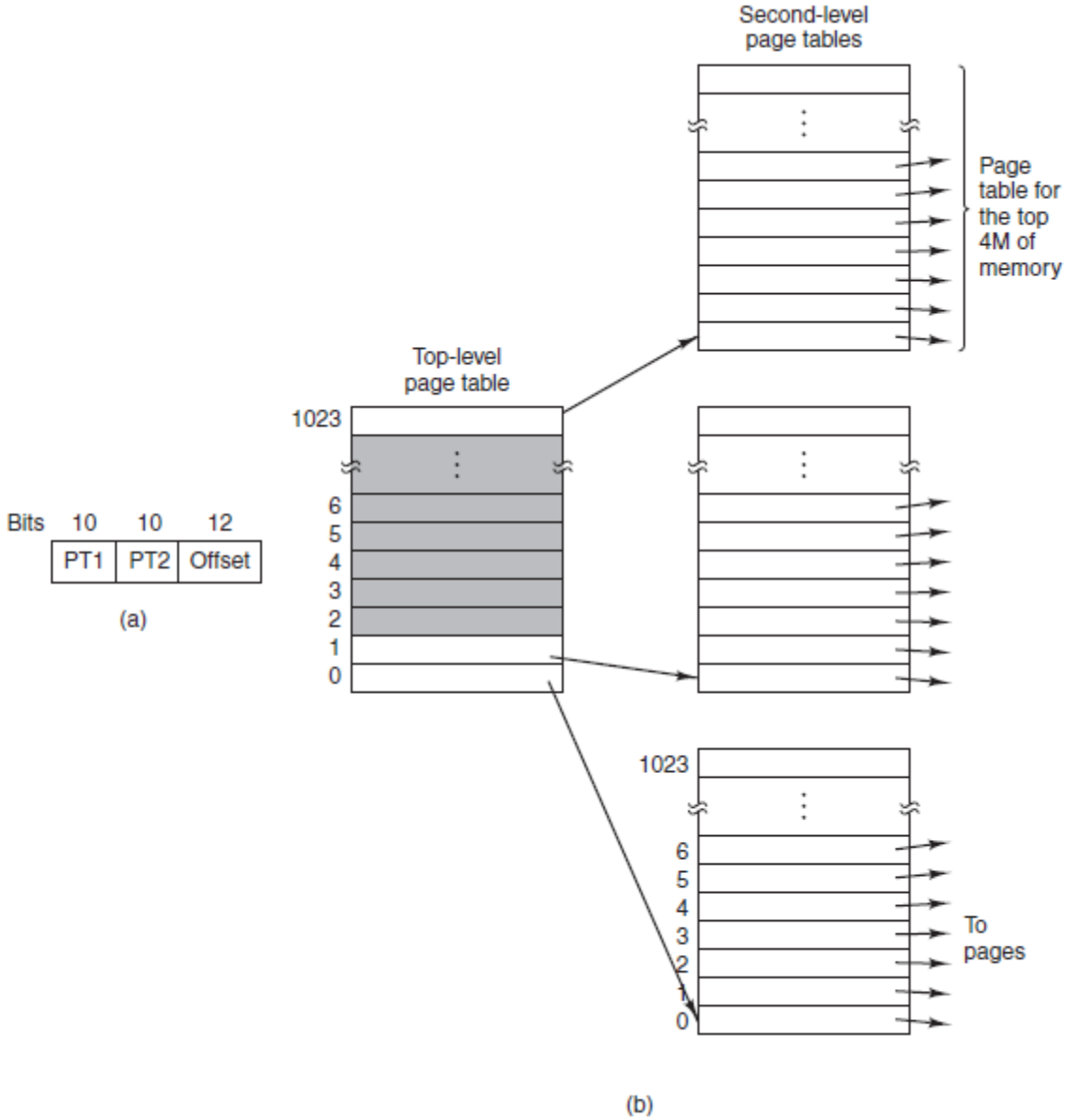


Figure 4-16 : (a) A 32-bit address with two page table fields. (b) Two-level page tables.

- In (a) we have a 32-bit virtual address that 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 4KB, and there are a total of 2^20 of them.
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
- In particular, those that are not needed should not be kept around.
- In (b) we see how the two-level page table works.
  - On the left we have the top-level page table, with 1024 entries, corresponding to the 10-bit PT1 field.
  - When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table.
  - Each of these 1024 entries represents 4M because the entire 4-gigabyte virtual address space has been chopped into chunks of 1024 bytes.
- The entry located by indexing into the top-level page table yields the address of the page frame # of a second-level page table.

Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits.
- The virtual page number is hashed into a page table

– This page table contains a chain of elements hashing to the same location

- Each element contains

  (1) the virtual page number

  (2) the value of the mapped page frame

  (3) a pointer to the next element

- Virtual page numbers are compared in this chain searching for a match

– If a match is found, the corresponding physical frame is extracted

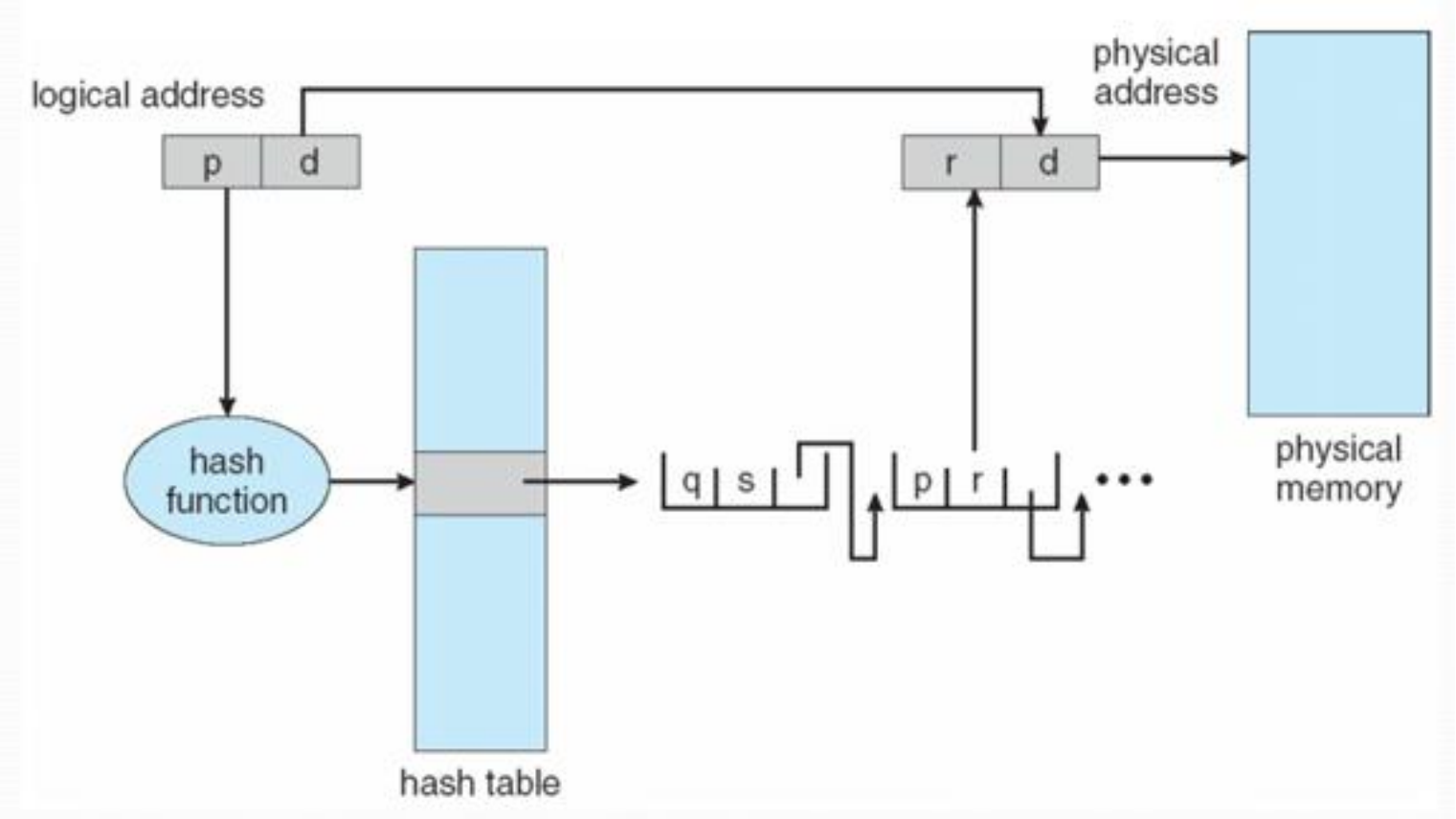


Figure 4-17 : Hashed Page Table

Inverted Page Tables

- Rather than each process having a page table and keeping track of all possible logical pages,

– track all physical pages

- One entry for each real page of memory
- Entry consists of

– the virtual address of the page stored in that real memory location,

– information about the process that owns that page

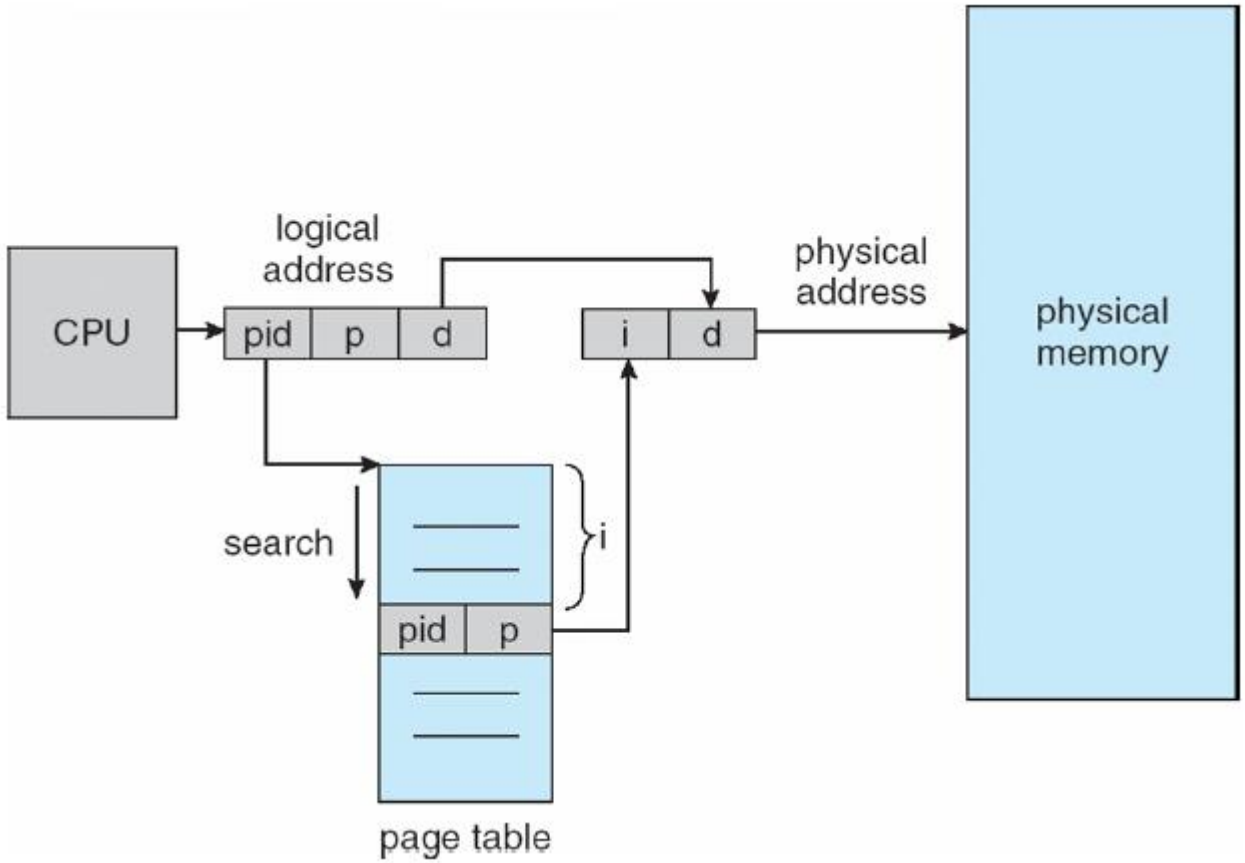


Figure 4-18 : Inverted Page Table

Structure of Page Table Entry

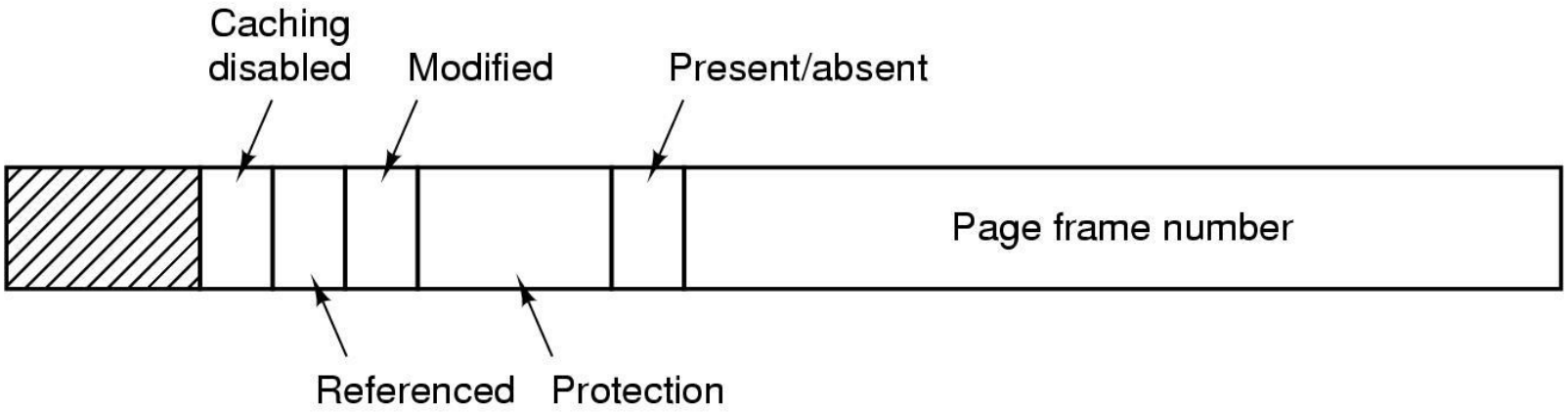


Figure 4-19 : A typical page table entry.

- The figure above shows a sample page entry.
- The exact layout of a page table entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine.
- The size varies from computer to computer, but 32 bits is a common size.

- The most important field is the **page frame number**.
  - The goal of the page mapping is to locate this value.
- Next to it we have the **present/absent** bit.
  - If this bit is 1, the entry is valid and can be used.
  - If it is 0, the virtual page to which the entry belongs is not currently in memory.
  - Accessing a page table entry with this bit set to 0 causes a page fault.
- The **Protection** bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only.
- The **Modified and Referenced** bits keep track of page usage. When a page is written to, the hardware automatically sets the Modified bit. This bit is of value when the operating system decides to reclaim a page frame.

- If the page in it has been modified (i.e., is “dirty”), it must be written back to the disk. If it has not been modified (i.e., is “clean”), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page’s state.
- The **Referenced** bit is set whenever a page is referenced, either for reading or for writing.
- Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory.

Locality of reference

- The concept of locality of reference states that, instead of loading the entire process in the main memory, OS can load only those number of pages in the main memory that are frequently accessed by the CPU and along with that, the OS can also load only those page table entries which are corresponding to those many pages.

Speeding Up Paging

Problems for paging:

1. The mapping from virtual address to physical address must be fast.

2. If the virtual address space is large, the page table will be large. 32 bit addresses now and 64 bits becoming more common

- A **translation lookaside buffer** (**TLB**) is a memory cache that is used to reduce the time taken to access a user memory location.
- It is a part of the chip's memory-management unit (MMU).

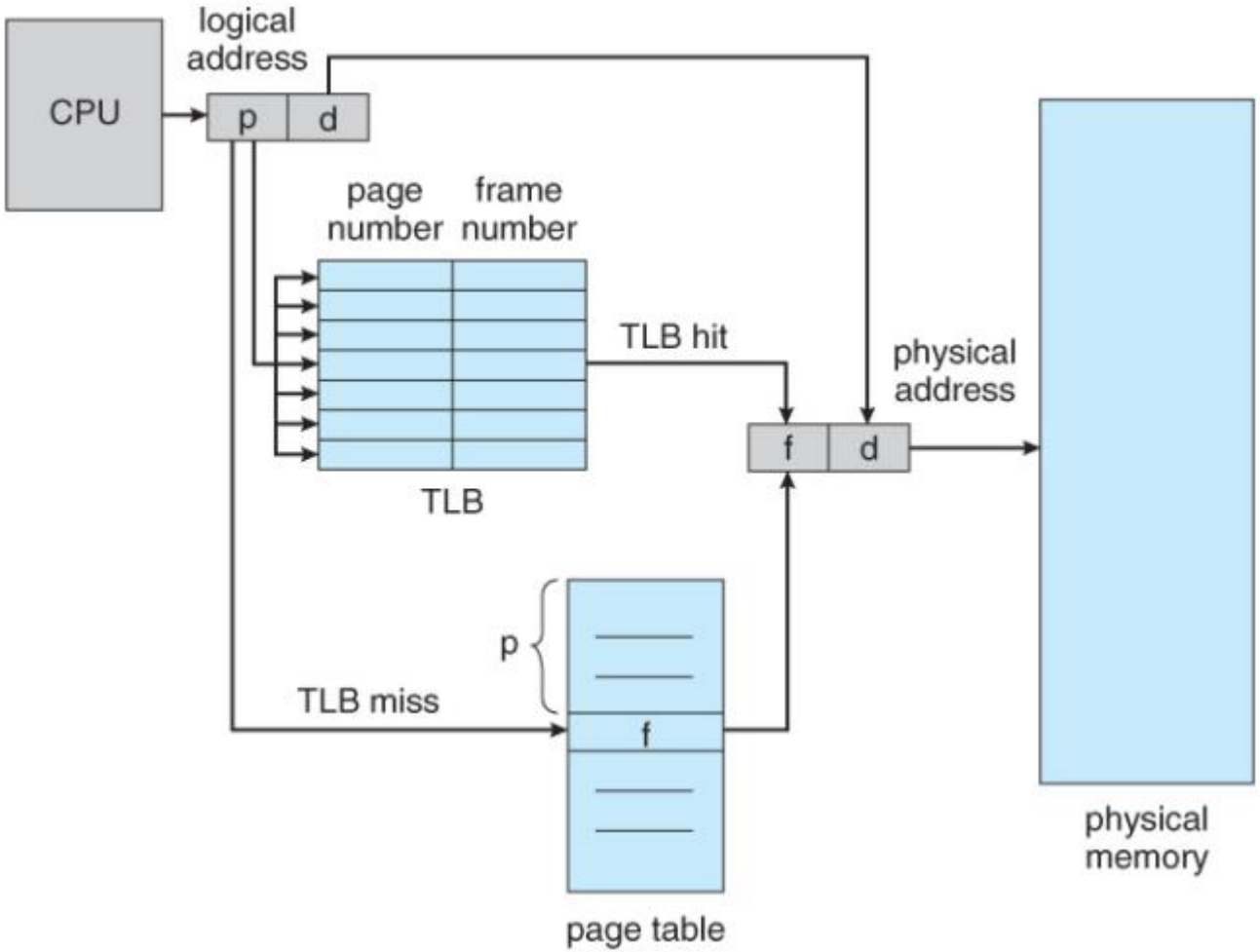


Figure 4-20 :TLB (Translation Look Aside Buffer)

- The CPU generates the logical address, which contains the page number and the page offset.
- The page number is used to index into the page table, to get the corresponding page frame number, and once we have the page frame of the physical memory(also called main memory), we can apply the page offset to get the right word of memory.
- The thing is that page table is stored in physical memory, and sometimes can be very large, **so to speed up the translation of logical address to physical address** , we use TLB, **which is made of expensive and faster associative memory**, So instead of going into page table first, we go into the TLB and use page number to index into the TLB.
- **TLB Miss:** If we don't find the page frame number inside the TLB, it is called a TLB miss only then we go to the page table to look for the corresponding page frame number.
- **TLB Hit:** If we find the page frame number in TLB, its called TLB hit, and we don't need to go to page table.

- **Page Fault**: Occurs when the page accessed by a running program is not present in physical memory. It means the page is present in the secondary memory but not yet loaded into a frame of physical memory.
- **Cache Hit**: Cache Memory is a small memory that operates at a faster speed than physical memory and we always go to cache before we go to physical memory. If we are able to locate the corresponding word in cache memory inside the cache, its called cache hit and we don't even need to go to the physical memory.
- **Cache Miss:** It is only after when mapping to cache memory is unable to find the corresponding block(block similar to physical memory page frame) of memory inside cache (called cache miss), then we go to physical memory and do all that process of going through page table or TLB.

Page Replacement Algorithms

- First-in First-Out (FIFO) page replacement
- Second chance page replacement
- Optimal page replacement
- Least Recently Used (LRU) page replacement
- Least Frequently Used (LFU) page replacement
- Clock page replacement
- WSClock page replacement
- Not recently used (NRU) page replacement

- If a process requests for page and that page is found in the main memory then it is called **page hit**, otherwise **page miss** or **page fault**.

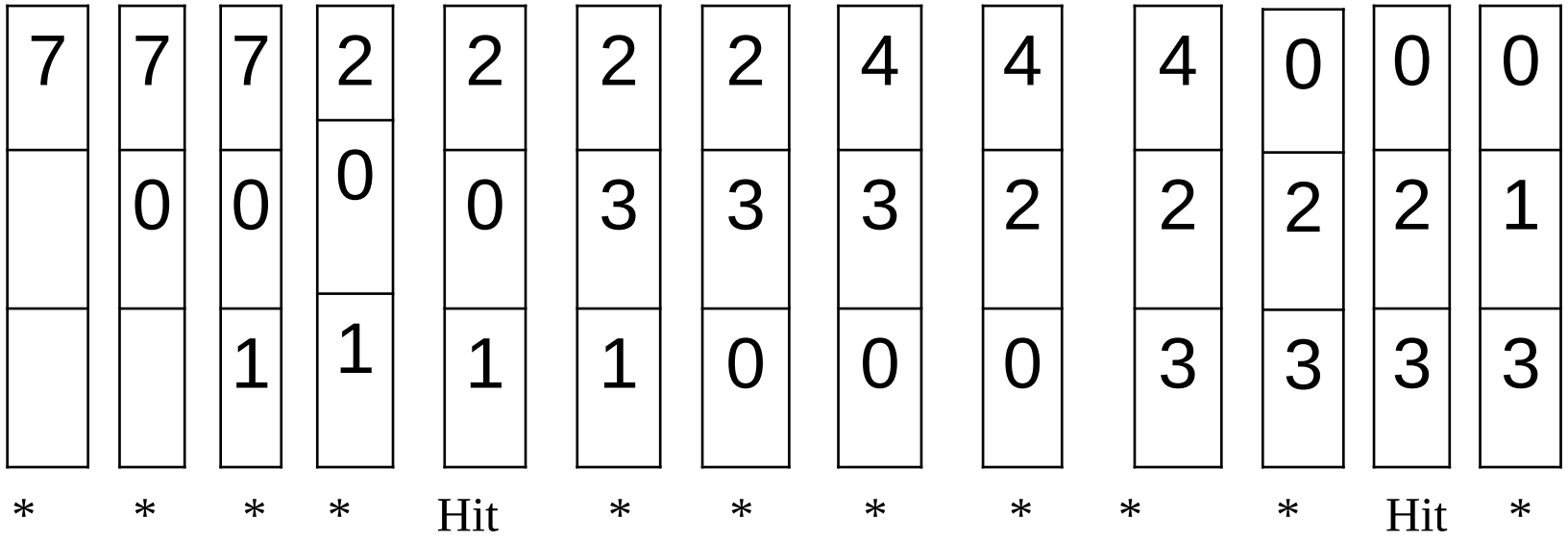
First-in, first-out

- The simplest page-replacement algorithm is a FIFO algorithm.
- The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book-keeping on the part of the operating system.
- The idea is obvious from the name - the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front.
- When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected.

  –While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences **Belady's anomaly**, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur.

Example:- Consider the main memory consists of 3 frames for FIFO Page Replacement algorithm and input pattern of memory pages.

7, 0 , 1 , 2 , 0 , 3, 0, 4 , 2 , 3 , 0, 3, 1



Total references=13

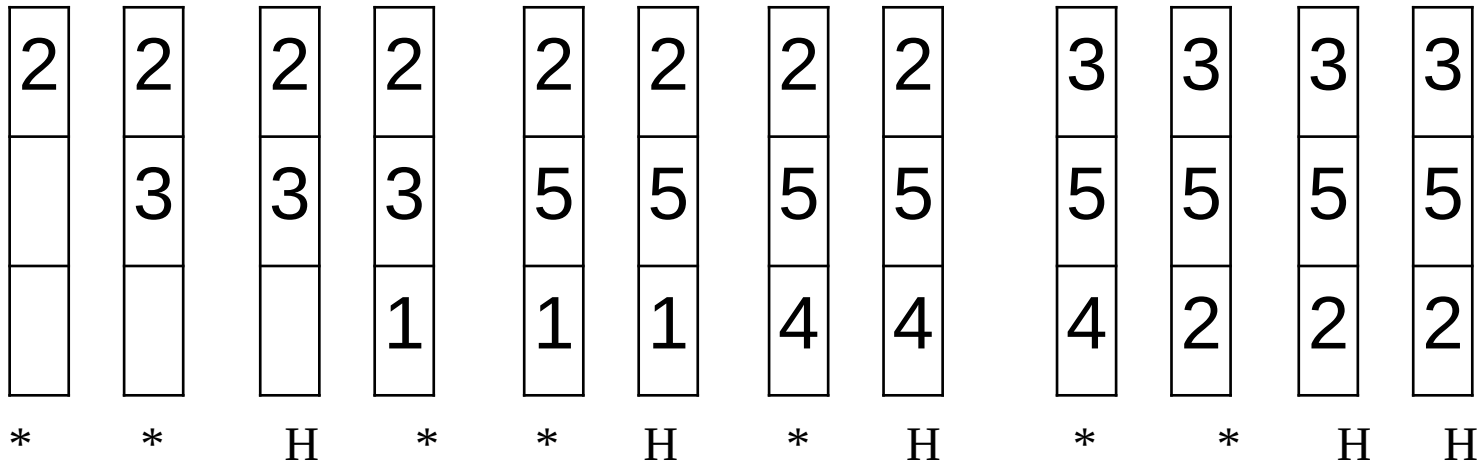
Total Page Hits=2 , [Hit ratio= no. of hits / no. of references=>2/13]

Total Page faults/page miss=11 , [Miss ratio= no. of miss / no. of references=>11/13]

Second Chance

Example:- Consider the main memory consists of 3 frames for Second chance Page Replacement algorithm and input pattern of memory pages.

2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 3, 5



Total Page faults= 7

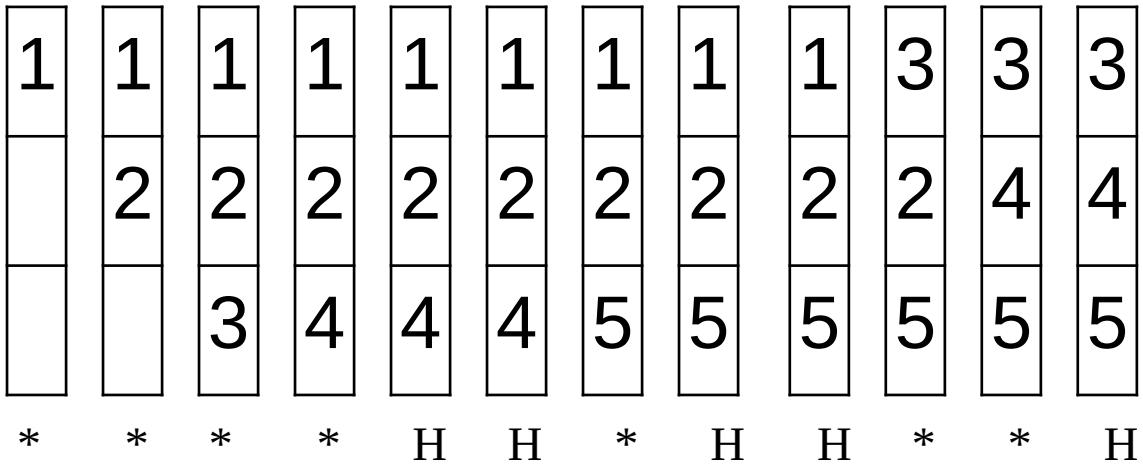
Total Page hits=5

Optimal page replacement(OPT)

- The theoretically optimal page replacement algorithm (also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy) is an algorithm that works as follows:
  - –when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future.
  - –For example, a page that is not going to be used for the next 6 seconds will be swapped out over a page that is going to be used within the next 0.4 seconds.
- This algorithm cannot be implemented in the general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to the static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis.

Example:- Consider the main memory consists of 3 frames for OPT Page Replacement algorithm and input pattern of memory pages.

1,2,3,4,1,2,5,1,2,3,4,5



Total page faults=7

Total page Hits=5

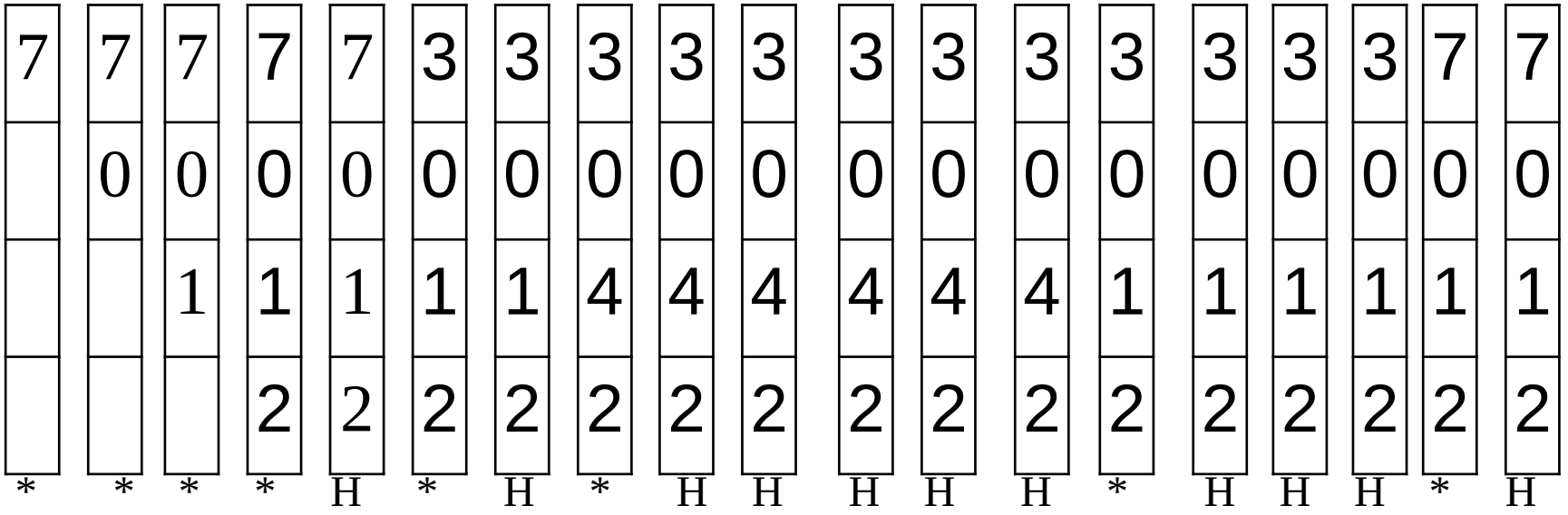
Least Recently Used (LRU)

- LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too.
- While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice.
- There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.
- The most expensive method is the linked list method, which uses a linked list containing all the pages in memory.
- At the back of this list is the least recently used page, and at the front is the most recently used page.

- The difficulty is that the list must be updated on every memory reference.
- Another method that requires hardware support is as follows: suppose the hardware has a 64-bit counter that is incremented at every instruction.
- Whenever a page is accessed, it gains a value equal to the counter at the time of page access.
- Whenever a page needs to be replaced, the operating system selects the page with the lowest counter and swaps it out.
- With present hardware, this is not feasible because the OS needs to examine the counter for every page in memory.

Example:- Consider the main memory consists of 4 frames for LRU Page Replacement algorithm and input pattern of memory pages.

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0



Page Faults=8

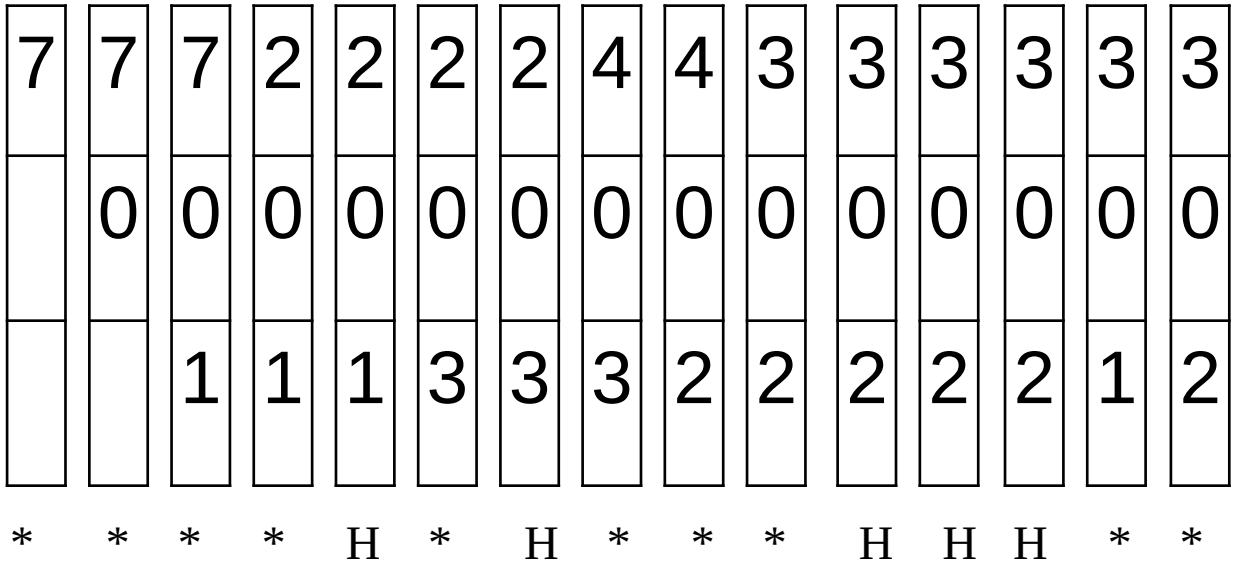
Page Hits=11

Least Frequently Used (LFU)

- Least Frequently Used (LFU) is a type of cache algorithm used to manage memory within a computer.
- The standard characteristics of this method involve the system keeping track of the number of times a block is referenced in memory.
- When the cache is full and requires more room the system will purge the item with the lowest reference frequency.

Example (using 3 frames):- Reference String is:

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2

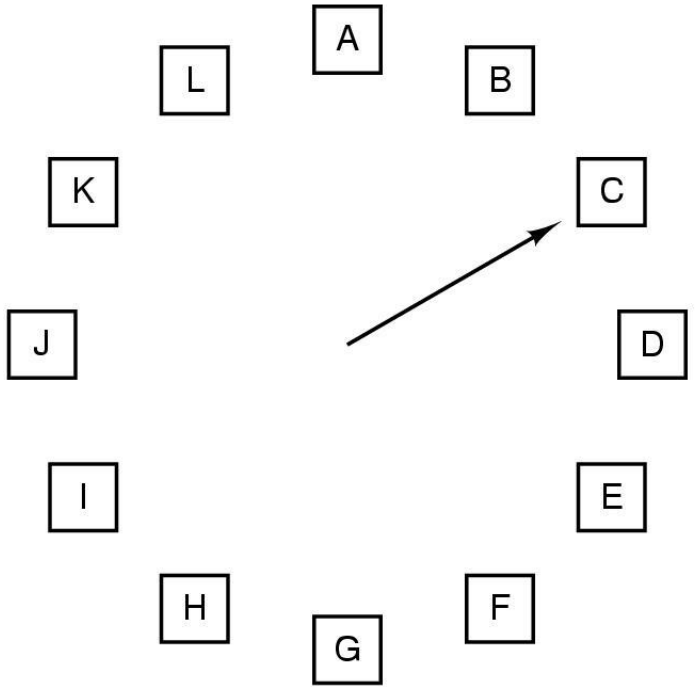


Total page Hits=5

Total page Faults=10

Clock

- keep all the page frames on a circular list in the form of a clock



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

- When a page fault occurs, the page being pointed to by the hand is inspected.
- If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.
- If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with *R* = 0. Not surprisingly, this algorithm is called **clock**.
- Doesn’t use age as a reason to evict page
- Faster-doesn’t manipulate a list
- Doesn’t distinguish between how long pages have not been referenced

The Working Set Model

- The working set of a process is the set of pages expected to be used by that process during some time interval.
- The "working set model" isn't a page replacement algorithm in the strict sense (it's actually a kind of medium-term scheduler)
- Working set is a concept in computer science which defines what memory a process requires in a given time interval.
- **Demand paging**-bring a process into memory by trying to execute first instruction and getting page fault. Continue until all pages that process needs to run are in memory (the working set)
- Try to make sure that working set is in memory before letting process run (pre-paging)
- **Thrashing-**memory is too small to contain working set, so page fault all of the time.

- **Locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multi-pass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

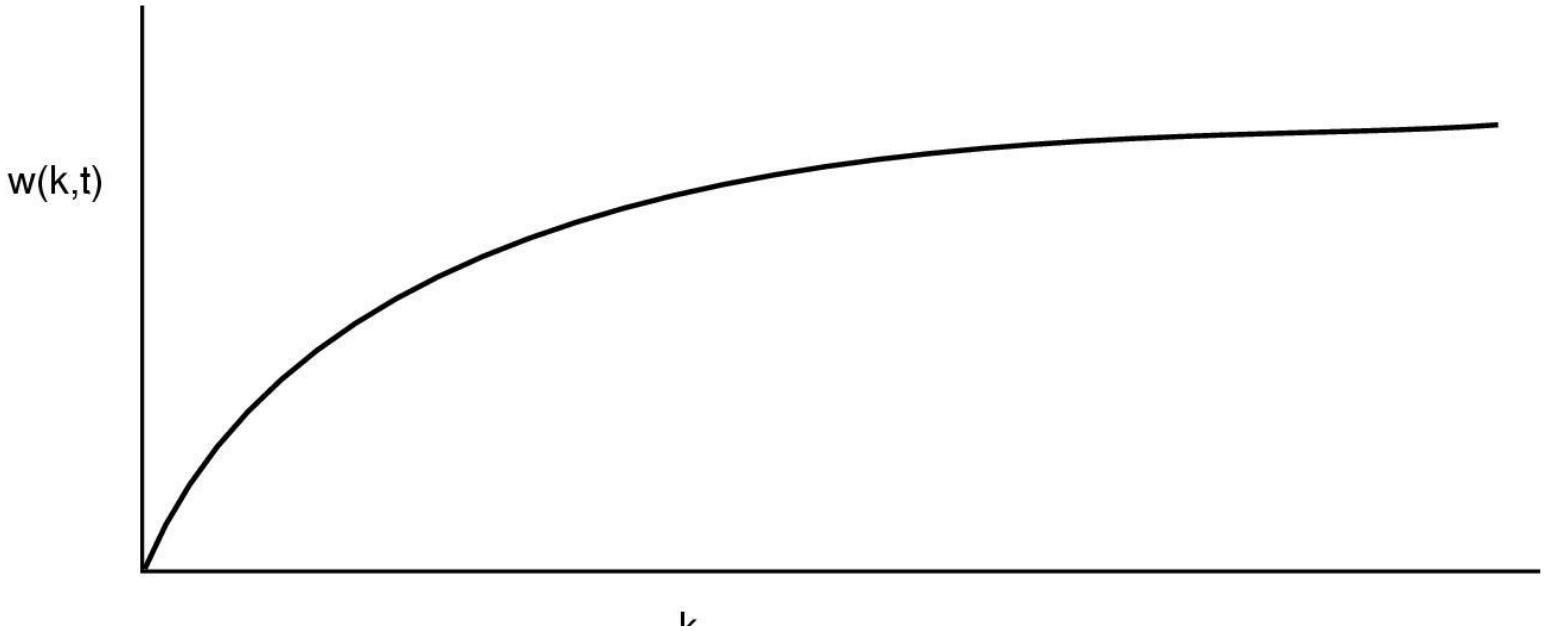


Figure 4-21: The working set is the set of pages used by the *k* most recent memory references. The function *w*(*k*, *t*) is the size of the working set at time *t*.

How to implement working set model

- When fault occurs can evict page not in working set (if there is such a page)
- Need to pick k
- Could keep track of pages in memory at every memory reference. Each k references results in a working set.
- Shift register implementation. Insert page number at each reference.
- Expensive

Working Set Page Replacement

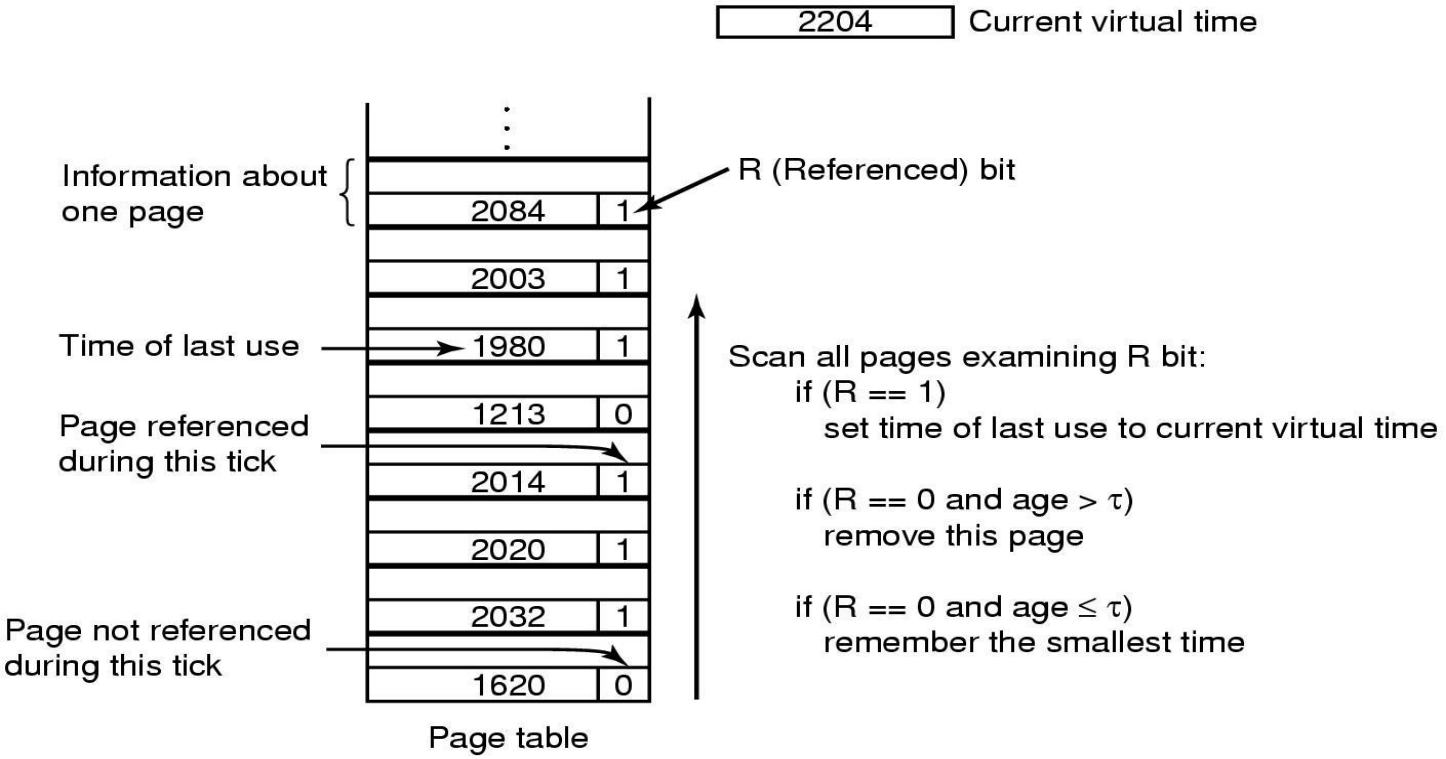


Figure 4-22. The working set algorithm.

Weakness with WS algorithm

- Need to scan entire page table at each page fault to find a victim
- Use clock idea with working set algorithm

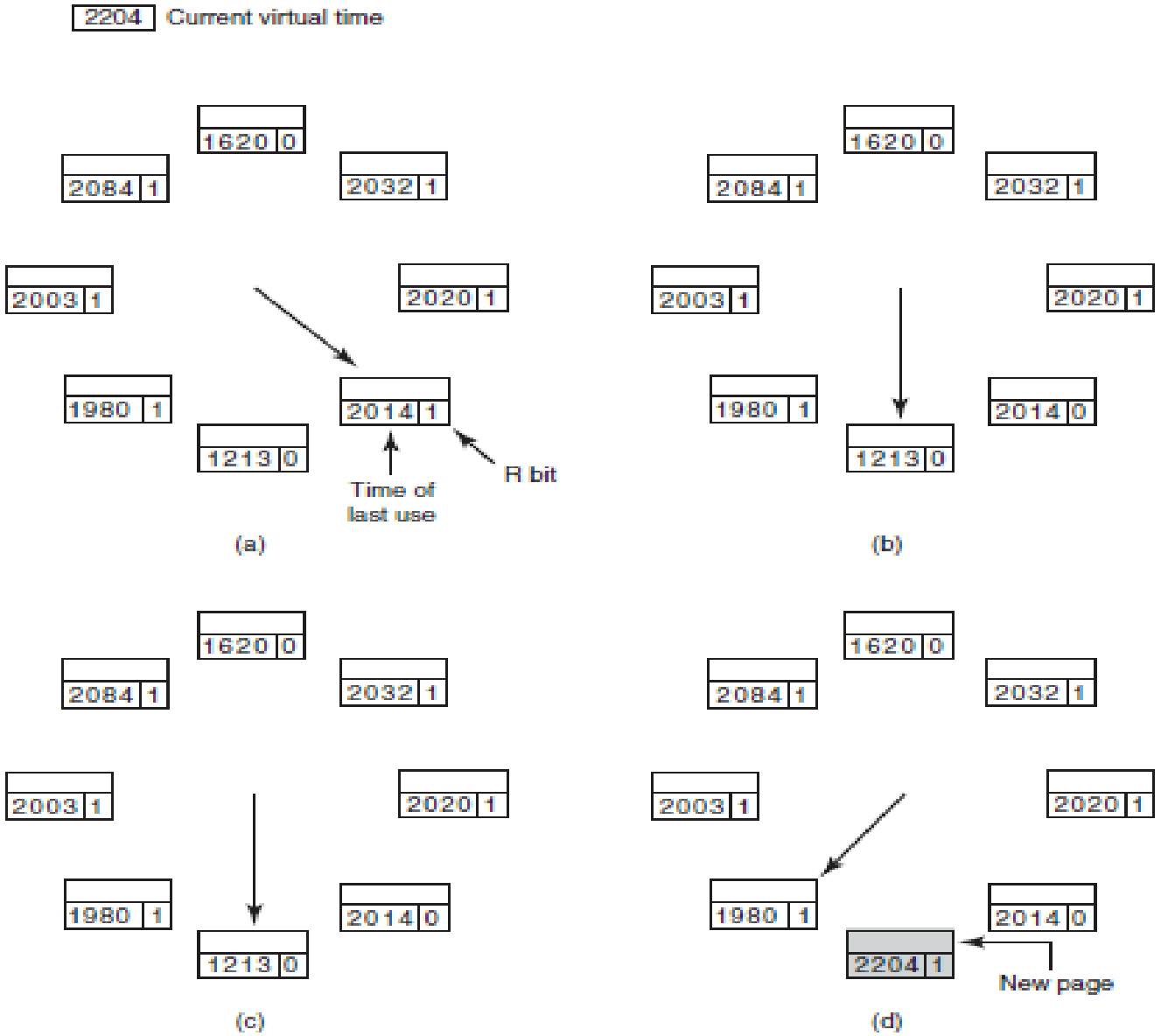


Figure 4-23: Operation of the WSClock algorithm. (a) and (b) give an example of what happens when *R* = 1. (c) and (d) give an example of *R* = 0.

The WSClock Page Replacement Algorithm

- If the hand comes all the way around to its starting point there are two cases to consider:

- At least one write has been scheduled.
  - Hand keeps moving looking for clean page. Finds it because a write eventually completes- evicts first clean page hand comes to.
- No writes have been scheduled.
  - Evict first (clean) page

Summary of Page Replacement Algorithms

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

Belady's Anomaly

- Belady's Anomaly is the phenomenon of increasing the number of page faults on increasing the number of frames in main memory.
- Following page replacement algorithms suffer from Belady's Anomaly.
  - FIFO Page Replacement Algorithm
  - Random Page Replacement Algorithm
  - Second Chance Algorithm
- "Algorithms suffer from Belady's Anomaly" does not mean that always the number of page faults will increase on increasing the number of frames in main memory.
- This unusual behavior is observed only sometimes.

Example: Consider FIFO page replacement

- Reference string :0 1 5 3 0 1 4 0 1 5 3 4 ,Case 1: 3 frames available

Case 2: 4 frames available

Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 3			5	5	5	1	1	1	1	1	3	3
Frame 2		1	1	1	0	0	0	0	0	5	5	5
Frame 1	0	0	0	3	3	3	4	4	4	4	4	4
Miss/Hit	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

Number of Page Faults = 9

Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 4				3	3	3	3	3	3	5	5	5
Frame 3			5	5	5	5	5	5	1	1	1	1
Frame 2		1	1	1	1	1	1	0	0	0	0	4
Frame 1	0	0	0	0	0	0	4	4	4	4	3	3
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

Number of Page Faults = 10

Therefore, in this example, the number of page faults is increasing by increasing the number of frames hence this suffers from Belady’s Anomaly.

Segmentation

- **Segmentation** is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process.
- The details about each segment are stored in a table called as segment table.
- For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

- Consider what happens if a program has an exceptionally large number of variables but a normal amount of everything else.
- The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables.
- A straightforward and extremely general solution is to provide the machine with many completely independent address spaces, called segments.
- Each segment consists of a linear sequence of addresses, from 0 to some maximum.
- Segment table contains mainly two information about segment:

  Base: It is the base address of the segment

  Limit: It is the length of the segment.

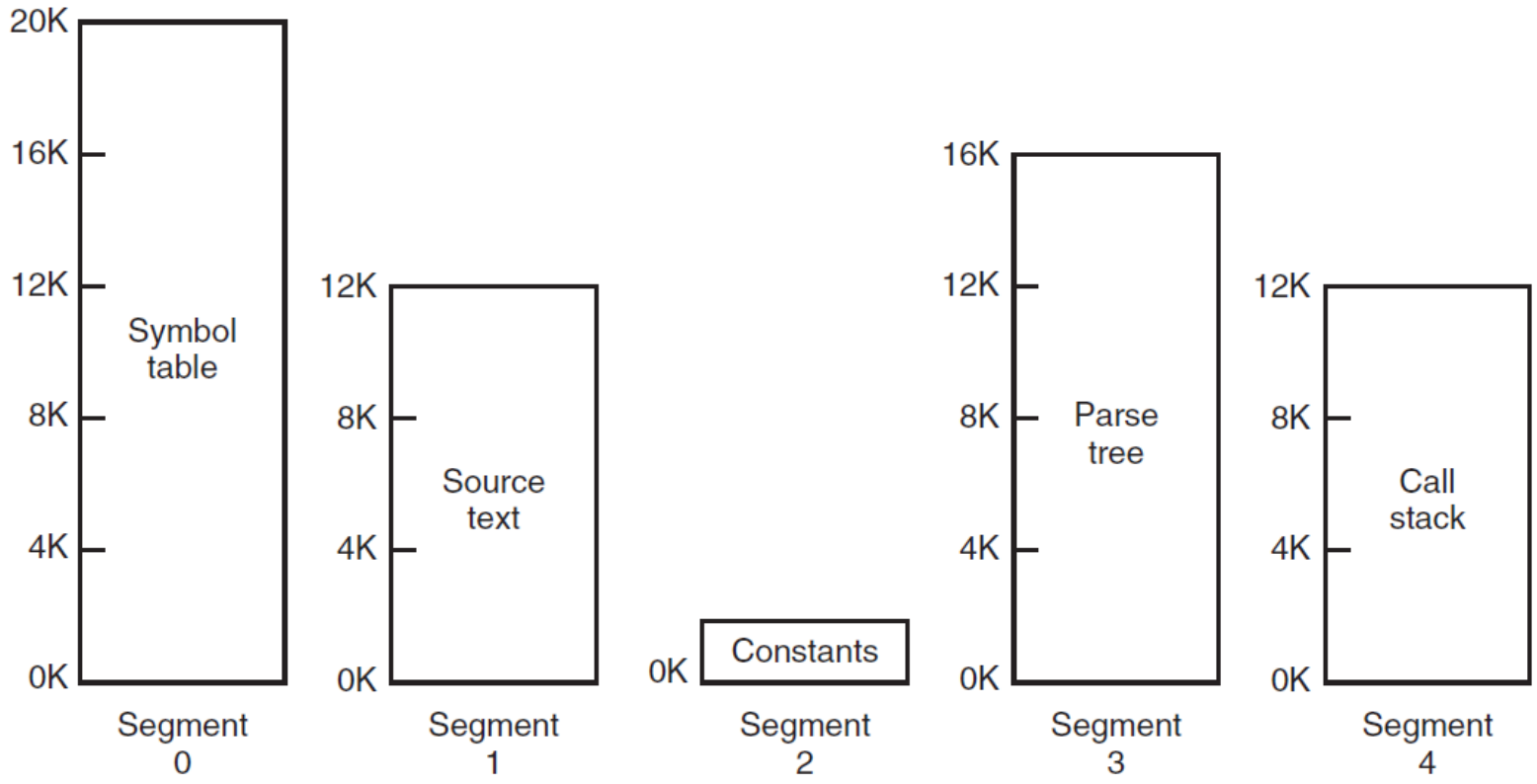


Figure 4-24: A segmented memory allows each table to grow or shrink independently of the other tables.

Advantages of Segmentation

- Simplifies handling of data structures which are growing and shrinking, Address space of segment n is of form (n, local address) where (n,0) is starting address
- Can compile segments separately from other segments
- Segmentation also facilitates sharing procedures or data between several processes. A common example is the shared library.
- Can have different protections (r,w,x) for different segments
- The segment table is of lesser size as compare to the page table in 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

Figure 4-25: Comparison of paging and segmentation.

Implementation of Pure Segmentation

- The implementation of segmentation differs from paging in an essential way: pages are of fixed size and segments are not.
- Figure (a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place.
- We arrive at the memory configuration of Fig.(b). Between segment 7 and segment 2 is an unused area—that is, a hole.
- Then segment 4 is replaced by segment 5, as in Fig.(c), and segment 3 is replaced by segment 6, as in Fig.(d).
- After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig.(e).

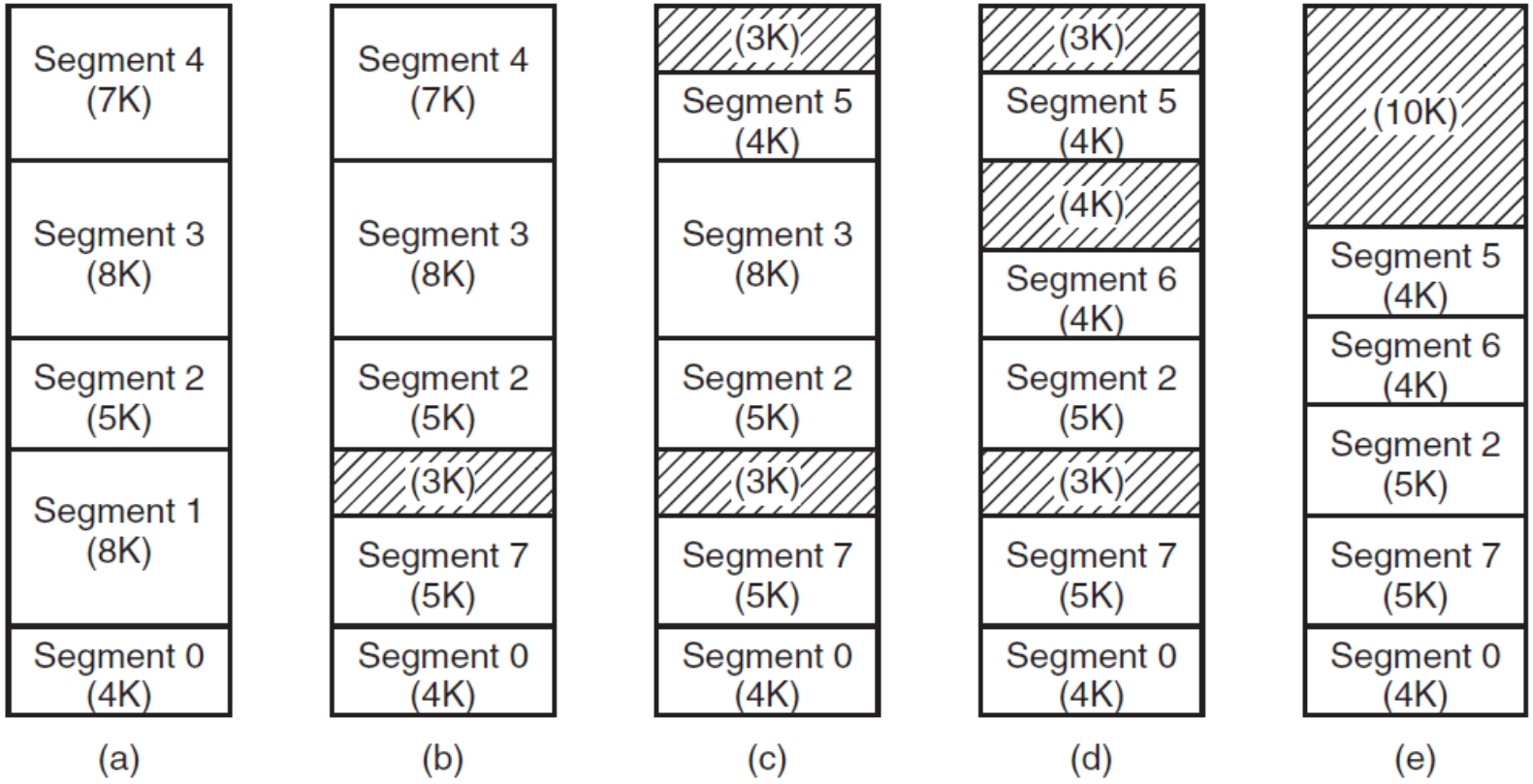


Figure 4-26 : (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging: MULTICS

- Multics --IBM mainframe systems. A pioneer of introducing segment to modern architecture.
- Every process can have multiple virtual address spaces (or segments)
- Each Segment has its own page table

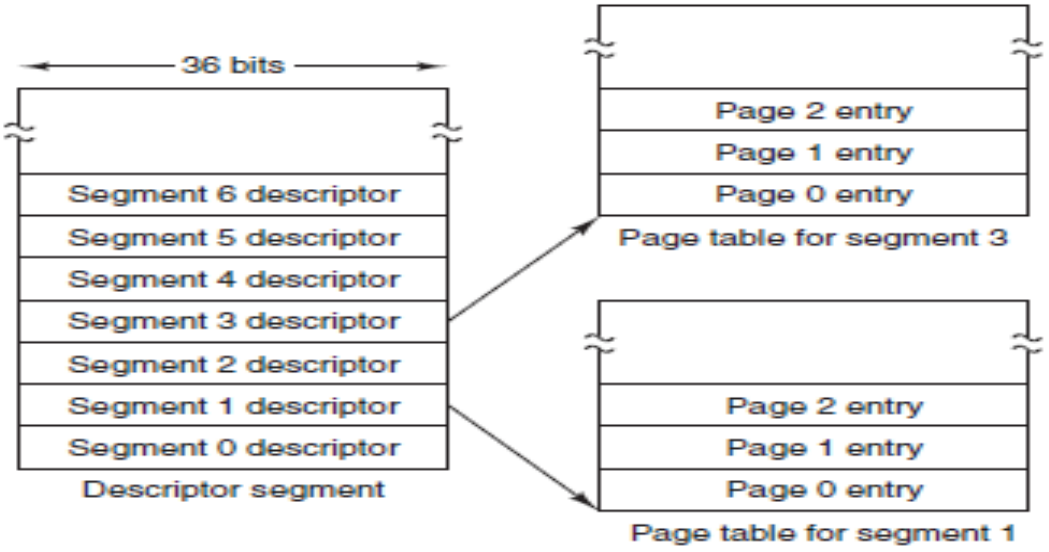
Advantage

- Each segment can have the full virtual address space allowed by number of address bits

Disadvantage

- Switching from one segment to another has a high context switch penalty, even within the same process.

Segmentation with Paging: MULTICS



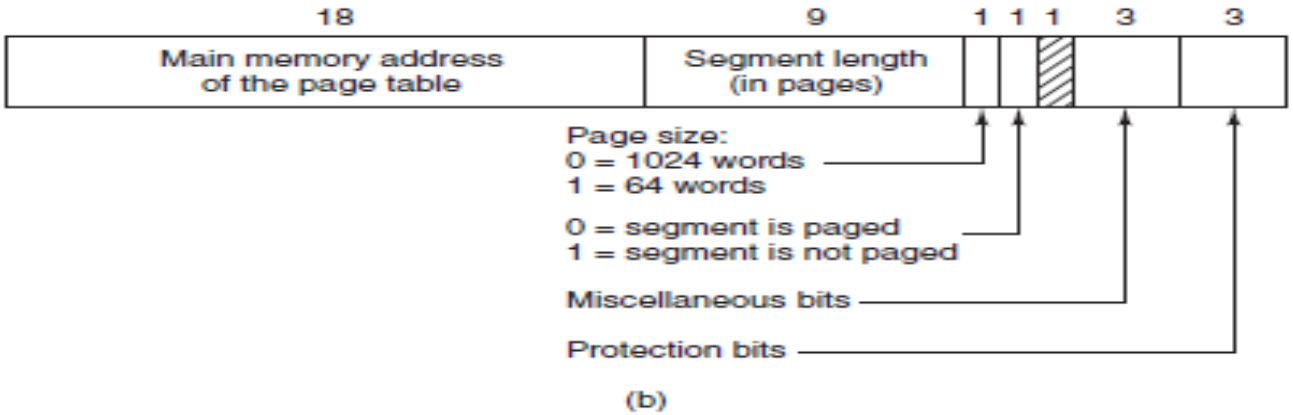


Figure 4-27: The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables. (b) A segment descriptor. The numbers are the field lengths.

- Each MULTICS program had a segment table, with one descriptor per segment.
- A segment descriptor contained an indication of whether the segment was in main memory or not. If any part of the segment was in memory, the segment was considered to be in memory, and its page table was in memory.
- If the segment was in memory, its descriptor contained an 18-bit pointer to its page table, as in Fig. (a).
- The descriptor also contained the segment size, the protection bits, and other items.
- Figure (b) illustrates a segment descriptor. The address of the segment in secondary memory was not in the segment descriptor but in another table used by the segment fault handler.
- Each segment descriptor points to a page table

- An address in MULTICS consisted of two parts: the segment and the address within the segment. The address within the segment was further divided into a page number and a word within the page.

  Segment Number : It points to the appropriate Segment Number.

  Page Number: It Points to the exact page within the segment

  Page Offset: Used as an offset within the page frame

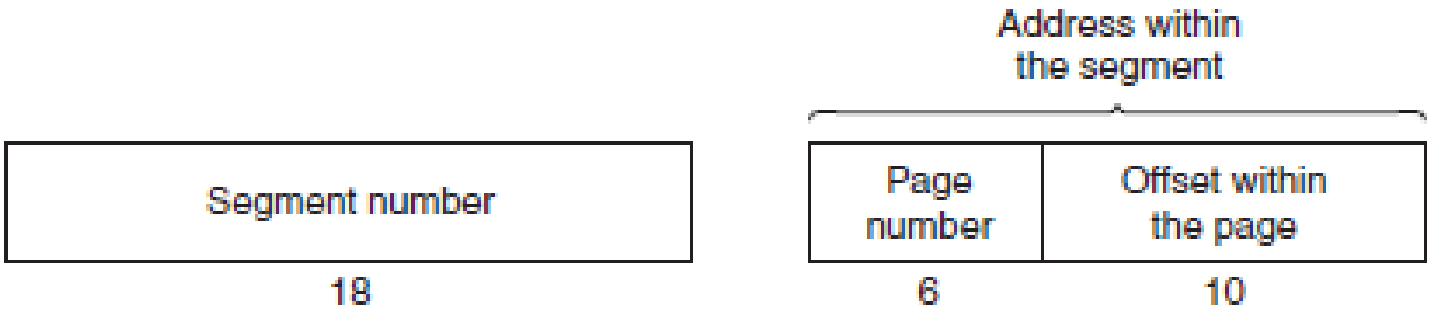


Figure 4-28: A 34-bit MULTICS virtual address.

Segmentation with Paging: The Intel x86

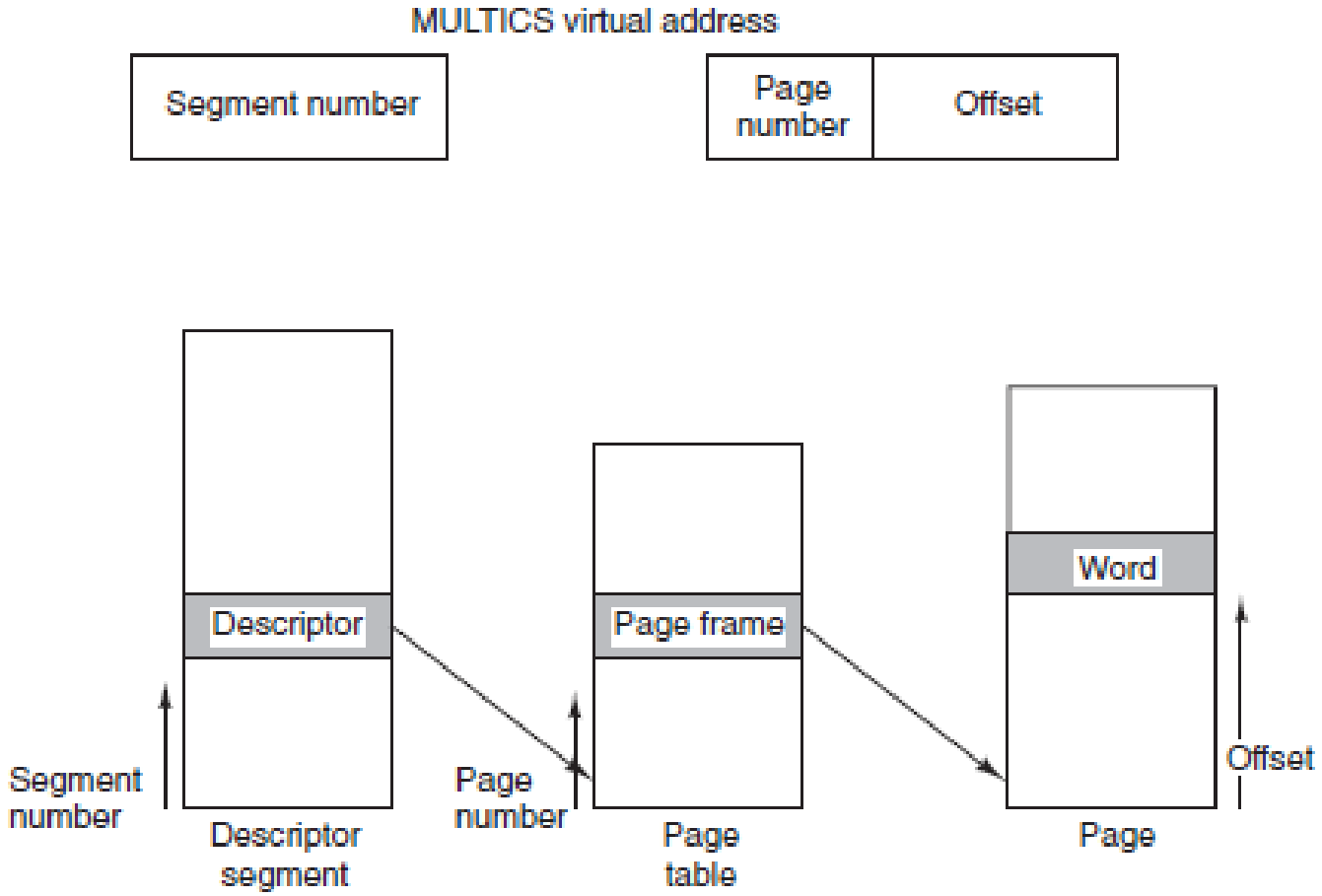


Figure 4-29: Conversion of a two-part MULTICS address into a main memory address.

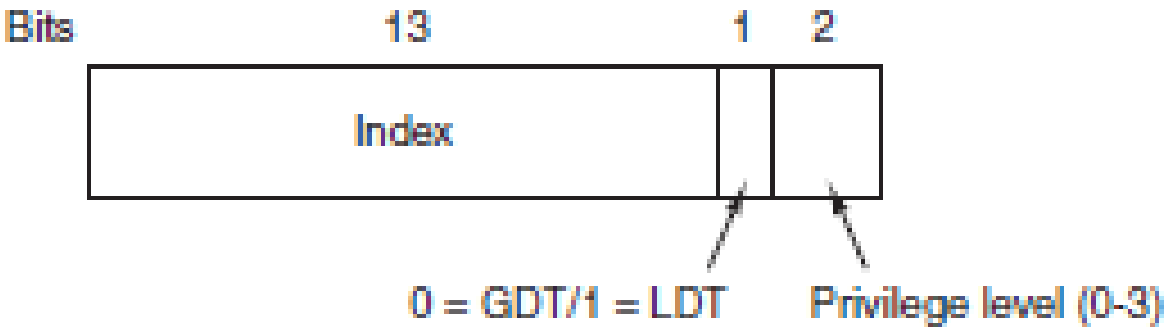


Figure 4-30: An x86 selector.

- The x86 virtual memory consists of two tables, called the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**).
- Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The
- LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

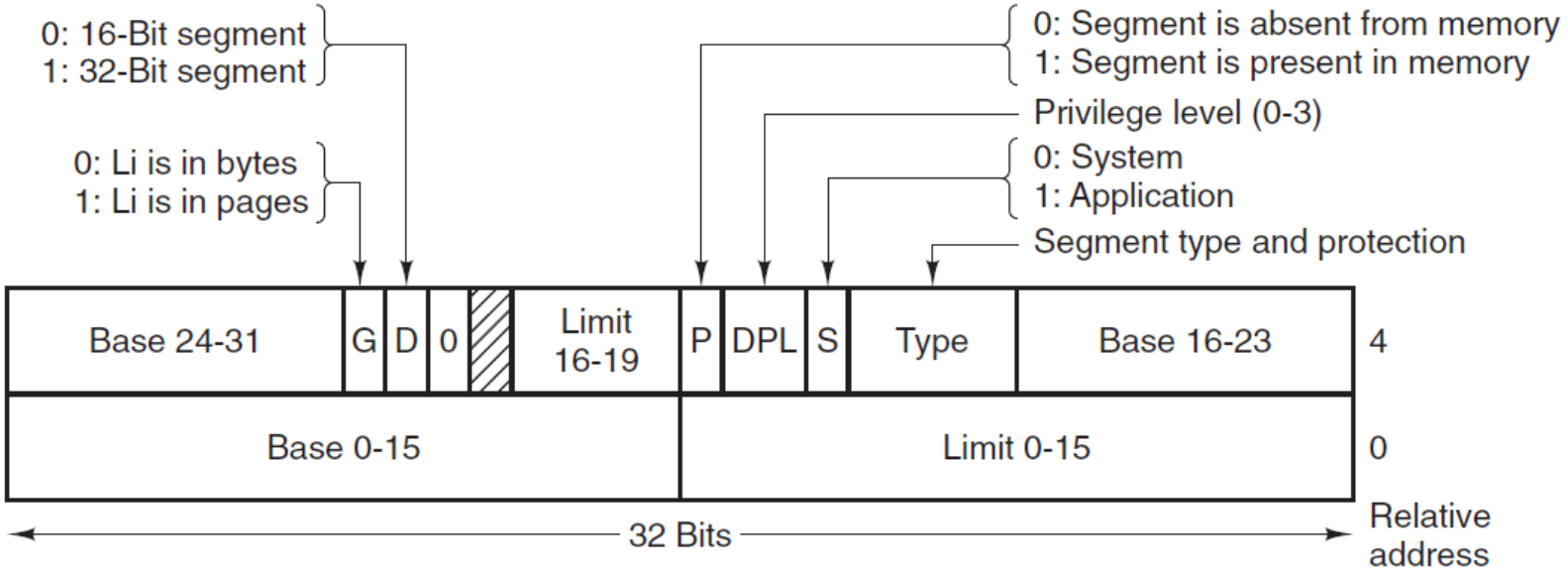


Figure 4-31: x86 code segment descriptor. Data segments differ slightly.

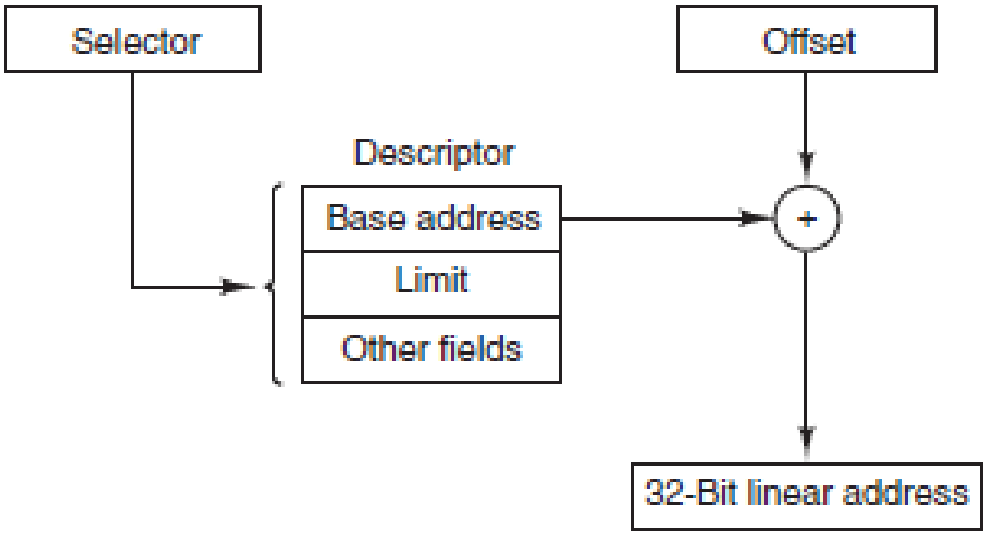


Figure 4-32: Conversion of a (selector, offset) pair to a linear address

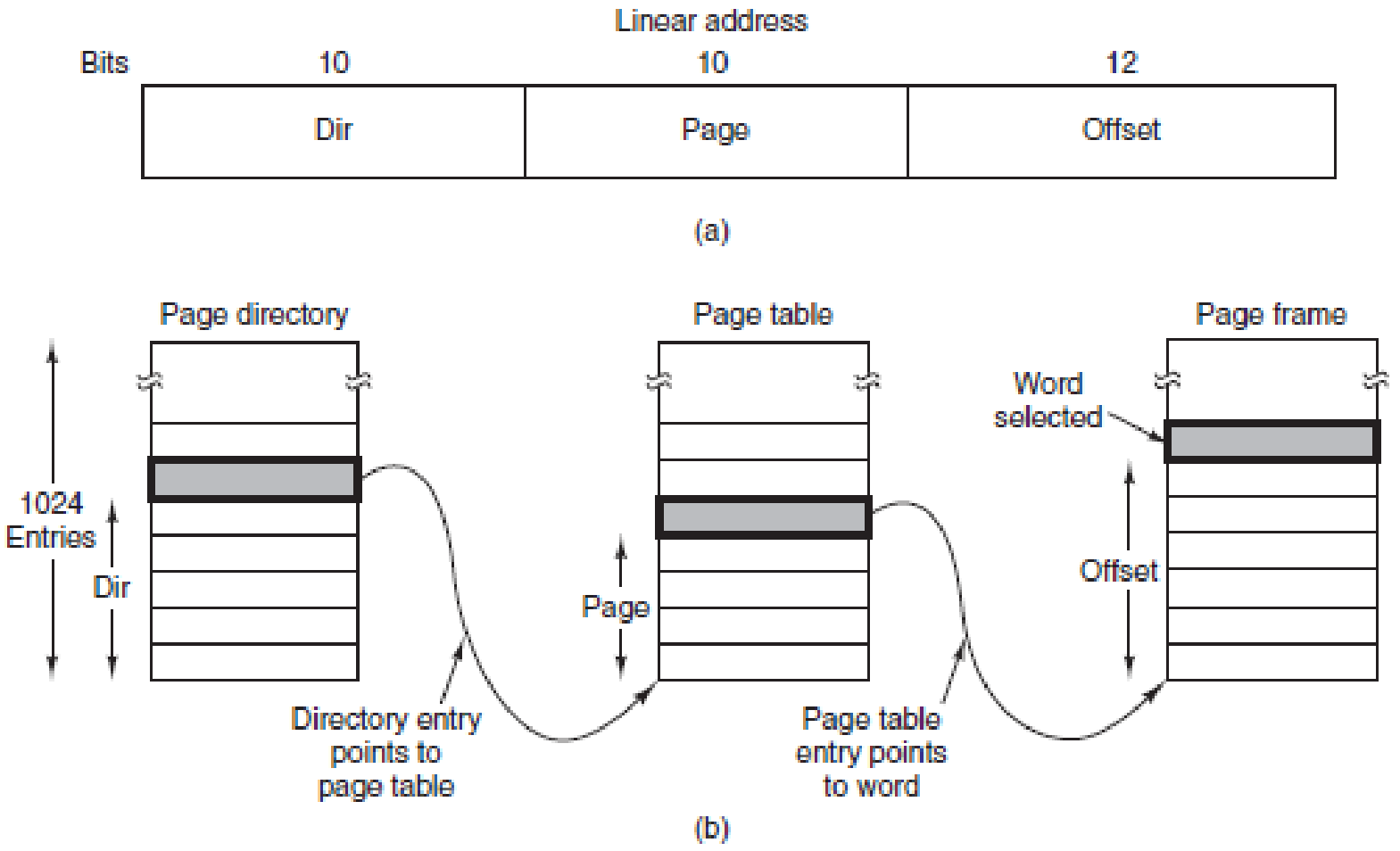


Figure 4-33: Mapping of a linear address onto a physical address.

- In fig.(A) linear address divided into three fields, dir, page, and offset.
- The **dir** field is used to index into the page directory to locate a pointer to the proper page table.
- The **page** field is used as an index into the page table to find the physical address of the page frame.
- The **offset** is added to the address of the page frame to get the physical address of the byte or word needed.