OPERATING SYSTEMS

#8

After A.S.Tanenbaum, Modern Operating Systems
3rd edition
Uses content with permission from Assoc. Prof. Florin Fortis, PhD

MEMORY MANAGEMENT

MEMORY MANAGEMENT

- The memory is one of the most important resources of a computing system.
  - Its management must be realized with caution: the memory is very important for the good functioning of the entire operating/computing system.

- Each computing system is able to offer several categories of memory devices (see the hierarchy of memory devices):
  - A small amount of cache memory;
  - A large amount of (RAM) memory, usually called the main memory;
  - A significant amount of long-term, non-volatile memory, like disk units.

MEMORY MANAGEMENT

- The main task of the memory manager is to… manage the hierarchy of memory devices, by:
  - Accounting memory usage;
  - Allocating memory to processes;
  - Getting back (released) memory from processes;
  - Managing the basic mechanisms, like swapping, whenever necessary.

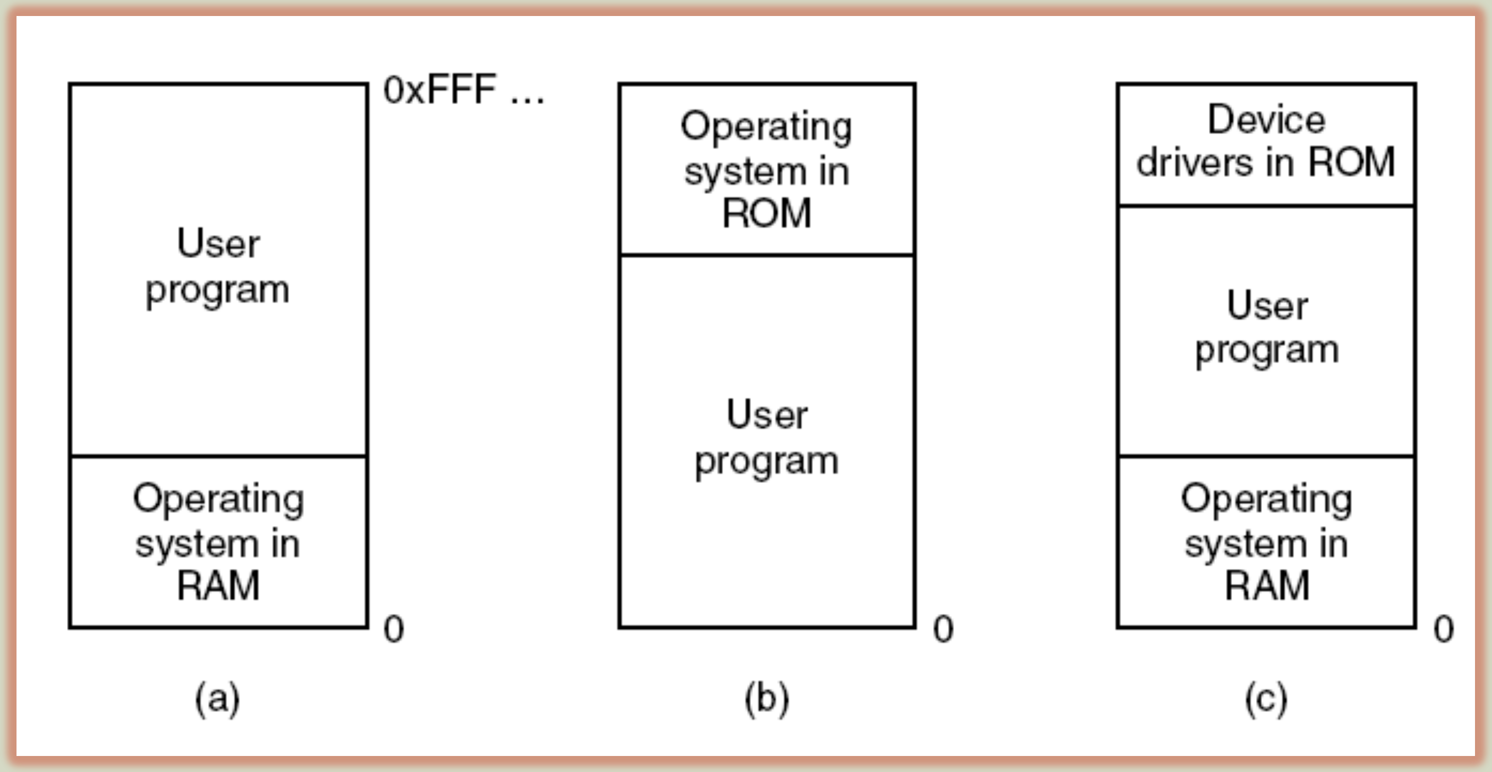
BASIC MEMORY MANAGEMENT

- There are two major categories of memory management systems:
  - Systems that are based on a permanent switch of processes between main memory and disc:
    - Systems based on *paging* or *swapping*.
  - Simple systems, without any process switch (see Figure, next slide).

MONO-PROGRAMMING

W/O swapping or paging

MONO-PROGRAMMING, WITHOUT SWAPPING OR PAGING



Three simple ways of organizing memory with an operating system and one user process.

MONO-PROGRAMMING, WITHOUT SWAPPING OR PAGING

- It is based on the idea that there is a unique program in memory at a time, together with the (core of the) operating system;
- There were three basic schemas for mono-programming memory organization, mentioned on previous slide:
  a) Schema used for mainframe operating systems and mini-computers.
  b) Schema used for palmtop and embedded systems. It could be still in use.
  c) Schema used for micro-computers.
     a) Notice the special part of (ROM) memory: the “Basic Input-Output System”.

MULTIPROGRAMMING

With fixed partitions

MULTI-PROGRAMMING WITH FIXED PARTITIONS

- It is characteristic for modern operating systems.
- A very simple method based on dividing the main memory in several partitions, possibly of the same size.
- Two situations are possible

1. A new job is put in the queue of a partition big enough to support the job.
2. All jobs are put in the same queue

MULTI-PROGRAMMING WITH FIXED PARTITIONS

1. A new job is put in the queue of a partition big enough to support the job.
   - In this situation, the unused space is considered to be lost.
   - In this method the jobs are sorted on their size and there are several queues used to hold the jobs.

- E.g. A system that is going to extensively use short sized jobs is going to keep big partitions unused most of the time.
  - It could be possible to have short size jobs waiting in the system, even if there is enough free space to satisfy these jobs!

MULTI-PROGRAMMING WITH FIXED PARTITIONS

2. All jobs are put in the same queue

- Now it is possible to avoid previous problems, but there is a high risk to have an inefficient usage of the existing partitions. There are several alternatives:
    1. When a partition is freed, this partition is offered to the job that best fits with the partition: it is possible to “forget” several small jobs;
    2. Permanently offer at least one small partition for small sized jobs.
    3. Offer a guarantee to processes: they are loaded in memory at least once during a fixed amount of time.

MODELING MULTI-PROGRAMMING

- By using multi-programming one could improve the level of processor usage;
  - E.g. If a process uses only 20% of computing resources, that one can think that 5 processes are going to use 100% ☺ However, this situation requires a perfect synchronization between these processes.

- In fact, in a system with n identical processes, the level of processor usage is at most 1-p^n, where p is the fraction of time used for input/output operations.
  - E.g. For our previous situation, the level of processor usage is, at most, 67%. At least 10 processes are required for a higher level of processor usage…

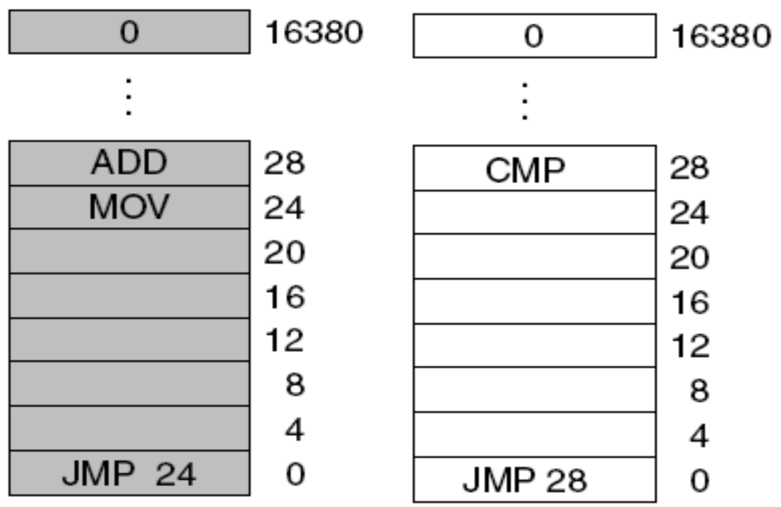
MEMORY MANAGEMENT

Protection and relocation

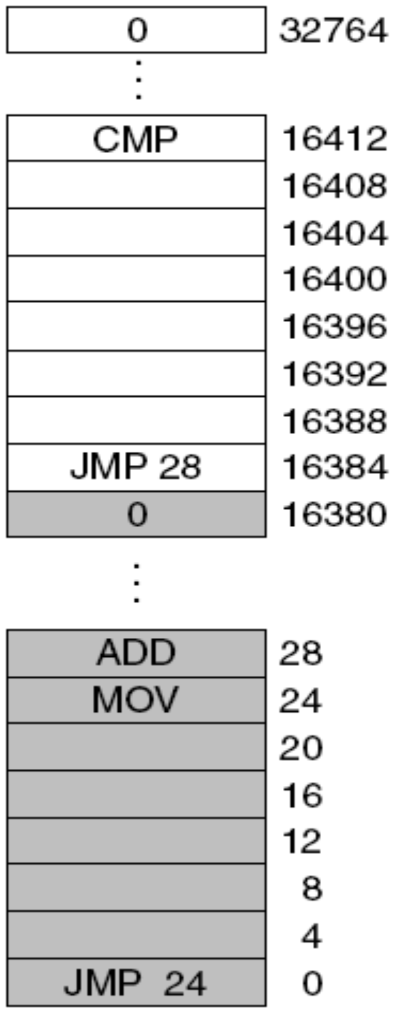
PROTECTION AND RELOCATION

- By multi-programming two major problems for memory management are identified:
  - memory protection and address relocation.
- A simple possibility to solve the relocation is to modify instructions while they are loaded in memory.
  - We cannot solve memory protection by using this rudimentary technique: user programs are able to generate new instructions and determine jumps to these instructions. (OS/MFT)

THE RELOCATION PROBLEM



(a) (b)



(c)

The relocation problem:

a) A 16 KB program
b) Another 16 KB program
c) The two programs loaded simultaneously in memory!

PROTECTION AND RELOCATION

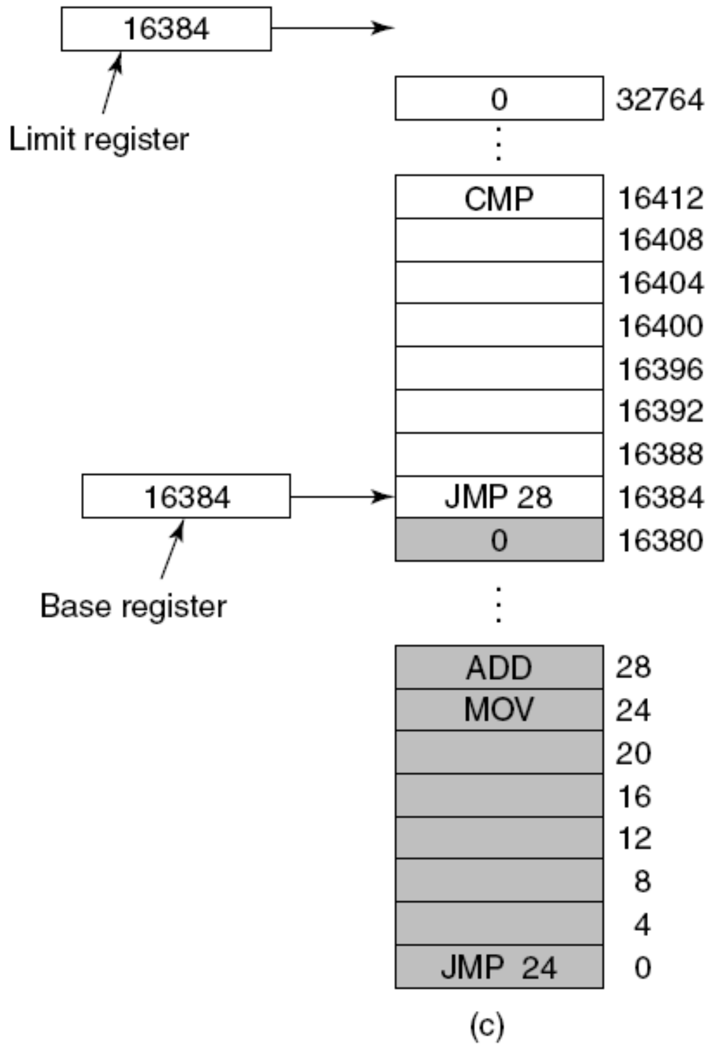
- A second variant makes the presumption that the memory is divided into small blocks of 2K, each block being protected by a 4-bits code, which is stored in the PSW.
  - This mechanism is able to offer protection since the only application able to modify protection codes is the operating system itself.
- Any attempt to access a (memory) block with a different protection code is promptly sanctioned by issuing a *trap*. (typical for OS360)

PROTECTION AND RELOCATION

- The solution for these two problems, widely used on Intel 8088 based systems, or CDC 6600 based systems, uses a pair of registers: the base register and the limit register (see also introductory lectures)
- Addresses are verified against the two values stored in the registers
  - The only application able to modify these registers is the operating system.
  - More, there is a hardware protection mechanism for these the two registers.
- This solution has, however, a major disadvantage: each memory reference requires several compares and additions.
  - The solution is acceptable for operating systems running on micro-computers.

PROTECTION AND RELOCATION

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



MEMORY MANAGEMENT

Swapping

SWAPPING

- For batch processing systems, a simple memory organization, based on fixed memory partitions, is enough to guarantee job execution and to maintain a high level of processor usage.
- For a time-shared system it is difficult to offer all the necessary resources to permanently store all processes in memory.
- For this kind of operating system, some of the existing jobs are temporarily stored on the disc, and reloaded into memory in a dynamic manner.

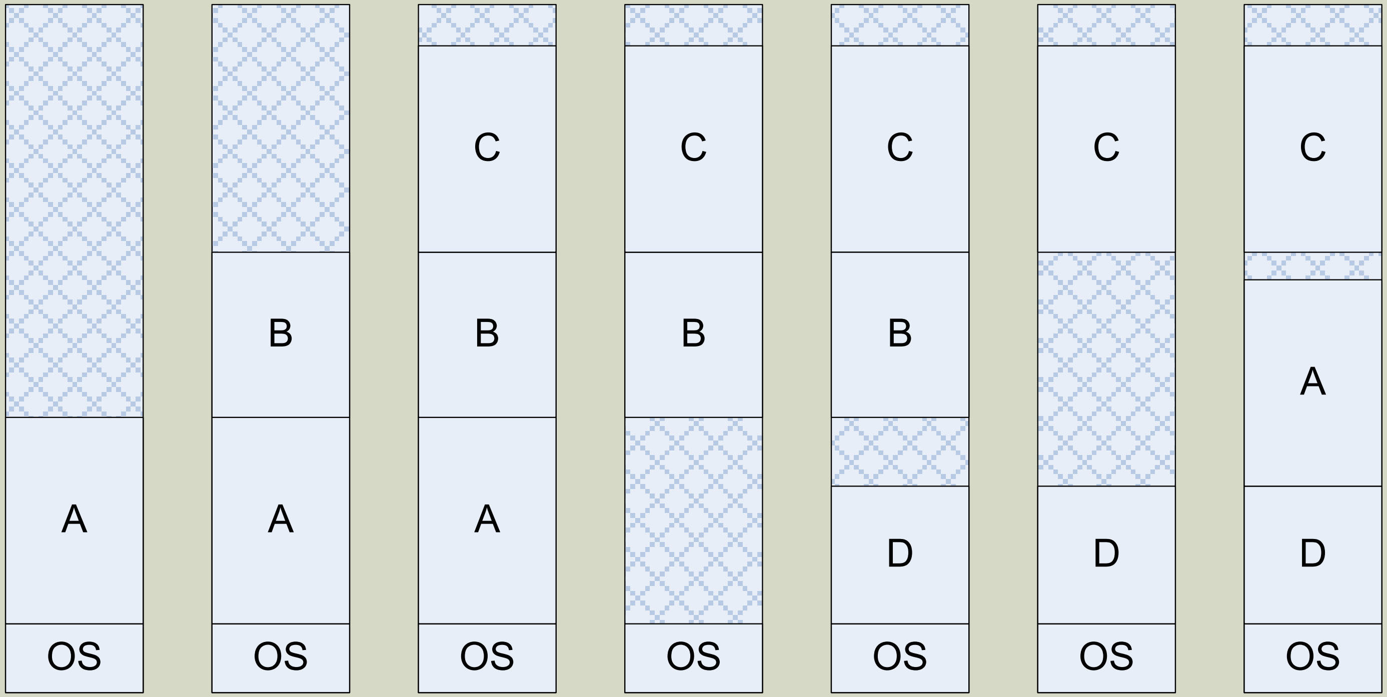
SWAPPING

- There are two major (and distinct) strategies:

1. The entire process is stored/reloaded into memory. This strategy is usually called *swapping*.
2. Processes can be executed even if they are partially loaded into memory. This time we are talking about *virtual memory*.

SWAPPING

- The swapping mechanism is quite simple, as shown in the picture below. Shaded regions are unused memory:



SWAPPING

- For swapping, the size, position and the number of partitions are dynamically modified, as (new) processes enter or leave the system.
- Observe that the problems related with partition sizes disappear.
  - However, there are some complications related with memory allocation and memory usage.
- Swapping technique can lead in time to several small holes in memory.
  - It is possible to have a situation when a program cannot be executed, even if there is enough (total) memory space.

SWAPPING

- In our initial example, after process D leaves the system, process B cannot enter the system, since the available contiguous space is to small, even if the total amount of available space is enough for this task!
  - Memory *compacting* techniques are used in order to transform a large amount of small memory holes into a small amount of large memory holes.
  - However, these techniques are time-consuming, so they are used only as a last solution.

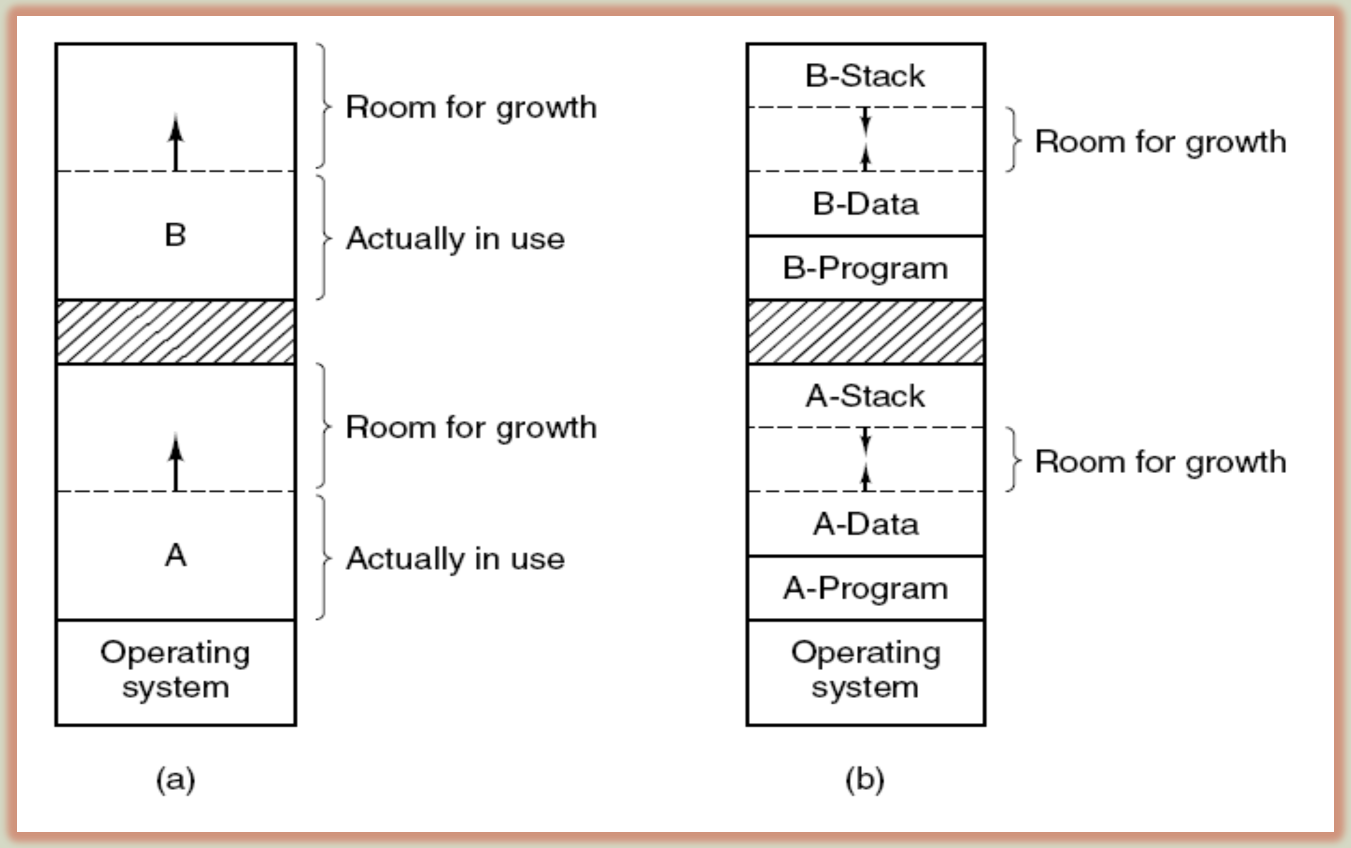
SWAPPING

- There is a different problem related with the amount of memory allocated to processes:
  - In the most simple situation, there are processes which do not require more memory during their existence than the initially allocated memory. For these processes there are is no special treatment from the operating system.
  - For the dynamic processes, one or several program segments are able to grow, based on process execution pattern.
  - Several interesting management issues can be identified:

SWAPPING

1. In the “lucky situation”, there is an extra memory block adjacent to the process:
   - the operating system could offer this extra memory block to the process.
2. Most of the processes are not so “lucky”:
   - the process is going to move to another position in memory, where there is enough space.
   - If there is not enough space, the operating system could preempt several processes in order to be able to offer the necessary space.
   - If there is still not enough space, the process itself could be preempted until there is a hole able to hold the entire process.
3. If a process cannot grow in memory and cannot be preempted, then it is going to wait in memory.
   - The final solution is to destroy the process, when this requires more than the operating system is able to offer…

SWAPPING



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

SWAPPING

- A simple solution to this problem:
  - To offer more memory to processes while they are loaded in memory or moved to another location.
- When a process is going to use two dynamic segments (Stack and Data, for example), the operating system has another possibility:
  - To offer some extra space between these two segments, and to specify that these segments are going to grow in opposite directions, using the “shared” extra space.

SWAPPING

Memory management techniques

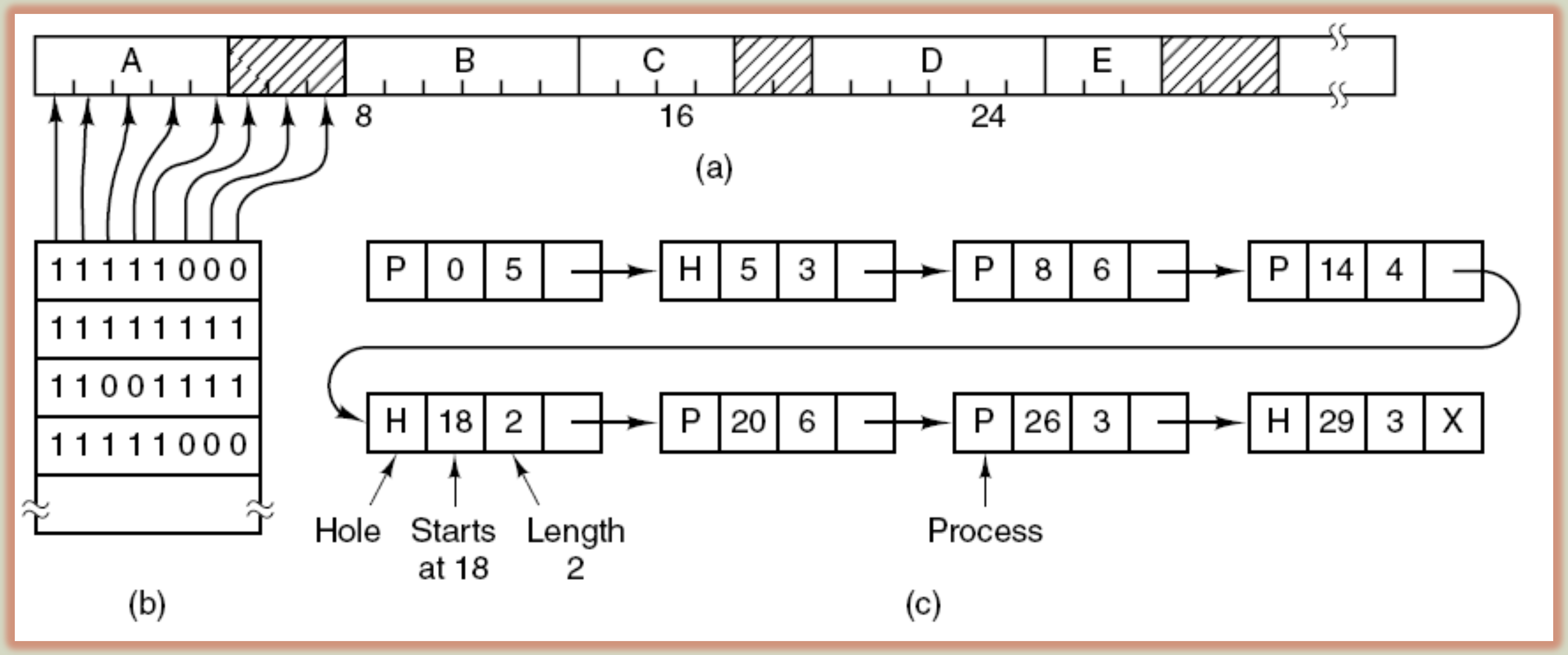
SWAPPING. MANAGEMENT BY BITMAPS.

- Bitmaps offer a simple follow-up technique for memory usage.
  - In this situation the memory is divided into several allocation units.
  - For each allocation unit there is a bit specifying if the allocation unit is free or occupied.
- The allocation map is permanently stored into memory, so the size of the allocation unit is very important.
  - E.g. For an allocation unit of 4 bytes, the allocation map requires 1/33 of memory space.

SWAPPING. MANAGEMENT BY BITMAPS.

- A small allocation size correspond with a large allocation map;
  - A large allocation size could lead to large quantities of lost memory, since it is possible to have large quantities of unused memory in an allocation unit.
- On the other hand, any decision to load a process whose size is made up of *k* allocation units requires a sequence of *k* consecutive “free”-bits in the allocation map.
  - Since the memory allocation map is stored in several consecutive bytes, this is not a trivial search.

SWAPPING. MANAGEMENT BY BITMAPS.

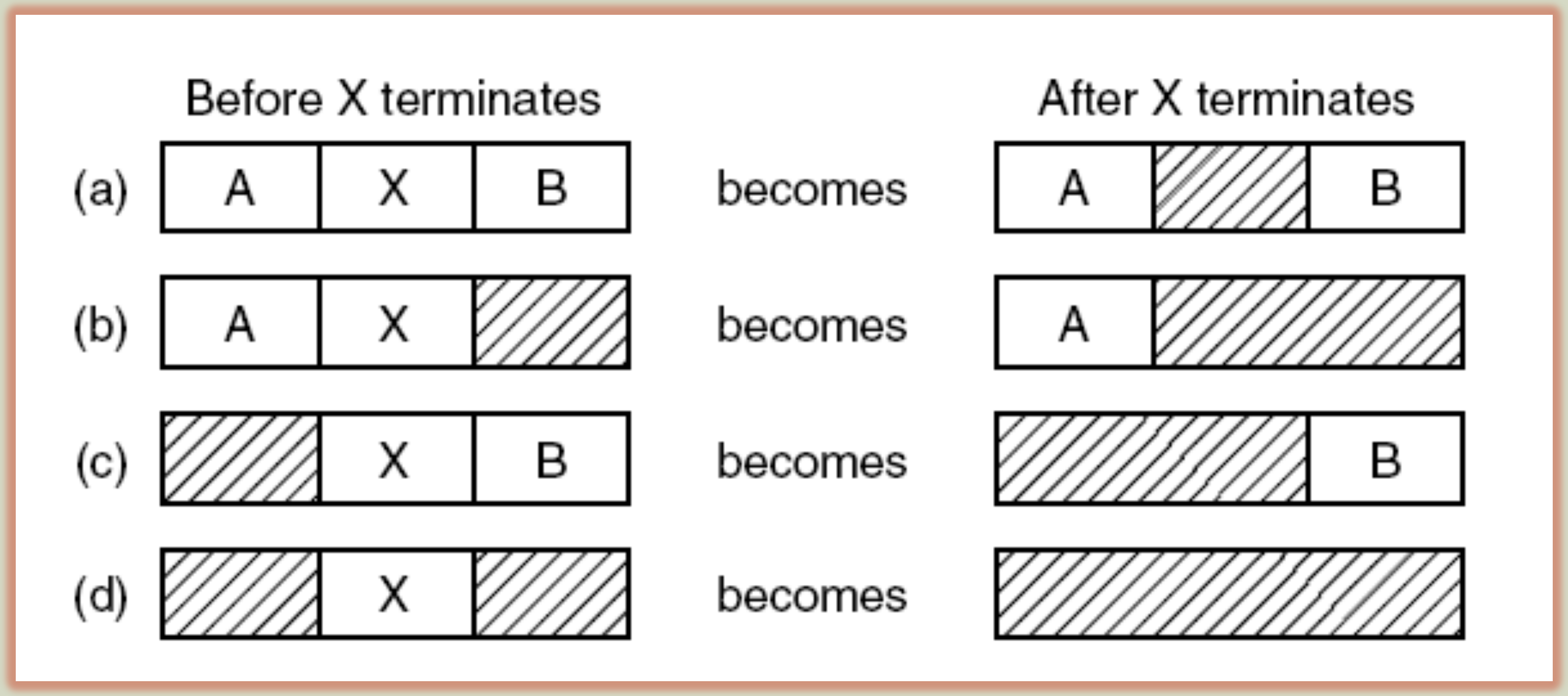


(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free.
(b) The corresponding bitmap.
(c) The same information as a list.

SWAPPING. CHAINED LISTS.

- There is another major method for following-up memory usage: use of chained lists of used and free (holes) memory segments.
  - For each memory zone the list stores the start address and the size. The list is sorted by start addresses.
  - When freeing a memory zone, one could obtain a free zone of at least the same size, after eventually combining the freed memory zone with another free, adjacent, memory zone.
  - In order to allocate free memory to processes, there are several algorithms that can be used:

SWAPPING. CHAINED LISTS.



Four neighbor combinations for the terminating process, **X**.

SWAPPING

Algorithms for chained lists management

SWAPPING. FIRST FIT

- This is the simplest algorithm:
  - parse the list of segments until one can find the first segment able to hold the process.
  - This segment is divided in two parts: one for the process and the other as a smaller hole.

SWAPPING. NEXT FIT

- This algorithm is a simple variation of the previous one:
  - next search is started where previous search stopped.
  - In this way it is possible to have a fast fit, since first segment could be a free (hole) segment.

SWAPPING. BEST FIT

- In this algorithm the entire list is searched for the segment that best fits with the process.
  - The algorithm tries to avoid dividing large segments when there are smaller segments available.
  - The results are not as expected, since it is possible to obtain a large amount of small-sized holes.

SWAPPING. WORST FIT.

- In order to avoid previous problem, now we are going to choose the largest available segment. Not so good idea for large programs.
  - The overall performance of this algorithm is not so bad, but not better than the first two algorithms.
  - Observe that one can speed-up these algorithms by offering distinct lists for processes and holes.
  - However there is a high price for this: complex list management.

SWAPPING. QUICK FIT.

- This time there are several lists for holes, based on the most used sizes.
  - New segments are stored in the most appropriate list (based on segment size).
  - The algorithm is able to quickly offer segments of the right size. However, it could be difficult to have an efficient memory compacting algorithm.
  - However, a memory compacting technique is required in order to prevent excessive memory fragmentation.

SWAPPING

Memory management problems

SWAPPING. MEMORY FRAGMENTATION (INTERNAL)

- **Internal fragmentation:**
  - this is a problem that occurs in the case of fixed-sized partitions. In this situation it is expected that programs are not able to use the entire space of a partition.
- Each partition is carrying with itself some unused space.
- The problem of the internal space of a partition that is unused due to a partition that is too large compared with the necessary amount of memory is the problem of *internal fragmentation*.

SWAPPING. MEMORY FRAGMENTATION (EXTERNAL)

- There is a similar situation for the case of dynamic partitions, due to the large amount of small-sized holes created after several operations of memory allocation.
- The problem of holes existing between the different allocated segments (processes) in memory is the problem of *external fragmentation*. Observe that, in this situation, the unallocated memory becomes more fragmented in time.
- One can eliminate external fragmentation by using some costly compacting techniques.

SWAPPING. BUDDY PARTITIONS

- Internal fragmentation is a major disadvantage both for systems based on fixed partitions and dynamic partitions.
- For dynamic partitions, external fragmentation could be “avoided” by using some compacting techniques.
- Buddy partitions were developed by Peterson and Knuth in order to minimize the effects of (internal and external) fragmentation.

SWAPPING. BUDDY PARTITIONS

1. The size of a memory block is of the form 2^k, where L<=k<=U.

2. The entire memory space is a block of size 2^U.

3. If there is a request for a space s between 2^{U-1} and 2^U, the entire memory space is allocated.

4. If the request is for a smaller size, the entire space is divided in two identical blocks, of size 2^{U-1}.

5. A request for a space s between 2^{U-2} and 2^{U-1} is satisfied by one of these blocks.

6. The process continues until there is a block of size 2^{U-k}, such that 2^{U-k-1} <s<=2^{U-k}.

7. When freeing a block, it is going to be compacted with its buddy block, if both of them are free.

Even if this algorithm is quite simple, it is not so efficient. However, it could be in use for some UNIX systems, to solve internal tasks of the kernel.

MEMORY MANAGEMENT

Virtual memory

VIRTUAL MEMORY. OVERLAYS

- The software development led to applications that are not able to fit in the physical (available) memory.
- The different swapping technologies used in multi-programmed systems, are not able to offer the possibility to go over the size of the physical memory.
- One of the initial solutions proposed to go over this memory limitations was based on splitting programs in several parts, named *overlays*.

VIRTUAL MEMORY. OVERLAYS

- The execution of a program is starting now with its first available part. Once this part is executed, another part is loaded from the disc.
- The operating system offers the possibility to keep in memory several parts (overlays) of a program. Thus, issues related with their management and design are quite difficult.
- The operating system is able to offer the necessary support for loading and unloading, in a dynamic manner, of the different parts (segments) of a program.
- However, splitting programs into several parts is a manual operation, realized by the programmer itself.

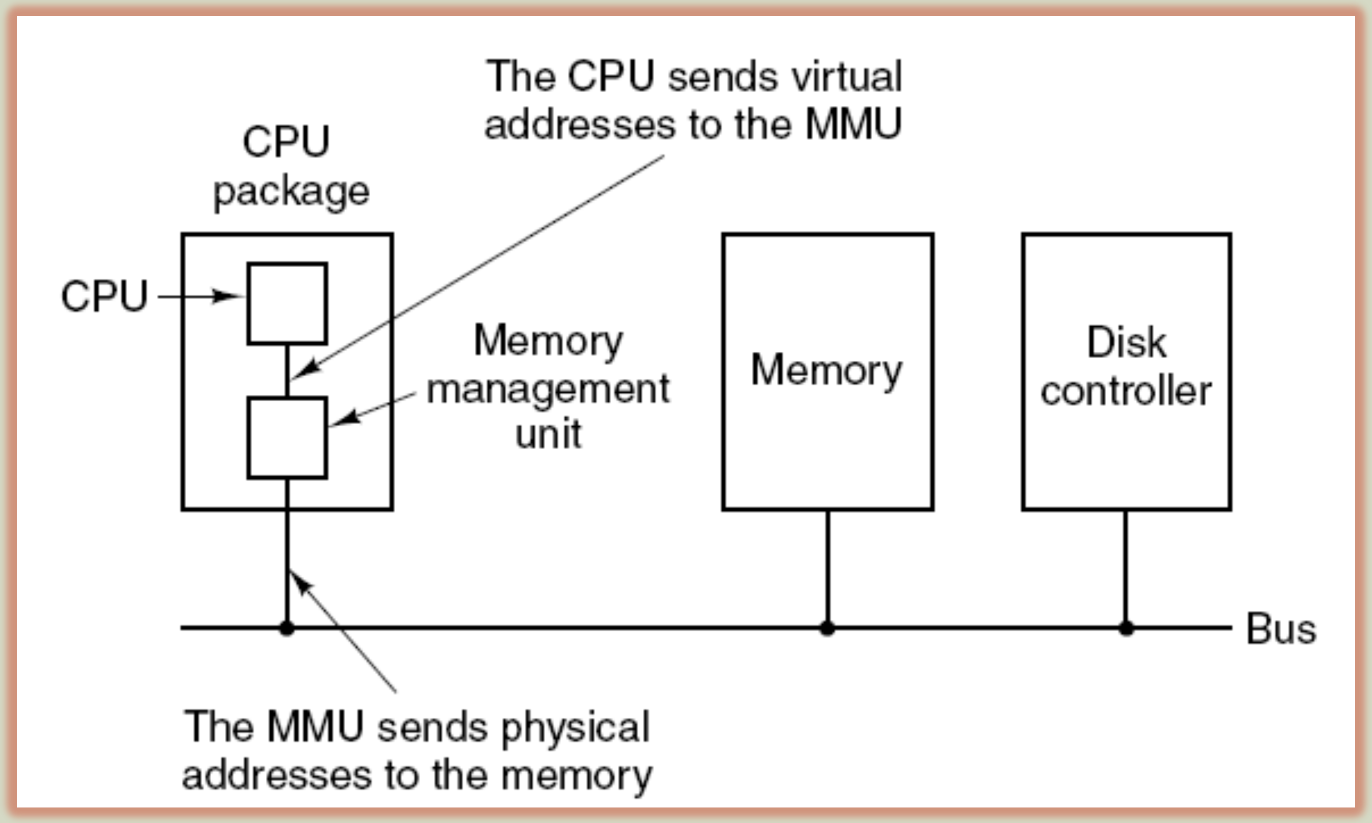
VIRTUAL MEMORY. OVERLAYS

- By automating the task of splitting programs in several parts (initially proposed by Fotheringham), the very first techniques of virtual memory were introduced.
- In this situation, the operating system is going to detect by itself if an application must be split into several parts, and to detect how it can realize this split.
- In a multi-programmed system, a process that is waiting for a segment to be loaded in memory is considered as being suspended for an input/output operation.

PAGING

- Paging is a basic technique for systems based on virtual memory.
- In a computing system there is a possibility to generate a limited amount of addresses. However, these addresses can be over the available physical memory:
  - E.g: for a 16 bits system, one can generate addresses up to the 64K limit. For these systems it was possible to have a physical memory of only… 32K.
- Addresses generated by applications are called virtual addresses, and they form the space of virtual addresses, sometimes different from the space of physical addresses. In order to solve these differences, virtual addresses are mapped on physical addresses through the MMU.

PAGING


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

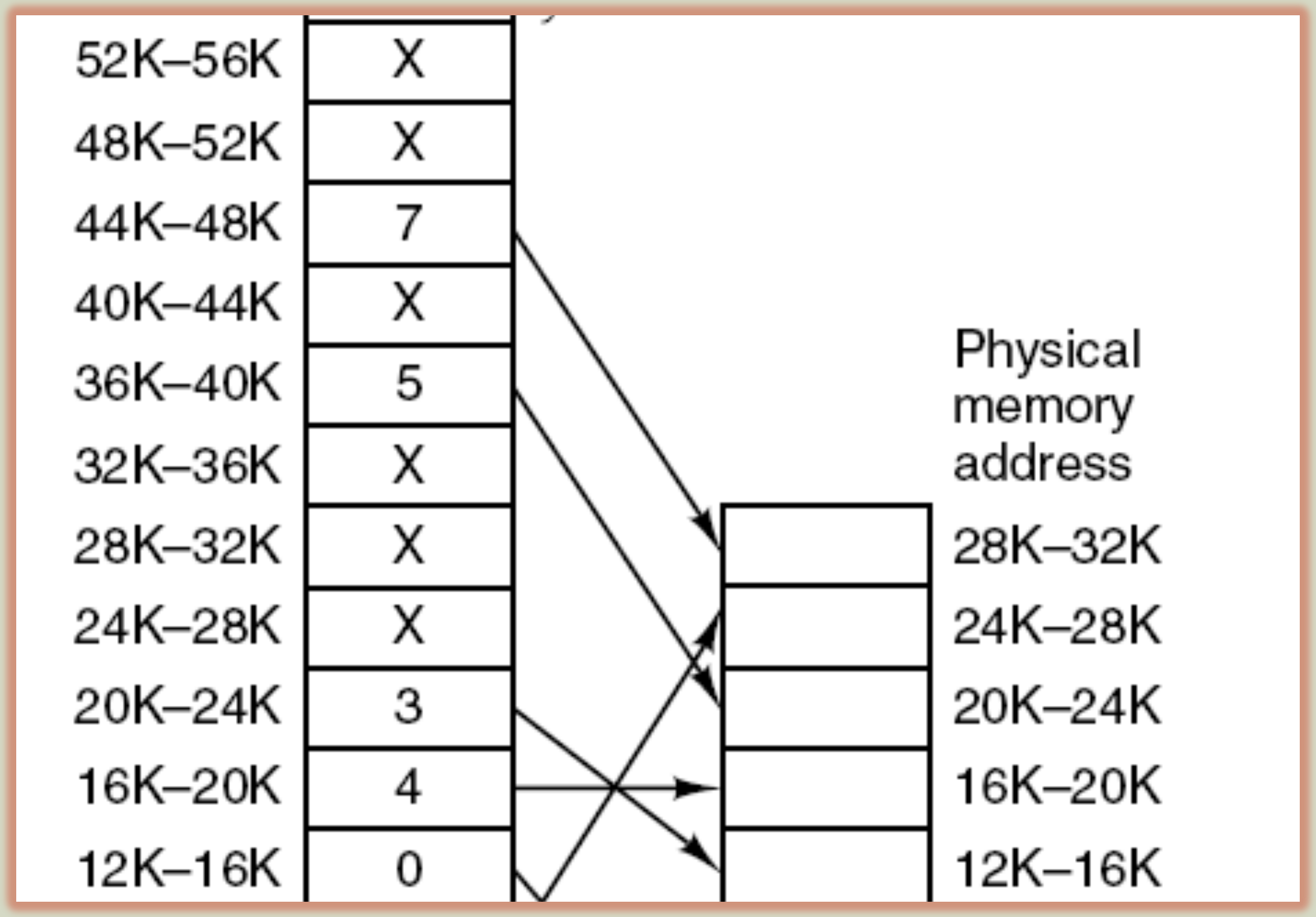
The position and function of the MMU – shown as being a part of the CPU chip (it commonly is nowadays).

PAGING

- If the space of virtual addresses is higher than the space of physical addresses, even if one can construct applications that are able to use the entire virtual space, observe that now it is impossible to load such an application in memory.
- The space of virtual addresses is usually divided into several *pages of addresses*. There is a correspondent for these pages in the physical memory: *page frames*.
- The requests for memory accesses are now solved as follows:
  - First identify the corresponding virtual page of addresses.
  - For this page, now we are going to identify the frame page on which it is mapped.
  - The “real” address is now computed by the MMU, relative to the starting address of the frame page, using the same offset as in the page of addresses.

PAGING

Relation between virtual addresses and physical memory addresses given by page table.
Every page begins on a multiple of 4096, ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.



PAGING

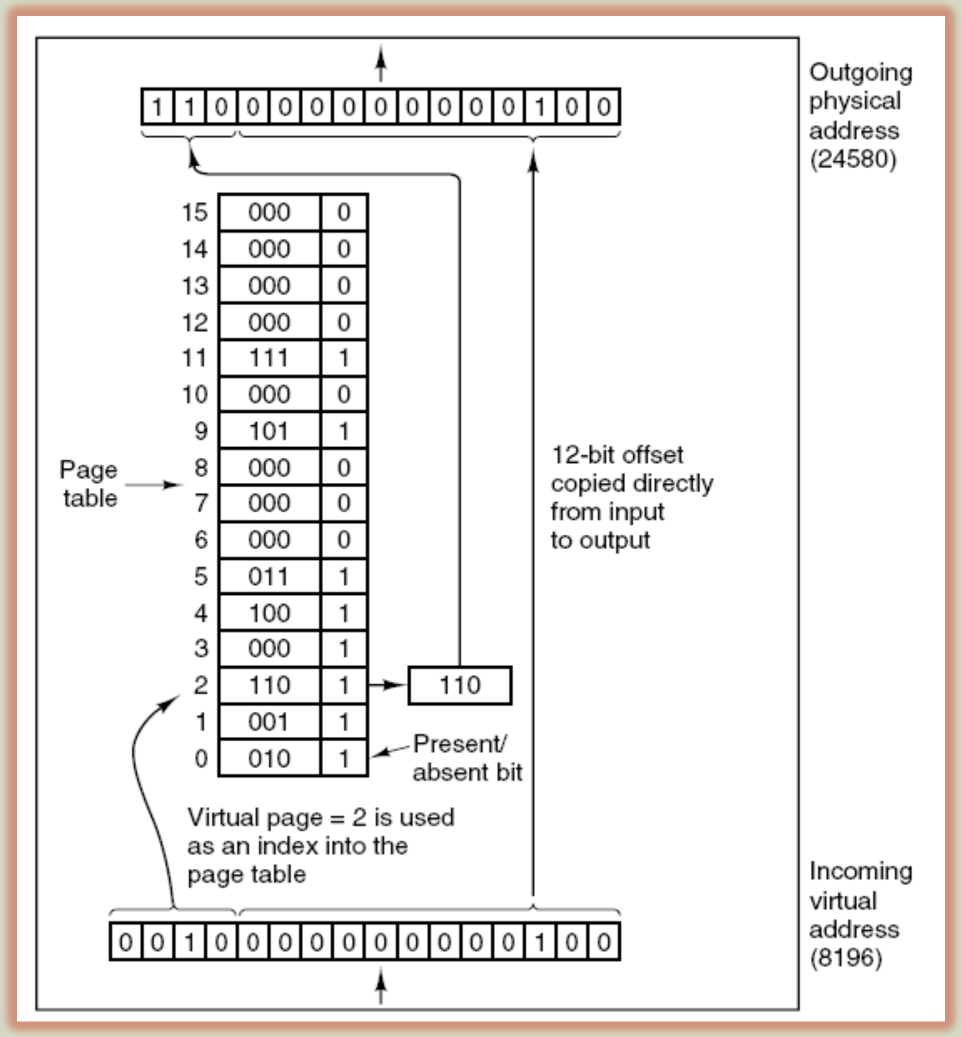
- If the page of addresses is not mapped on any frame page, a page fault is issued, which is going to determine a replacement of one of the pages already mapped in memory with the newly accessed page.
  - In order to support this operation, usually a *presence bit* is used. This bit is able to signal if a virtual page is already mapped on a frame page.
- After solving this remapping, the initial mapping process is re-starting by the MMU.

PAGING

- Internal MMU operations are solved as follows:
  - A virtual address on 16 bits is decomposed in a virtual page number (on 4 bits) and an offset (12 bits).
  - The virtual page number is used as an index in a page table, to determine the value of the frame page. If the presence bit is not set, a page remapping process is first performed.
  - The page number (now, only 3 bits) is appended to the offset, together they form the physical address.

PAGING. INTERNAL MMU OPERATIONS

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



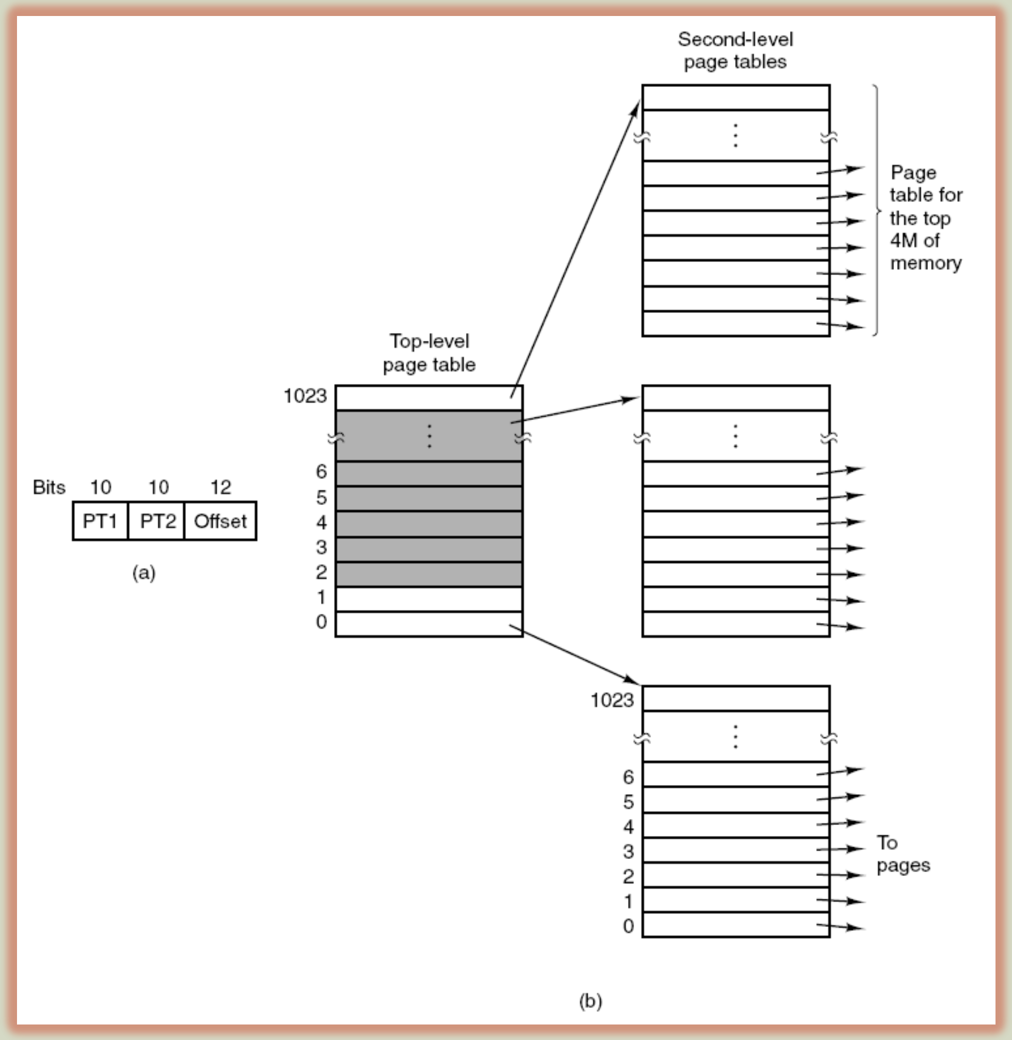
PAGING. PAGE TABLES.

- Page tables were developed in order to support the mapping process from virtual pages to physical pages.
- A page table has to answer to several requirements, coming from the following observations:
  - The size of a page table could be very big;
  - The mapping process must be solved very fast;
- E.g. For a 32 bit system, with a page size of only 4K, the total number of virtual pages is 1048576… Each process will use a page table of this size!

PAGING. PAGE TABLES

- The mapping is performed on every memory operation, and it is supposed to have one or more references to the page table. For efficient tables, one must have very short access times.
- A possible solution is to let the operating system to load the page table on an array of fast hardware registers. However, this is a costly solution, since this table could be quite large.
- An alternative to this solution is to keep the entire table in memory for a process, and to use only one register to keep the start address of the table in memory. However, the mapping process is quite slow, since now we need a couple of references to memory.

PAGING. MULTILEVEL PAGE TABLES

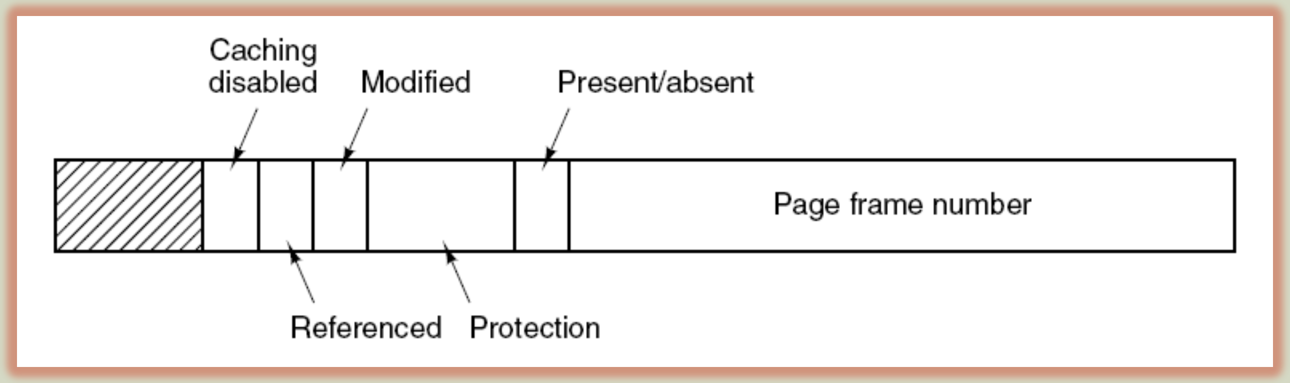


(a) A 32-bit address with two page table fields.
(b) Two-level page tables.

PAGING. PAGE TABLES

- In order to avoid keeping in memory the entire page table, one can use a solution based on multi-level tables.
- Thus, for 32 bits addresses, the necessary amount of pages, at a size of 4K, is 2^{20}.
- In a multi-level solution, the 32 bits address is divided in three parts: 10+10+12 bits.
- First 10 bits are used for an indexation in the higher level table (size: 4M).
- By using this information, the next 10 bits are used for an indexation in the identified level 2 table.
- Finally, the last 12 bits are used to identify the address inside the frame page.
- In this situation we are going to use again paging faults in order to signal that a page is not present in memory and to request its loading.
- Usually, an application is going to use only 4 tables: one on level 1 and 3 on level 2.

PAGING. PAGE TABLES



A typical page table entry

PAGING. PAGE TABLES

- The following information could be stored in entries from a page table:
  - The “caching” bit: used to avoid page caching; useful for systems that are mapping pages on I/O registers.
  - Reference bit: this bit is set on every reference to a page. Used to preempt pages from memory.
  - Modification bit (dirty bit): used to specify if the page was modified, to detect if it must be saved.
  - Protection bits: used to specify different types of access that are allowed.
  - Presence bit.
  - Frame page number.

PAGING. ASSOCIATIVE TABLES

- The different paging schemes are keeping page tables in memory, even if they are quite big.
- Using paging could have a bad impact over the performance of the entire system, since every memory access requires supplemental accesses due to the usage of… page tables.
- However, most of computer programs are using a large amount of references to a small amount of pages.

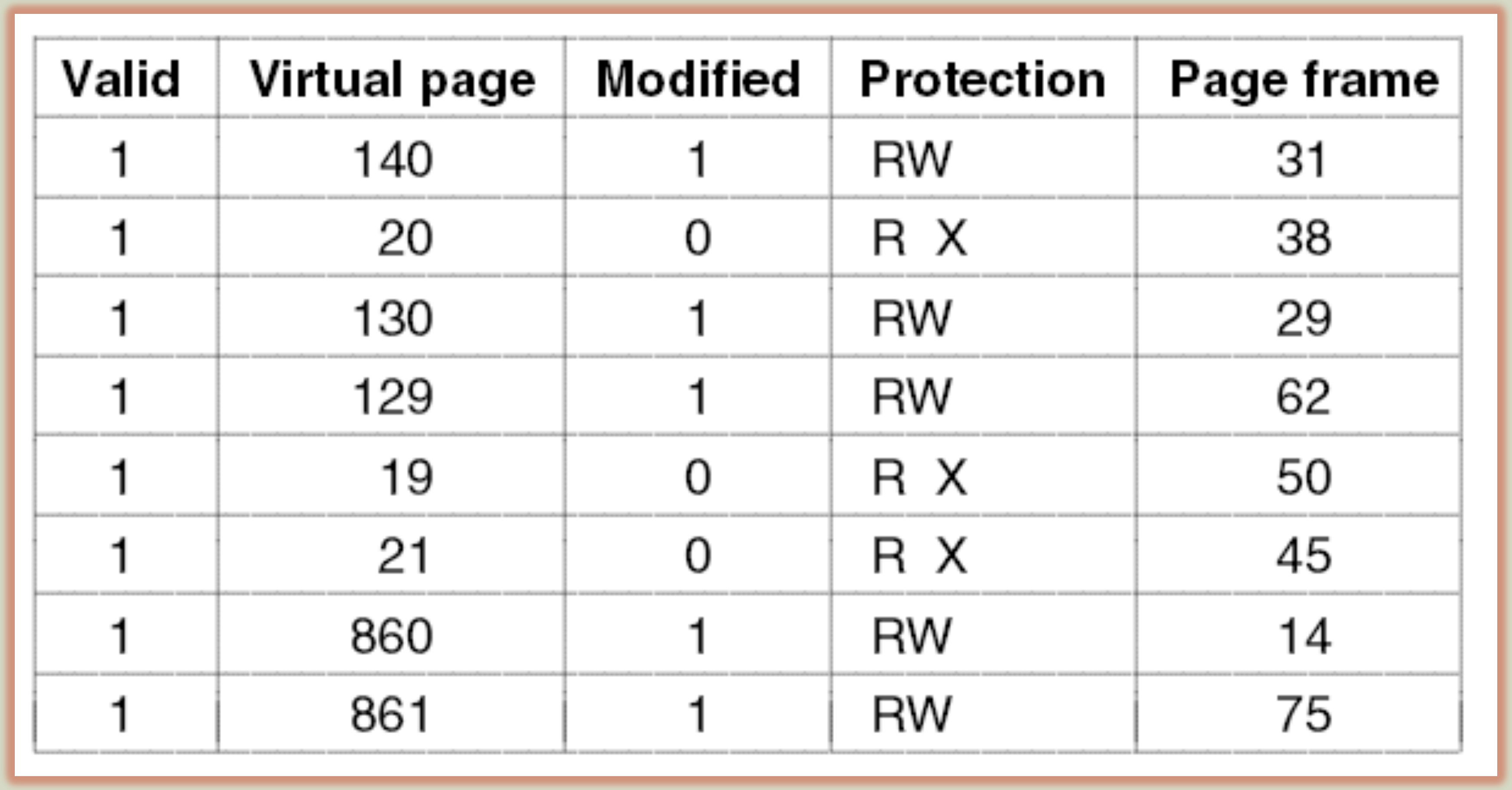
PAGING. ASSOCIATIVE TABLES

- The associative memory is only a hardware device, integrated in the MMU, that has several entries. It is used to quickly solve the mapping of virtual addresses on physical addresses, avoiding the usage of page tables.
- An entry in the associative memory area could contain information like: virtual page number, the modification bit, the protection code, the (physical) frame page, and an extra bit indicating used pages.

PAGING. ASSOCIATIVE TABLES

- The function of TLB (Translation Lookaside Buffers) is as follows:
  - MMU receives a virtual address for translation: verify first if the associated physical page is in TLB.
  - If there is a math, and if the values for protection bits are correct, the frame page is picked from the TLB; otherwise a page fault is issued.
  - If there is no match, the page is searched via the MMU by using the ordinary page tables mechanism; then a TLB entry is replaced with the new entry.

PAGING. TRANSLATION LOOKASIDE BUFFERS



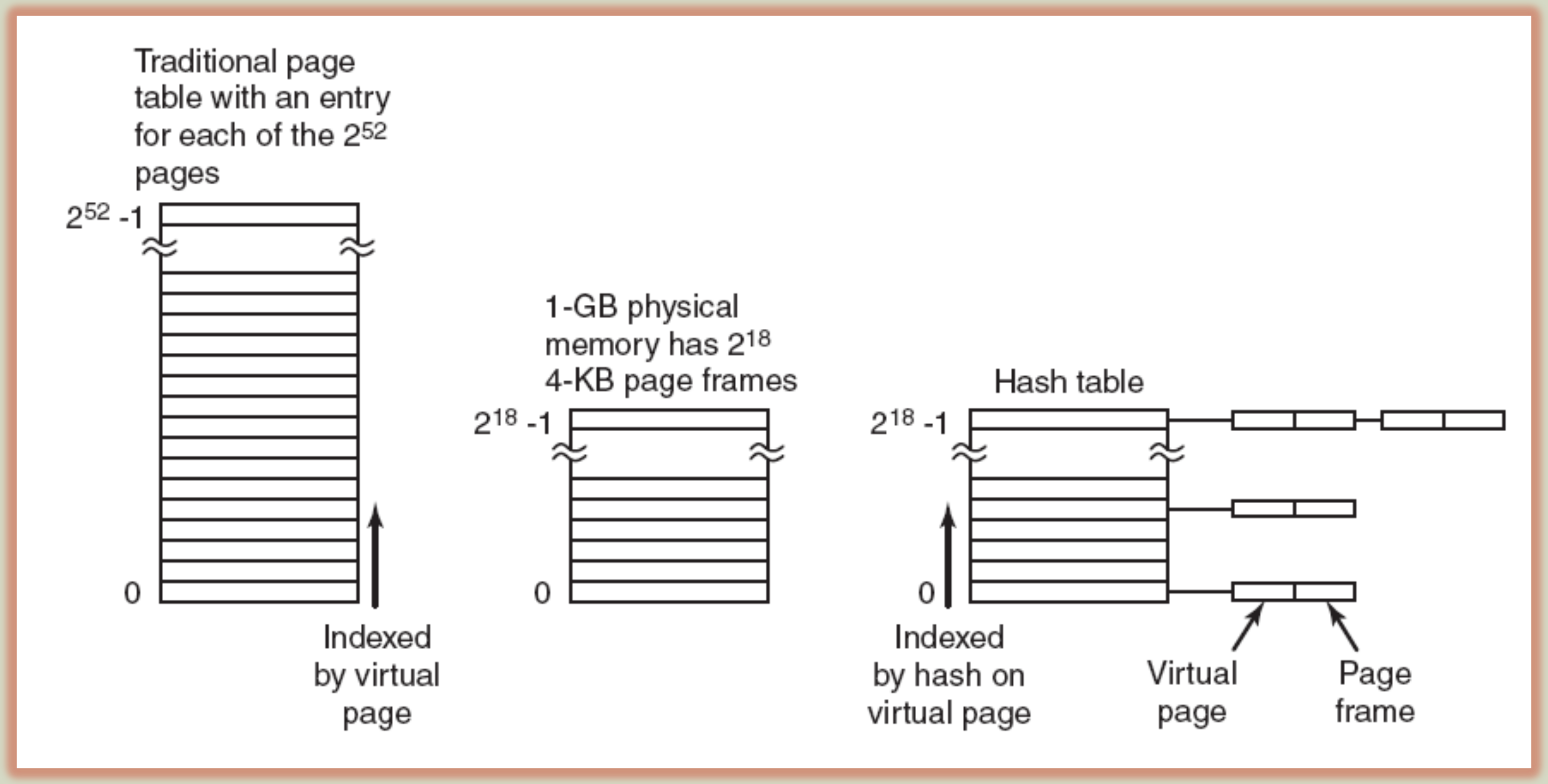
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB structure for speeding up paging

PAGING. INVERTED TABLES

- Traditional page tables offer one entry for each virtual page. For a system on 32 bits, the size of a page table could be only 4M.
- For a 64 bits system, the table could reach a size of $2^{64}/2^{12}$, approx. 33554432G!
- The inverted table is trying to offer an alternative solution: instead of having an entry for each virtual page, we are going to keep in memory an entry for every frame page.
- Each entry is able now to emphasize their “inhabitants”. A reference to a virtual page is now solved through a search in the entire table…

PAGING. INVERTED TABLES



Traditional page table with an entry for each of the 2^52 pages
2^52 -1
0
Indexed by virtual page
1-GB physical memory has 2^18 4-KB page frames
2^18 -1
0
Hash table
2^18 -1
0
Indexed by hash on virtual page
Virtual page
Page frame

Traditional page table vs. inverted page table

PAGING.

Page replacement algorithms

REPLACEMENT ALGORITHMS

- When there is a pagination error, the operating system is obliged to identify the page to be eliminated from memory, to make room for a new page.
- There are several policies to be used for this process: random choice; choosing a page with low usage; choosing a page in a pre-determined order, etc.

REPLACEMENT ALGORITHMS

- Optimal page replacement algorithm
- Not recently used page replacement
- First-In, First-Out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement
- Working set page replacement
- WSClock page replacement

OPTIMAL ALGORITHM

- This algorithm is based on a very simple idea: whenever there is a pagination error, there is only one page to be referred for the next instruction.
- The idea is to label pages with the number of next instructions to be executed, in order: one should eliminate the page with the higher label available.
- This algorithm is hard to be used in a real system, since the operating system is not able to predict the moments when it is going to use the different memory pages. However, the simulation of this algorithm could be useful in order to model the “real” algorithms.

NRU ALGORITHM

- This algorithm is based on two distinct bits: R (reference to a page), and M (modified page). These two bits are updated on each reference to memory.
  - When a process is started, all entries are marked as not being present in memory: first access is going to generate a pagination error.
  - Following this initial step, the R bit is set and the corresponding entry in the page table is modified.
  - When the page is modified, a pagination error is issued, such that the operating system is able to operate on the M bit.
- Based on these observation, the algorithm is as follows:
  - When a process is starting, the R and M bits are unset for all pages.
  - From time to time, the R bit is reset. This is necessary to make the difference between recently referred pages and not-referred pages.
  - When a paging error is issued, all pages are classified in one of the following categories:
    - 0: not-referred and not-modified pages
    - 1: not-referred pages but modified
    - 2: referred pages but not-modified
    - 3: referred and modified pages
  - The operating system randomly choose a page from the lowest available class.

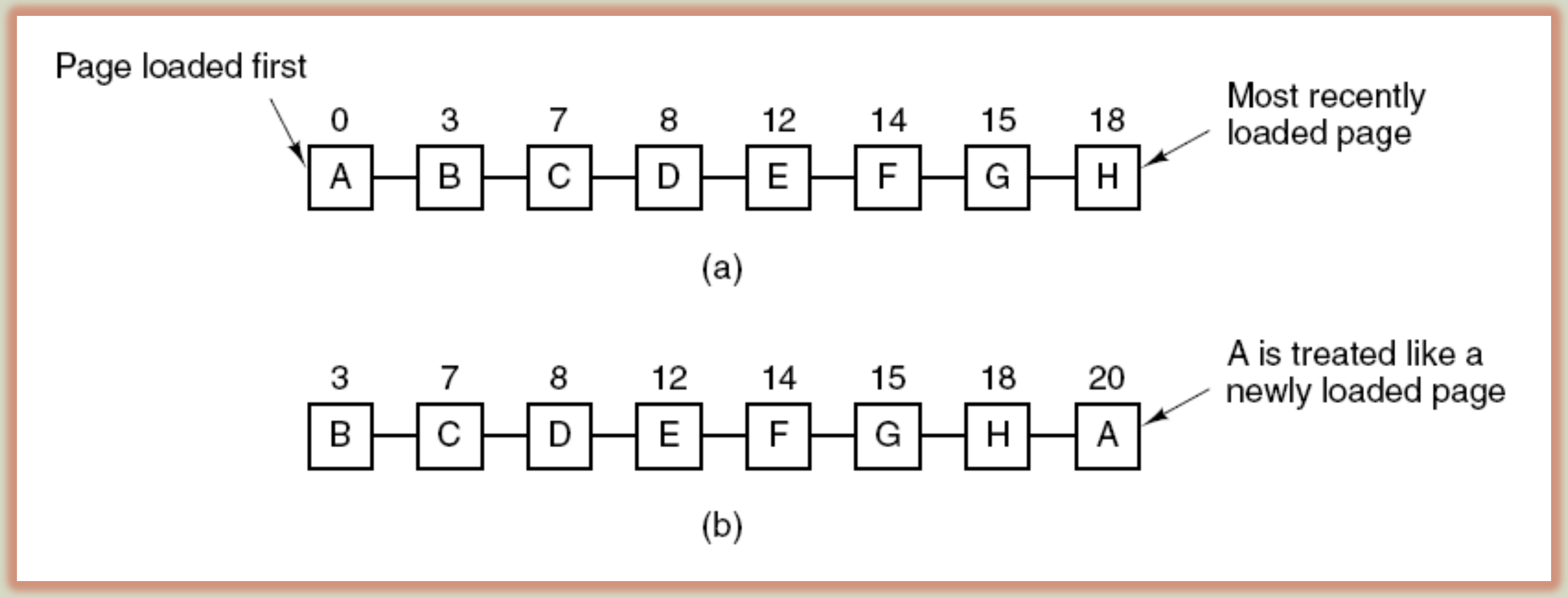
FIFO ALGORITHM

- The idea of this algorithm is very simple: one should choose the oldest available page in the system.
- For this task, the operating system is maintaining a list of all pages in the system. First page in the list is the oldest page…
- When there is a pagination error, one should choose the first page in the list.

SECOND CHANCE ALGORITHM

- This algorithm is based on the observation that FIFO algorithm is able to eliminate a useful page (old, but intensely used).
- This algorithm is trying to avoid this operation for a recently used page: if the R bit is set, the page is moved to the end of the list, with the R bit unset.
- If the R bit is not set, the page is eliminated and saved if necessary (the M bit is also set).

SECOND CHANCE ALGORITHM

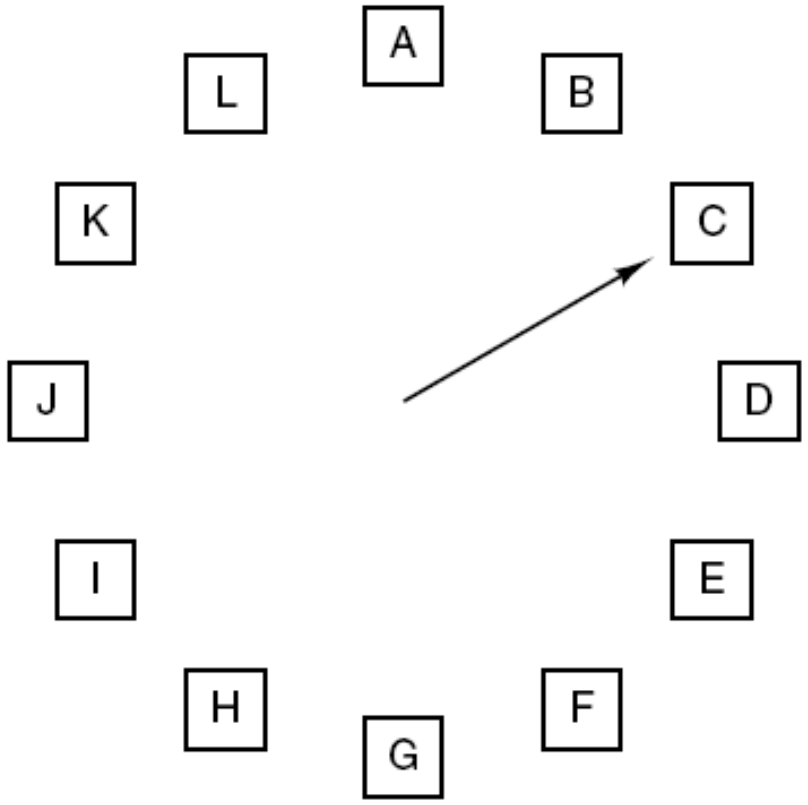


Operation of second chance.
(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

CLOCK ALGORITHM

- Previous algorithm requires an intense activity for page list management.
- The clock algorithm is based on a circular list, where there is a pointer for the oldest page in the system.
- This time, a pagination error is followed by a replacement of the page if the R bit is not set, or change the pointer to next page, and reset the R bit.

CLOCK ALGORITHM



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

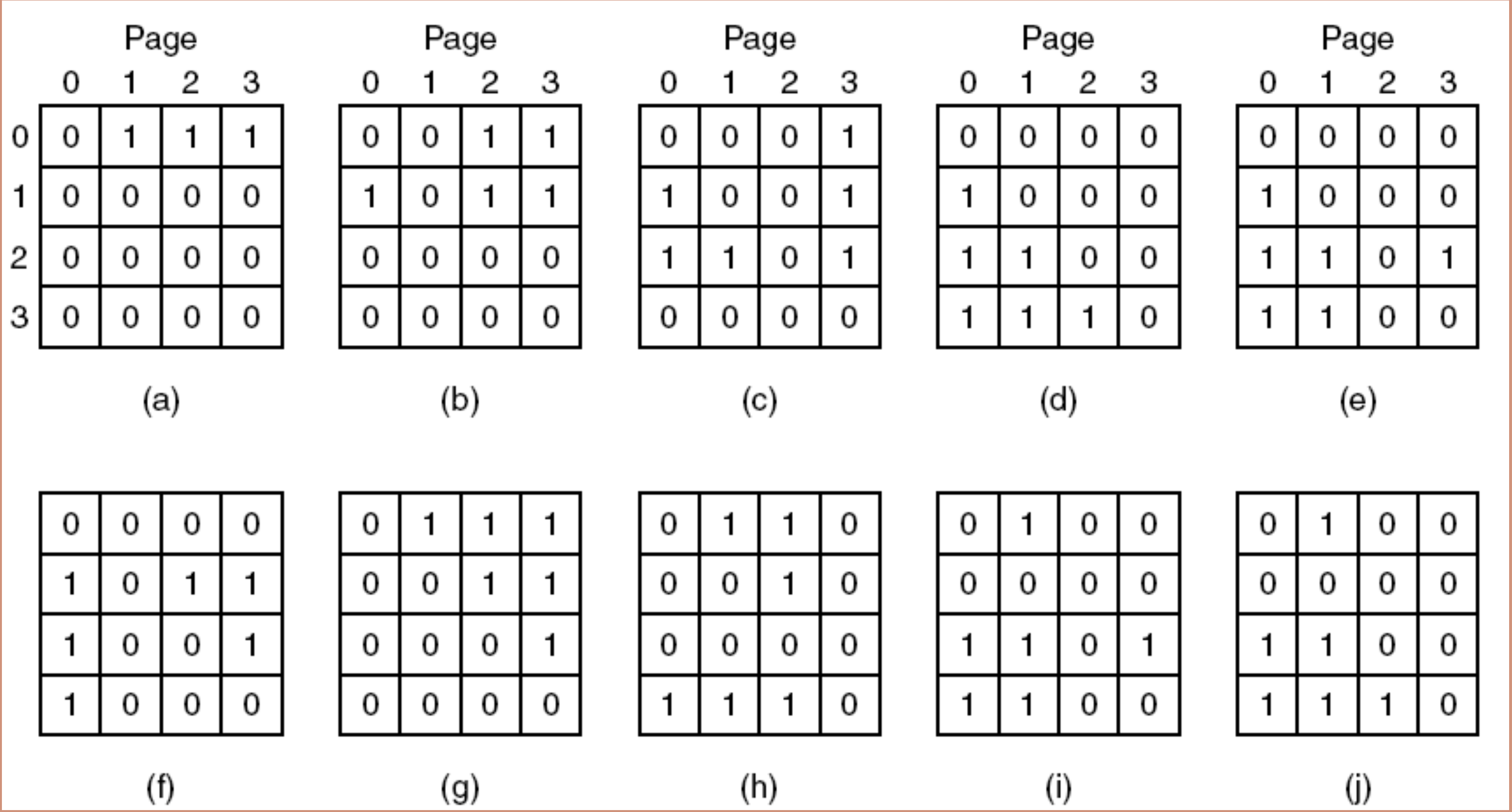
R = 0: Evict the page

R = 1: Clear R and advance hand

THE LRU ALGORITHM

- This algorithm is based on the idea that intensely used pages are supposed to have an intense usage in the future. When there is a pagination fault, one of the unused pages is going to be removed from memory.
- This algorithm could have a high cost: one must keep a list of pages that are in memory, and to permanently update this list based on page usage.
- This algorithm could be used if there is some hardware support. For example, a hardware device containing a matrix of nxn bits, initially unset. On a reference to a frame page k, the bits on the k^{th} line are set and those on the k^{th} column are unset. The page with the lowest binary value is chosen for replacement.

LRU ALGORITHM



Page

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

(a)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}$$

(b)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$

(c)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$

(d)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{array}$$

(e)

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{array}$$

(f)

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$

(g)

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$

(h)

$$\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{array}$$

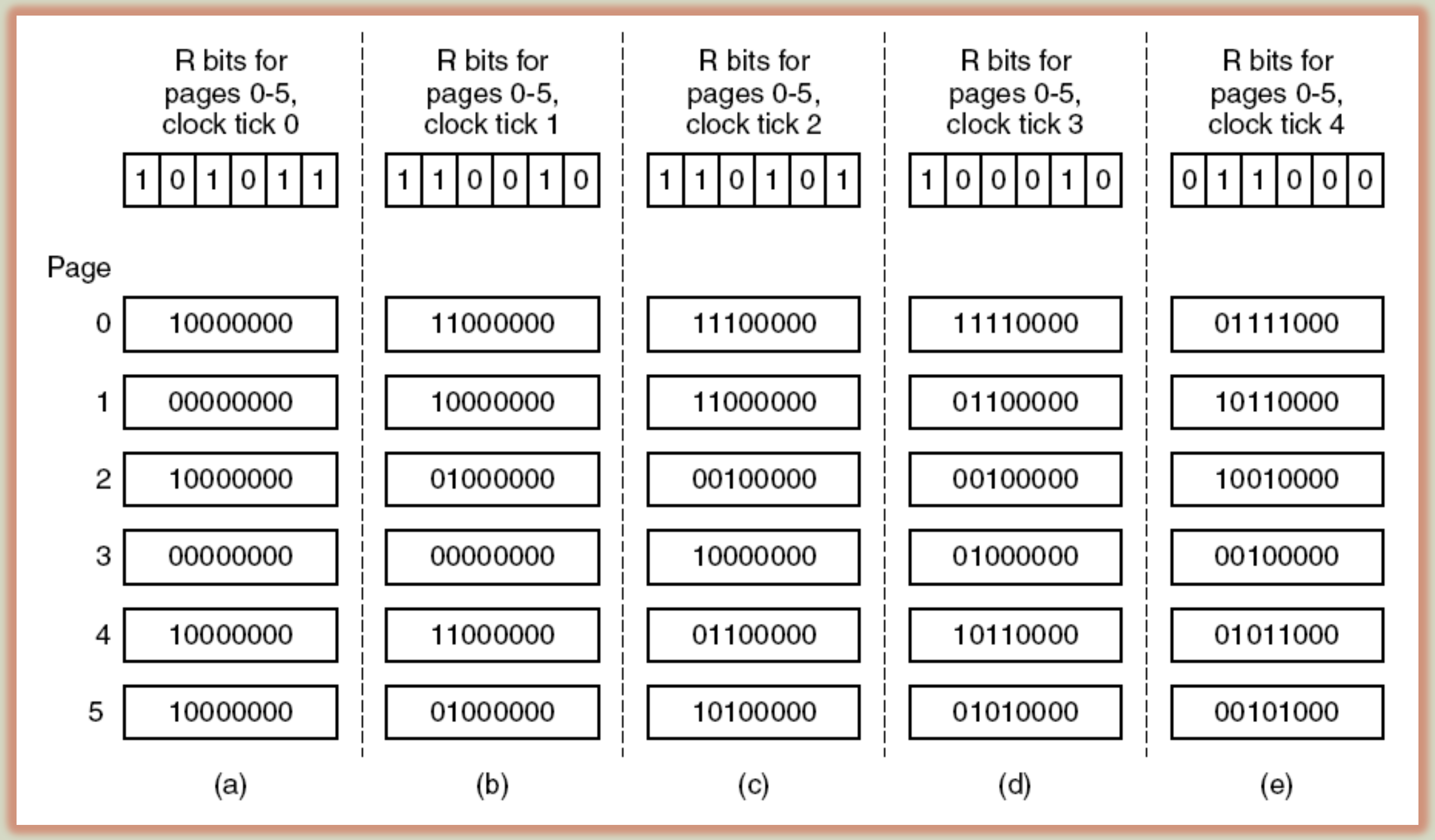
(i)

$$\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$

(j)

The LRU algorithm, using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

LRU ALGORITHM



The aging algorithm simulates LRU in software.
Shown are six pages for five clock ticks.
The five clock ticks are represented by (a) to (e).

PAGE FAULT HANDLING

1. The hardware traps to the kernel, saving the program counter on the stack.
2. An assembly code routine is started to save the general registers and other volatile information.
3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed.
4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access

PAGE FAULT HANDLING

5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
6. When page frame is clean, operating system looks up the disk address where the needed page is, schedules a disk operation to bring it in.
7. When disk interrupt indicates page has arrived, page tables updated to reflect position, frame marked as being in normal state.

PAGE FAULT HANDLING

8. Faulting instruction backed up to state it had when it began and program counter reset to point to that instruction.
9. Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
10. This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.

SEGMENTATION

SEGMENTATION

- In the case of virtual memory the addresses are in a fixed range of values, starting from a minimal value (usually 0) and going to a maximum, and the values being specified as successive values.
- Many problems could rely on the existence of two or many distinctive spaces of addresses. For example, a compiler could use during its functioning at least five different tables.
- The solution that one can consider is to offer to the machine a large number of distinct address spaces, called ***segments***.
  - A segment is a linear space of addresses.
  - Different segments could have different sizes.
  - Segment dimensions modify in time based on application needs.
- In order to represent addresses in a bi-dimensional space one must identify two parts: segment number and the internal address.

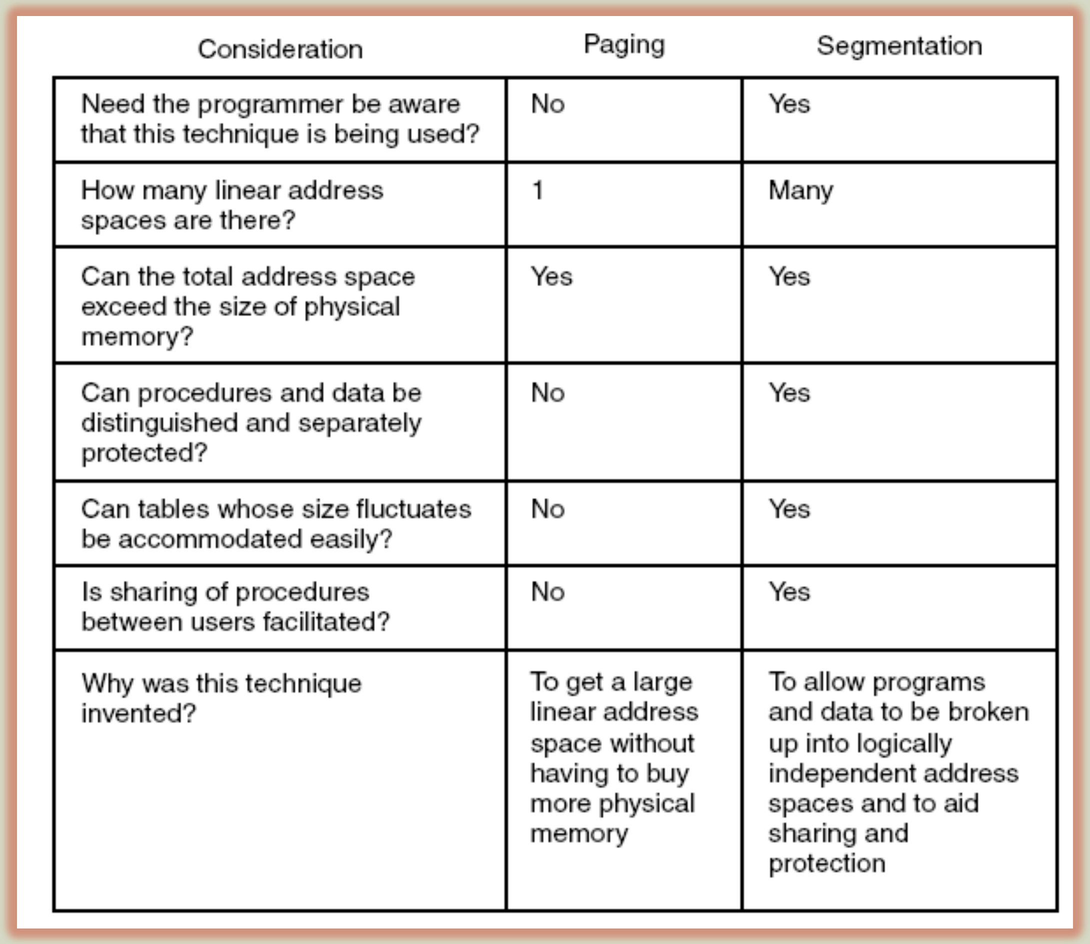
SEGMENTATION

- By using segmented memory one can simplify the handling of data structures with variable dimensions.
- Linking compiled procedures is a simple process:
  - just put every procedure in a separate segment; a call to a procedure is translated to an address of the form (n,0), where n is the segment holding the procedure.
- Segmentation could also offer the possibility to share procedures or data between several processes.

PURE SEGMENTATION

- Segmentation implementation is different from pagination implementation. Segment sizes are very different, opposed to page sizes.
- In this case, external segmentation is a crude reality, after enough segment load/unload operations from memory.

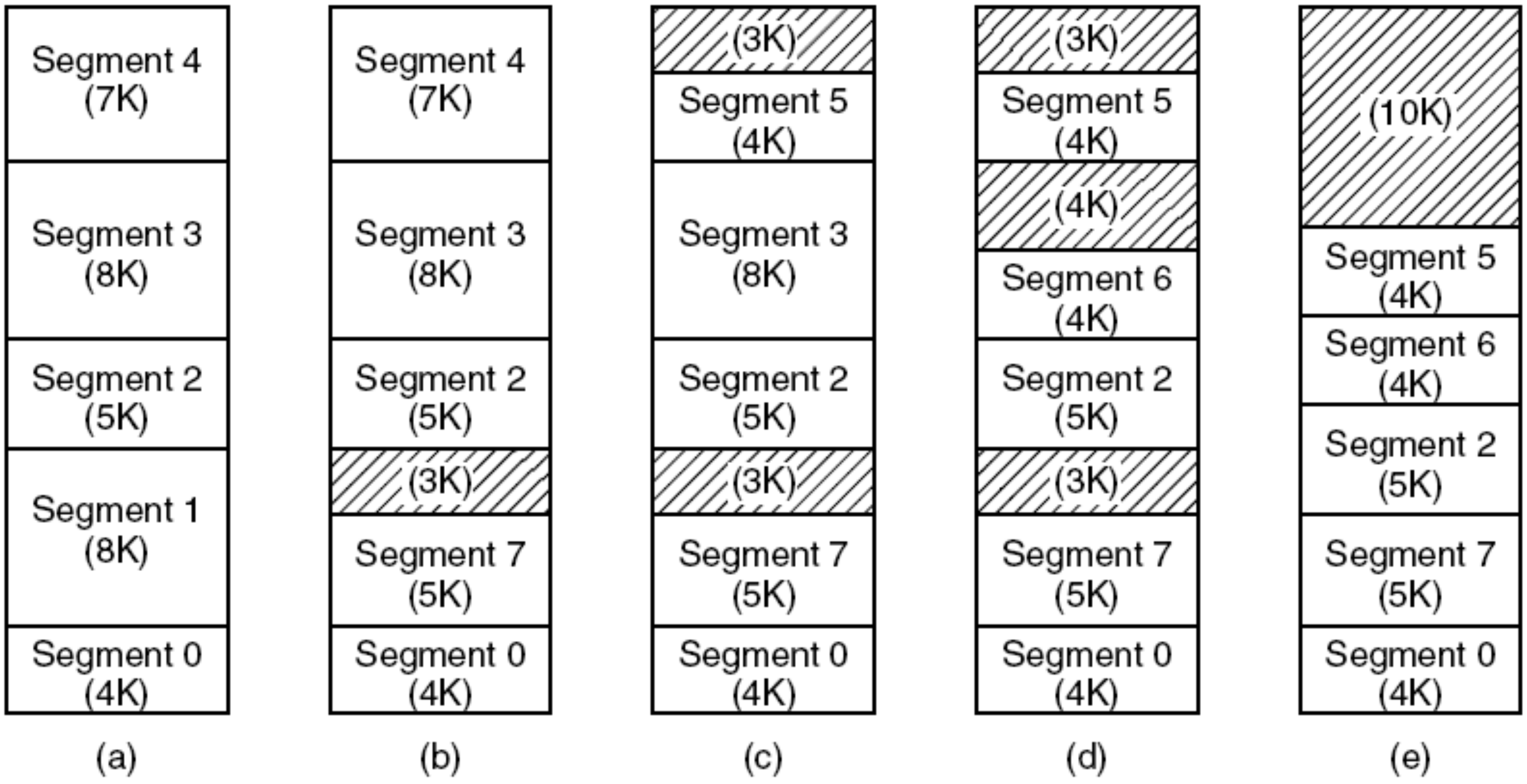
PURE SEGMENTATION



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

Comparison of paging and segmentation

SEGMENTATION. CHECKERBOARDING



Checkerboarding development and removal by compaction