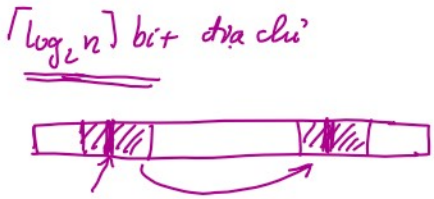
2. Memory management

Tuesday, April 7, 2020 4:29 PM

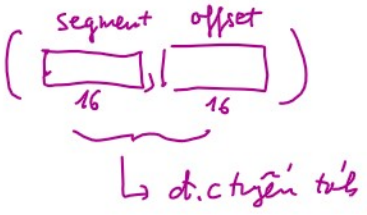
2.1 Memory addressing



Linear address

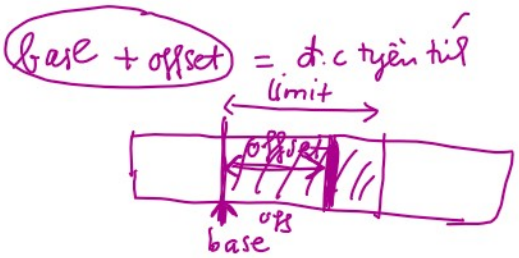
Bytes in memory are numbered from 0 .. n-1, where n is the size of the memory (in byte).

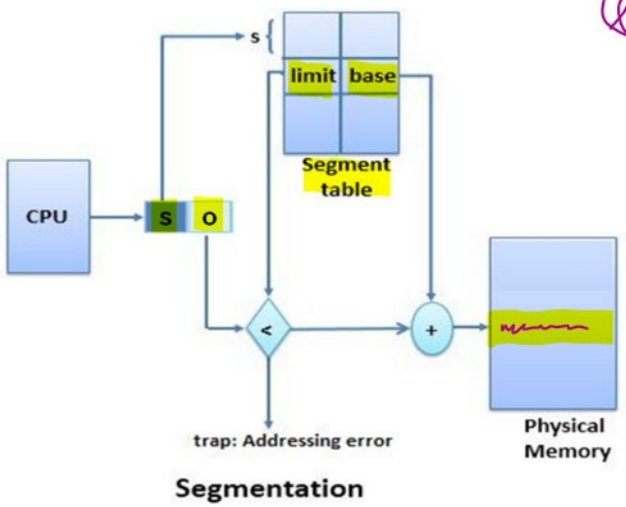
Segmentation address



An address is divided into segment and offset addresses.

- Segment address is used as an index pointing to a record in Segment Table
- A record in Segment Table consists of 2 fields: base and limit
- Base address is a linear address to a segment of memory; Limit determines the size of the segment
- Offset addess is a relative address starting from the beginning of a segment
- If $offset \leq limit$ this is a valid address, and the linear address is calculated as $base\ address + offset\ address$
- If $offset > limit$ raise segmentation fault

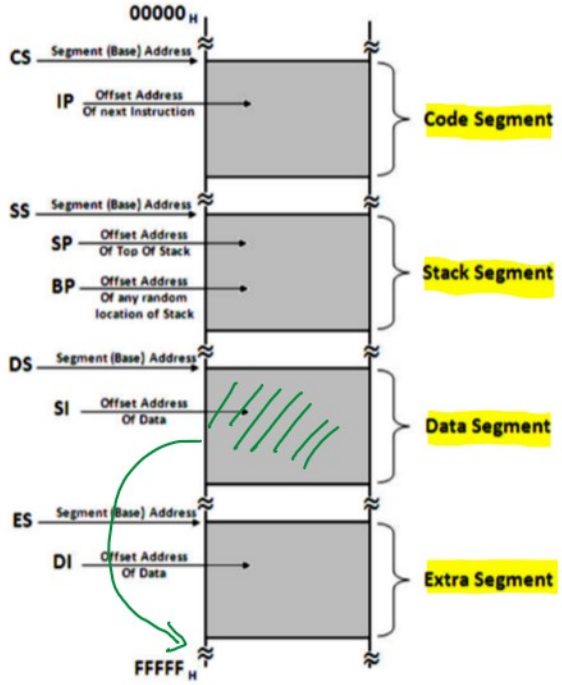




Segmentation

A program is compiled into different segments: code, data, stack, extra segments. Memory access within a segment requires only offset address. It fairly easy to move a segment around without changing the offset address.

Segmentation is used in 32-bit CPUs of Intel x86. Starting from Intel x86 64 bit, segmentation is no longer necessary and only used for backward compatibility.



quora.com

segment : offset

CS : IP 16 / 32

Seg : offs

SS : SP
BP

Seg : offs

DS : SI

ES : DI

Example of segmentation in Intel 8086: As CPU is 16-bit, an address is also 16-bit length. Thus, each segment is maximal 64KB in size.

Intel x86

Segment 2^{16} B = 64kB

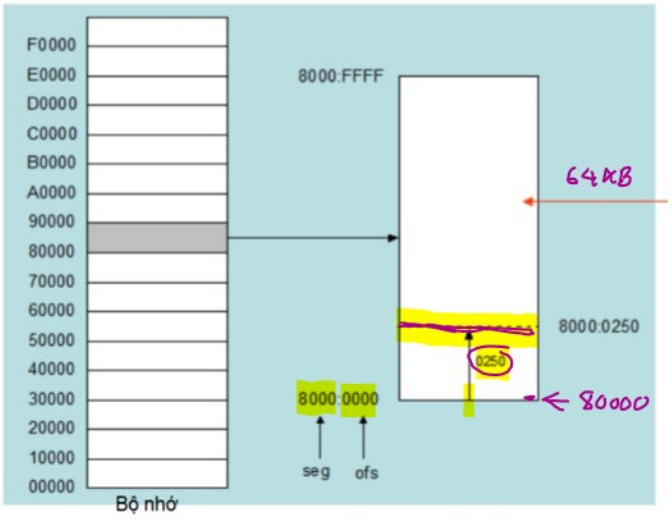
$2^{16} \cdot 2^4 = 2^{20}$

2^{20} B = 1MB

seg + offset = dc thực
16 16 16b

$$Linear\ address = segment \times 16 + offset$$

Maximal memory size is 1MB, consisting of 16 segments.



Segmentation

0 x 16 = 0
1 x 16 = 16
2 x 16 = 32

16 x 64kB = 1MB

seg : offs 8000 : 0000

$8000_H : 0250_H$

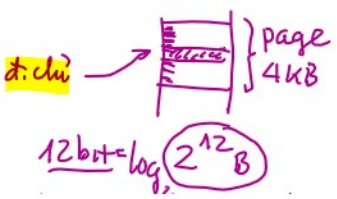
$\Rightarrow 8000_H \times 10_H + 0250$

$$\begin{array}{r} 80000 \\ 0250 \\ \hline 80250_H \end{array}$$ 20 bit

Paging

Memory is divided into pages for allocation, of 4KB size, usually.

A linear address can be understood as 2 parts: page address and offset address within the page.

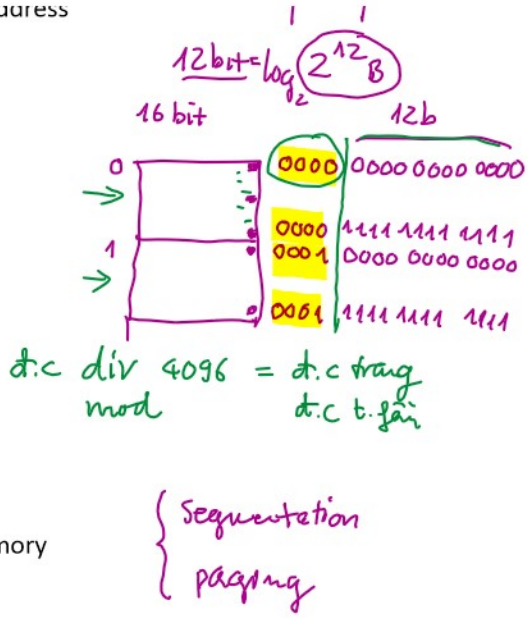


12bit = $\log_2(2^{12}$ B)

A linear address can be understood as 2 parts: page address and offset address within the page.

Example: address 4098 points the 3rd byte in page 1.

$4098_D = 0001\ 0000\ 0000\ 0010_B$
Page address is 0001_B, offset address is $0000\ 0000\ 0010_B$



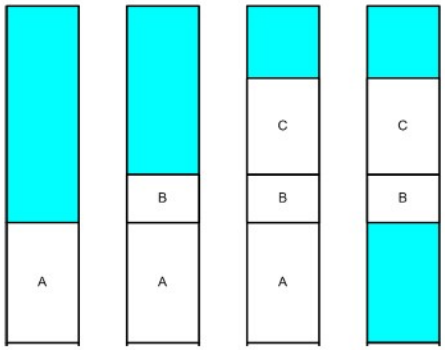
Application of paging in memory allocation/deallocation and virtual memory management will be discussed hereafter.

Segmentation
paging

2.2 Memory allocation and deallocation

Memory allocation for processes

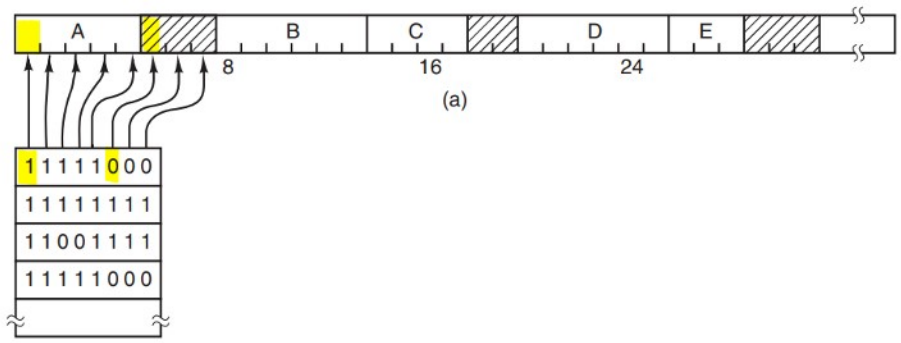
Processes are loaded into memory, execute and finish, then their memory is deallocated which makes the memory becomes fragmented. Fragmentation issue becomes even more serious when the memory is full and some pages have to be swapped into external memory to create room for other processes.
It is important to have efficient memory allocation and deallocation algorithms.



Using Bitmap

Memory is divided into 4KB pages as the unit for allocation and deallocation. A part of memory is used for the bit map keeping the states of memory pages.

- bit i=1 means page i has been allocated
- bit i=0 means page i is free



Allocation:

- Process' memory request is rounded up to k pages
- Search for k zero bits in the bit map
- Reverse the k zero bits to 1s

Deallocation:

- Process terminates, release k pages
- Set the corresponding k bits to 0

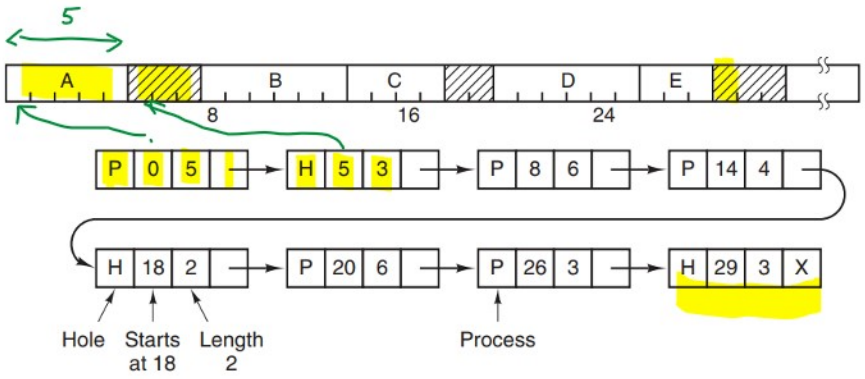
Remarks:

- Require hardware support to accelerate bit operations
- Each page is represented by one bit ; pages with similar properties are managed separately

Using Linked list

Memory is divided into 4KB pages as the unit for allocation and deallocation. A part of memory is used for the linked list managing areas of memory

- A memory area consits of pages with the same state (available / occupied)
- Available areas are labeled as H (Hole), occupied areas are labeled as P (Process)
- A linked list element consists of:
  - Labe of the area
  - Starting page of the area
  - Number of pages
  - Pointer to the next linked list node



Allocation:

- Process' memory request is rounded up to k pages
- Search the linked list for an element with label H and number of pages $\geq k$
- Split the elemenet into 2, one points to the area allocated to the process, and other to the remaining free area

Deallocation:

- Browse through the linked list until an element pointing to the process memory is met

remaining free area

Deallocation:

- Browse through the linked list until an element pointing to the process memory is met
- Change the label from P to H
- Unify the free neighbor areas, if any, to create a bigger area with lable H

O(n)

Remarks:

- Number of linked list elements is actually many times less than the number of pages
- Not so efficient when the memory is highly fragmented

Using buddy blocks of power-of-2 size

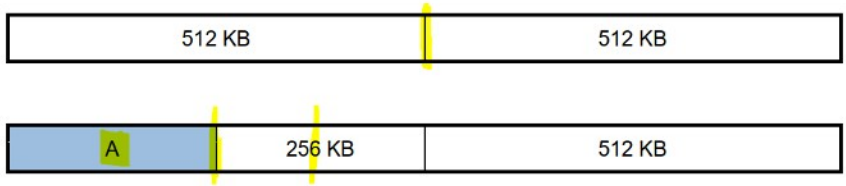
The initial memory must be of size power of 2 (i.e a buddy). Application request shall be rounded up to a buddy.

Allocation: cut a buddy in half until we can find a buddy of the requested size.

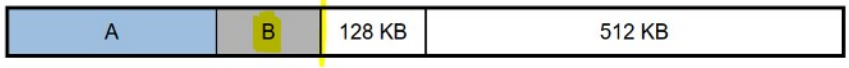
Deallocation: unify neighbor buddies until we get the biggest possible buddy.

Example: Memory of size 1024KB

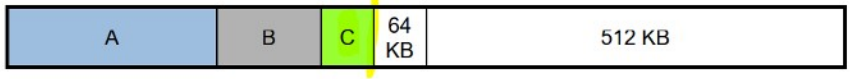
Process A requests 200KB, which is rounded up to 256KB



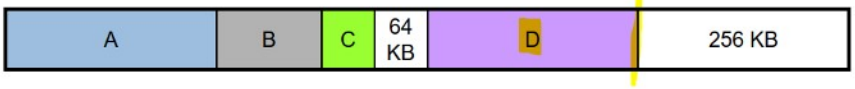
Process B request 120KB, which is rounded up to 128KB



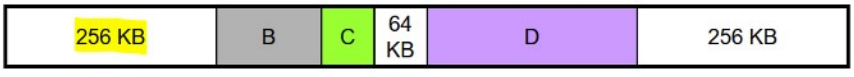
Process C requests 40KB, which is rounded up to 64KB



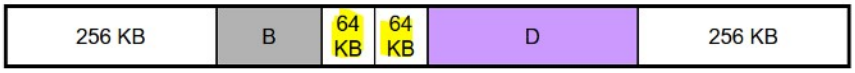
Process D requests 130KB, which is rounded up to 256KB

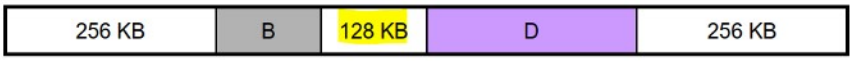


Process A terminates

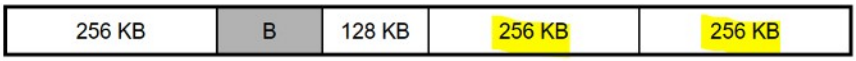


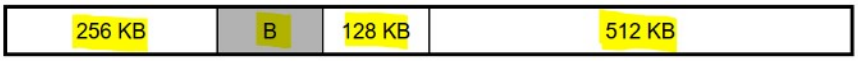
Process C Terminates





Process D terminates





Process B terminates

Remarks:

- Both allocation and deallocation are really fast, with complexity of $log_2 n$
- Drawback: waste of memory due to the power-of-2 size roundup. This method is suitable for allocation on external memory.

Allocation strategies

First Fit: Search from the beginning until the 1st sufficient memory area for the process is met

Next Fit: Continue from the position of previous search, until the 1st sufficient memory area for the process is met

Worst Fit: Search for the largest memory area to allocate to the process

Best Fit: Search for the smallest memory area to allocate to the process

Example:

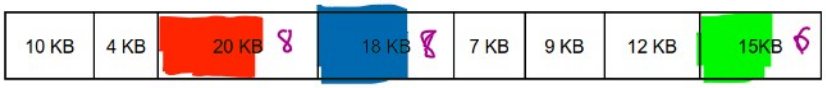
First fit

Next fit

Requests: A – 12 KB B – 10 KB C – 9 KB

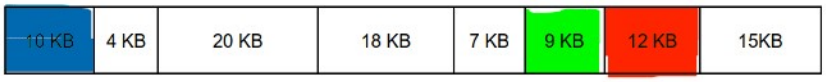
Worst fit

Requests: A – 12 KB B – 10 KB C – 9 KB



Best fit

Requests: A – 12 KB B – 10 KB C – 9 KB



Remarks:

- Best fit: the remaining pieces are too small to be used for other processes
- Worst fit: always cut off big area into small chunks, thus may not be able to allocate to process with large memory requests
- Both best and worst fit require browsing through the whole memory to find the appropriate area
- First fit and next fit only find the fist satisfied area => faster. Next fit yields better results during the initial run.

2.3 Virtual Memory

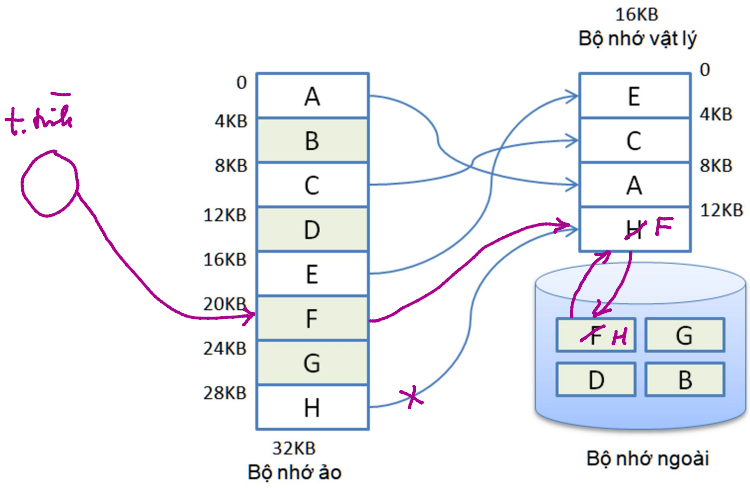
Concept of Virtual Memory

Virtual memory is a logical address space, divided into pages; some pages are directly mapped onto physical memory and others are kept in external storage.

- Content of the pages that currently mapped onto physical memory (RAM) can be accessed directly through an address resolution from virtual to physical address.
- Content of pages in external storage cannot be accessed directly, but must be first loaded into physical memory and then create a mapping relationship between the virtual page and physical page. Loading a new page into physical memory may require page swapping between physical and external memory.

Why virtual memory?
Because virutal memory is much larger than physical memory, which allows

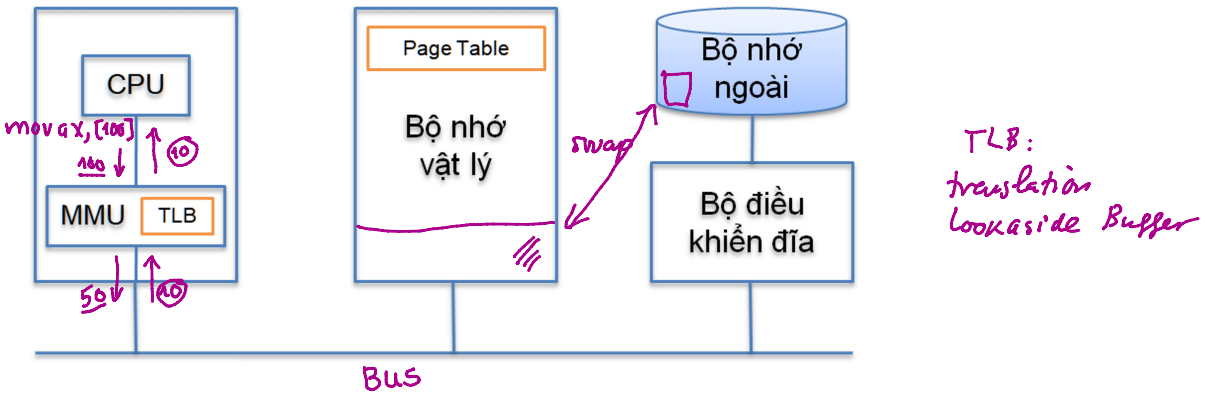
- more working space for processes
- trading off between memory size and system performance



Memory Management Unit - MMU

Implementation of virutal memory requires support of both hardware and software (i.e OS). MMU is a special device standing between CPU and computer bus to carry out the following tasks:

- Address conversion, from virtual to physical.
- Work in line with OS to swap pages between external and physical memory; establish new address mapping relationships.

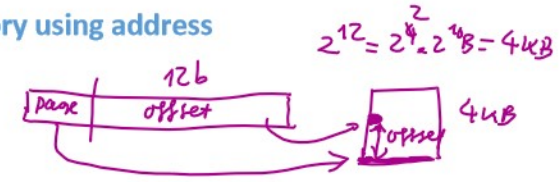


Specifically, the following steps are carried out:

- CPU sends a request to a virtual page which is currently mapped to a physical page. MMU shall do the address conversion to physical address by
  - Search through a cache called Translation Lookaside Buffer (TLB) for the physical address
  - If not found, MMU shall look at Page Table for the address resolution, then update the new mapping relationship into TLB
- In case the requested address is not currently mapped to physical memory, MMU cannot resolute the address, so
  - MMU shall create a Page Fault event and pass to the OS' Page Supervisor module for further processing
  - Page Supervisor shall carry out the page swapping and update the new address relationship into Page Table
  - Page Supervisor then pass the control back to MMU for the address resolution task

More about Page Table, TLB and page swapping algorithms will be discussed in the sections below.

Address conversion from virutal to physical memory using address resolution table



An address can be split into 2 parts:
page address + offset address.

For a memory page of 4KB, the offset address is the last 12 bits and the other bits in front represent the page address.

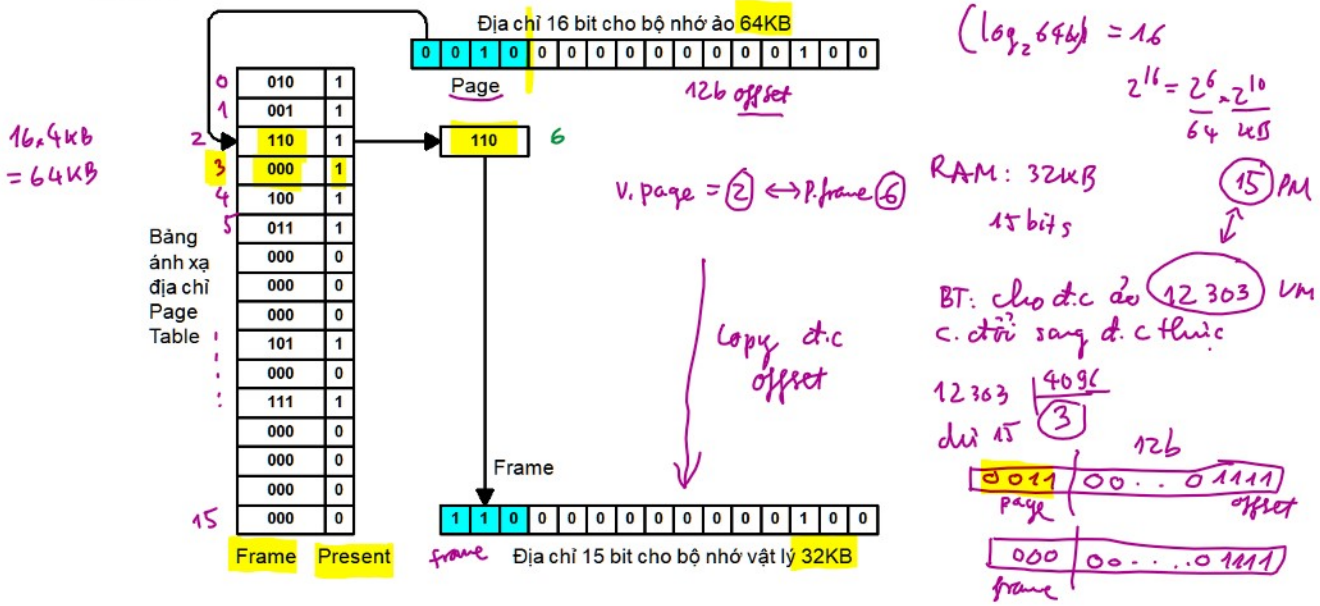
As a virtual page is mapped onto a physical page, the address conversion task actually is just to convert the virtual page address into physical page address, keeping the offset address unchanged.

Naming convention: virtual page is often referred as page, physical page is referred as frame (page frame).

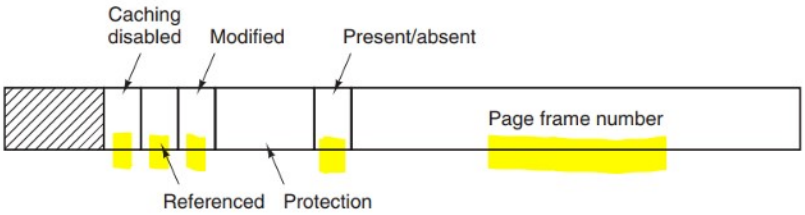
Page Table is an address resolution table, consitings of 2 columns:

- Frame: keeps the corresponding physcial page address
- Present: whether the mapping relationship between virtual and physical pages is still valid; ==1 means Yes; ==0 means No

Example: we have a virtual memory of size 64KB and physical memory of 32KB. Relationships between virtual and physical pages are given in Page Table as follows

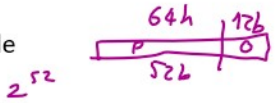


A record in Page Table, in detail



- Page Frame Number: physical page address
- Present/Absent: is the page currently mapped onto physical memory? If not Page Fault event is raised up
- Protection: page access permission (Read, Write, Execute)
- Bit M: Modified. Has page content been modified?
- Bit R: Referenced. Has the page been referred to?
- Caching disabled: CPU caching is not allowed for the page

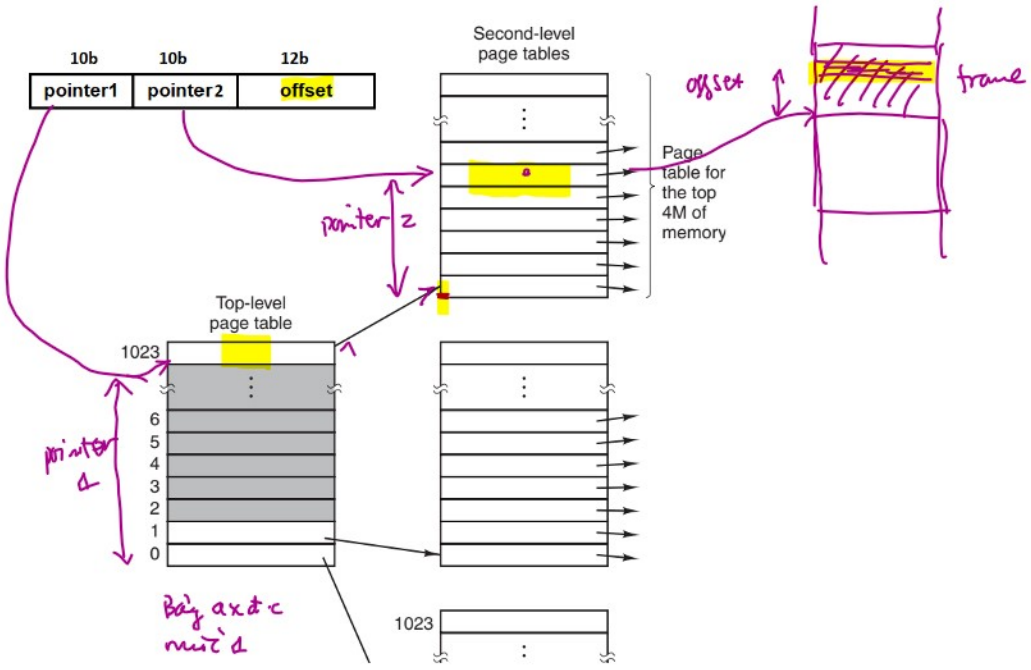
Remarks: If we just use linear addresses for the Page Table, size of the Page Table may be large.

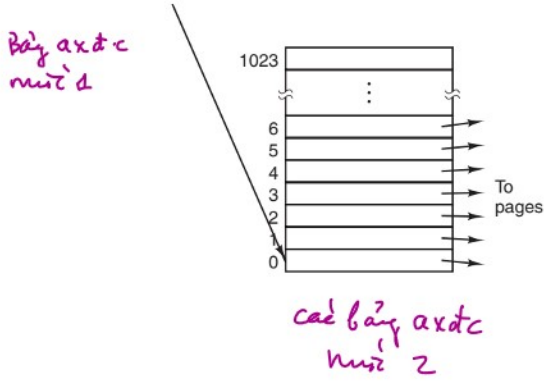


Multiple-level address resultion table

An address is split into multiple levels of page address. Below is an example with a 32-bit address with 2 paging level

- **Pointer 1** points to a row in the address resolution level 1
- Each row in a level-1 table points to a level-2 table
- **Pointer 2** points to a row in a level-2 table
- At the last level, a row of address resolution table points to a physical page frame





Remarks:

- Break a single address resolution table into many smaller tables
- Require multiple steps to resolute an address

Translation Lookaside Buffer / Associative Memory

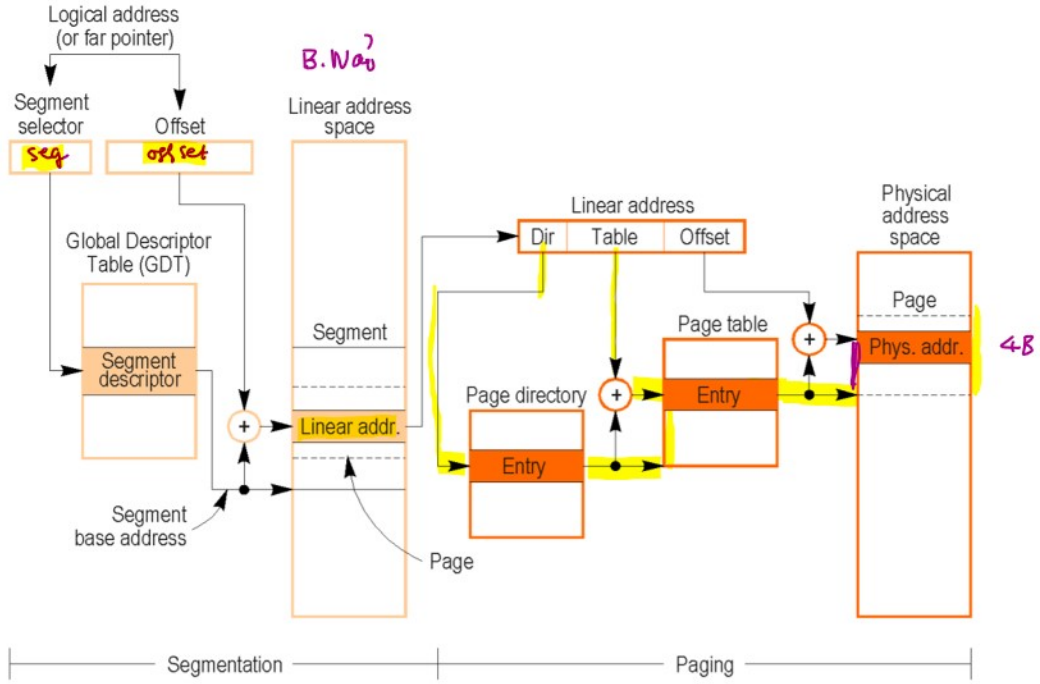
TLB is a cache in MMU consisting of multiple registers which allow searching in parallel for an address relationship between virtual pages and physical page frames. Only the most frequently used addresses are kept in TLB

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

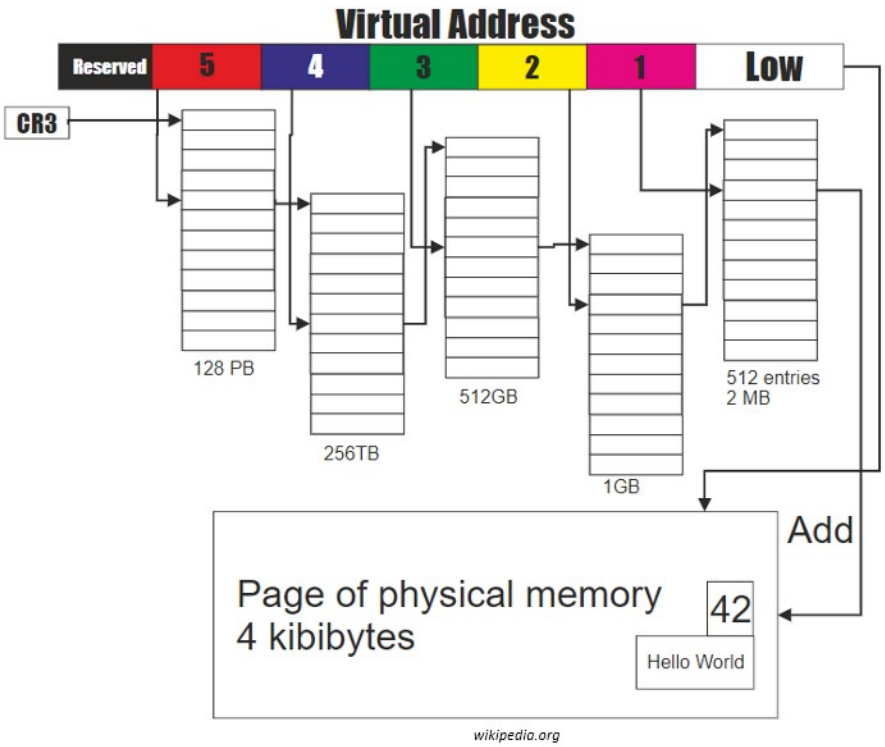
Whenever there is a request for address resolution, MMU shall

- Look for the address relationship in TLB
- If not found, Page Table shall be used. The new address relationship shall be updated into TLB
- If not found in Page Table, a Page Fault event shall be raised to OS to swap the content of a physical frame with a page kept in external memory, then the new address relationship shall be updated to Page Table and control shall be passed back to MMU to carry out the address resolution.

Summary of segmentation and paging in Intel x86



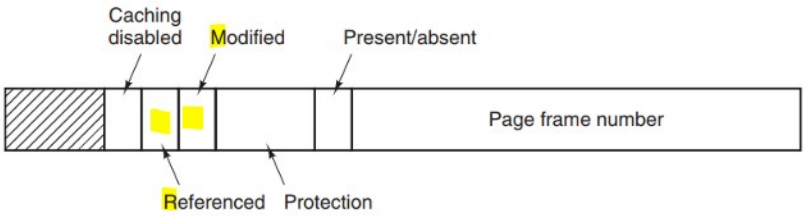
Notes: Starting from 64-bit Intel x86, segmentation is no longer deployed, only supported for backward compatibility. 64-bit Intel x86 uses 5 paging levels



Page swapping algorithms

NRU Algorithm

NRU - Not Recently Used:



Each memory page is attached with 2-bit attributes

- Bit R (Referenced): turned on when the page is referred to
- Bit M (Modified): turned on when the page content has been modified

Pages can be classified into 4 groups of R and M bits:

1. Not Referenced, Not Modified
2. Not Referenced, Modified
3. Referenced, Not Modified
4. Referenced, Modified

Swapping order is group 1, 2, 3, 4; and in each group a random page can be chosen.

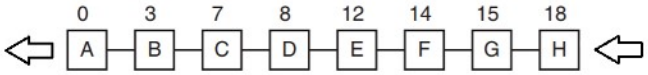
Please note that a page with modified content, but not referred to recently, is rated higher for the swapping than an unmodified page but recently used.

Remarks:

- Only based on 2 attribute bits
- In each group, a random page is picked for page swapping

FIFO algorithm

Pages are sorted in the order of time being loaded into the memory. The oldest page will be chosen for the swapping.



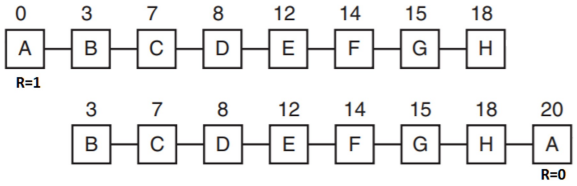
Remarks:

- Pages loaded long ago may be the pages that are used frequently, such as pages belong to system processes

Second-Chance algorithm

Each page is attached with 1 bit attribute R (Referrenced, set to 1 when being

referred to). if the page to be swapped out has bit R=1, it shall has a second chance, which means being put at the end of the queue as a new loaded page with bit R set to 0.

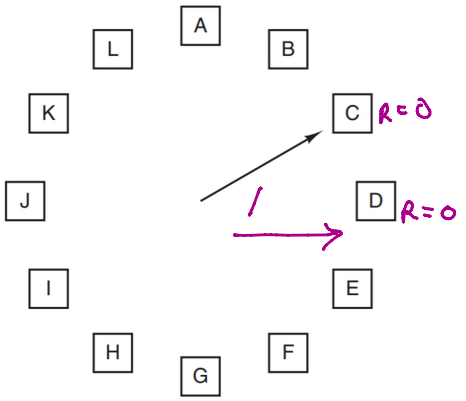


Remarks:

- Bit R is used to avoid swapping frequently used page. However pages with high usage may still be swapped out.
- Need to maintain and manage memory pages as a linked list

CLOCK algorithm

Pages are ordered as a round by page number. Initially, the clock hand points to the 1st loaded page.



Bit R = 0: swap out

Bit R = 1: erase bit R, move to the next page

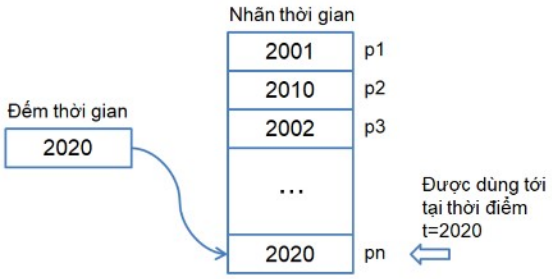
Remarks:

- Don't have to maintain a linked list
- Highly used pages may still be swapped out if not referred to in each round

LRU algorithm with hardware support

LRU - Least Recently Used is the swapping criteria. OS expects that pages which was used long ago will be the pages that are unlikely to be used again soon.

Implementation 1:
OS maintains 64-bit registers to keep time labels. One register keeps the current time t, and other n registers keep the time label of n memory pages. When a page is referred to, its time label will be set to t.
When a Page Fault event occurs, the page with lowest time label, i.e least recently used, will be swapped out.



Implementation 2:
OS maintains an $n \times n$ bit matrix, where n is the number of pages.
When page i is referred to, the matrix is updated as follows

- Turn on all bits of row i
- Turn off all bits of column i

At the time of a Page Fault event, the page with lowest bit value by row will be swapped out.

0	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

Page 0 is used

0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

Page 1 is used

0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

Page 2 is used

Remarks:

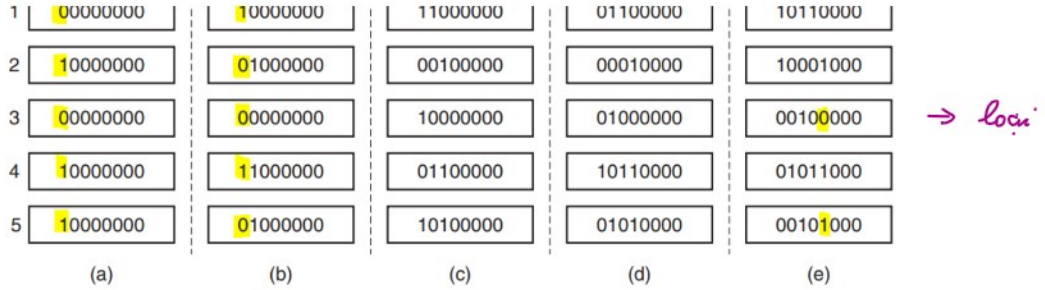
- Beside bits R and M in each record of Page Table, additional storage is required to keep time labels of each page
- Hardware support is needed for bit operations on the matrix

Aging algorithm using software

Aging is used to simulate LRU but using software.

Each page is attached with an l bit register keeping the R bit of l most recent clock ticks.
Initially, all registers are set to 0.
After each clock tick, all registers are shifted right 1 bit and the left most bit is set to the R bit of the corresponding page.
A the time of a Page Fault event, the page with lowest value in the register will be swapped out.

Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
	$l = 8$				
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000



Remarks:

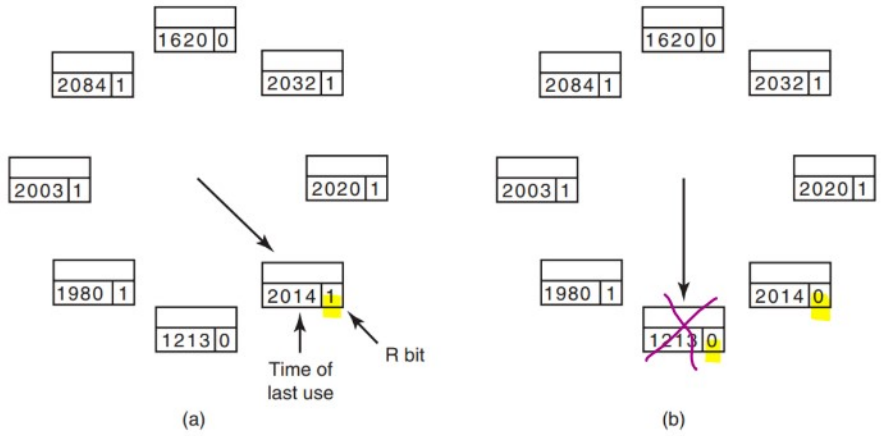
- Aging allows to remember the history of at most l previous clock ticks
- Has to maintains a register for each page
- Pretty high cost: shifting all registers, set the left most bit; find the lowest value register

WS-CLOCK algorithm

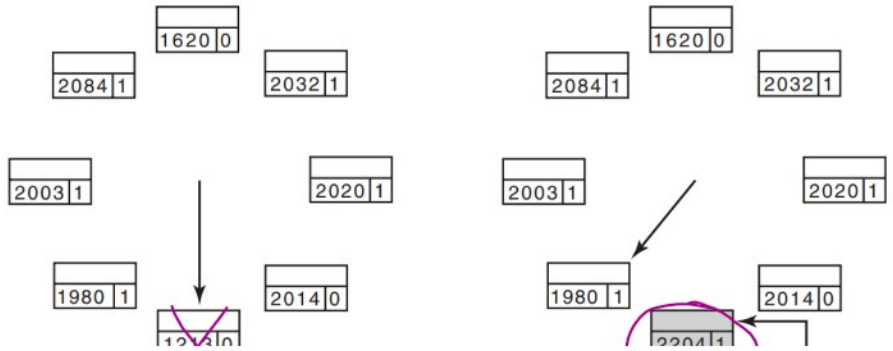
Working Set-CLOCK is a modified version of CLOCK algorithm with a τ time limit such as all processes that are used within time $\leq \tau$ (belongs to the working set) will not be swapped out. Each page will be attached with 2 attribute bits R and M.

Example: assume that the current time is 2204, $\tau = 100$. Pages with time labels equal or greater than 2104 will not be swapped out by the algorithm, the clock hand will move to the next page instead.

Case 1, the page pointed to by the clock hand has bit R=1, CLOCK algorithm will erase bit R and move to the next page.



Case 2, the current page has bit R=0 and its time label does not belong to the working set τ, then the page will be swapped out, bit R of the newly loaded page will be set to 1 and the clock hand will move to the next page.



Problems with bit M

- If the current page has bit M=1, we need to write down the page content to external memory before the swapping. As it takes time, the write request will be placed in a queue, meanwhile the clock hand will move to the next page.
- In case none of the candidates found, the clock hands loops back one round, then
  - If there are pages queuing for the write request, soon or later a request will be fulfilled and that page will be swapped out
  - Otherwise, we pick a page with bit M=0 at random for the swapping

Remarks:

- Low running cost, feasible for software implementation
- However we need to maintain a time label for each memory page