MODERN OPERATING SYSTEMS

ANDREW S. TANENBAUM

Chapter 3
Memory Management

1

Memory Management

Basic memory management
Swapping
Virtual memory
Design issues for paging systems
Implementation issues
Segmentation

2

2

Memory Management

- Memory (RAM) is an important and rare resource
  - Programs expand to fill the memory available to them
- Programmer's view
  - Memory should be private, infinitely large, infinitely fast, nonvolatile…
- Reality
  - Best of people's mind: memory hierarchy
  - Register, cache, memory, disk, tape
- Memory manager
  - Efficiently manage memory
  - Keep track the free memory, allocate memory to programs…

3

Memory management

- The memory management in this chapter ranges from very simple to highly sophisticated…

4

No Memory Abstraction

- Early mainframe, early minicomputers, early personal computers had no memory abstraction…
  - MOV REGISTER1, 1000
  - Here 1000 means move the content of physical memory address1000 to register
- Impossible to have two programs in memory

5

No Memory Abstraction

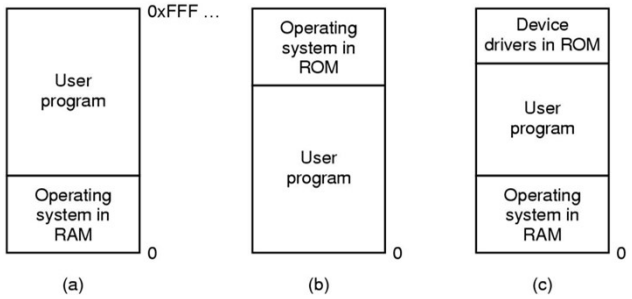


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.

6

Multiple problems without abstraction

- IBM 360
  - Memory divided into 2-KB blocks and each one with a 4-bit protection key
  - PSW also has a 4-bit protection key
  - Hardware will trap any attempt tries to access memory with a protection code different from PSW key

7

Multiple Programs Without Memory Abstraction: Drawback (nhược điểm)

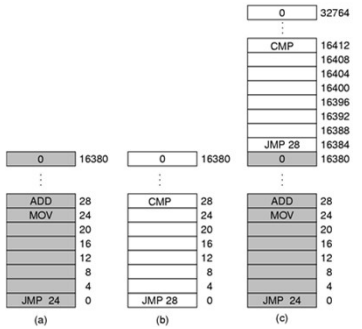


Figure 3-2. Illustration (minh họa) of the relocation problem.

8

Drawback of no abstraction

- Problem is that both programs reference absolute physical memory
- People hope that they can have a private space, that is, addresses local to it
- IBM 360
  - Modify the second program on the fly (nhanh chóng) as it loaded it into memory
  - Static relocation
    - When a program is loaded into 16384, then the constant is added to every address
    - Slow down loading, needs extra information
- No abstraction memory still used in embedded and smart systems

9

Abstraction: address space

- Not to expose (phơi bày) physical address to programmers
  - Crash (phá vỡ) OS
  - Hard to parallelize
- Two problems to solve:
  - Protection
  - Relocation
- Address space:
  - A set of memory processes can use to address memory
  - Each process has its own address space, independent of each other
  - How?

10

Dynamic relocation

- Equip CPU with two special register: base and limit
  - Program be loaded into a consecutive (liên tiếp, liên tục) space
  - No relocation during loading
  - When process is run and reference an address, CPU automatically adds the base to that address; as well as check whether it exceeds (vượt quá) the limit register

11

Base and Limit Registers

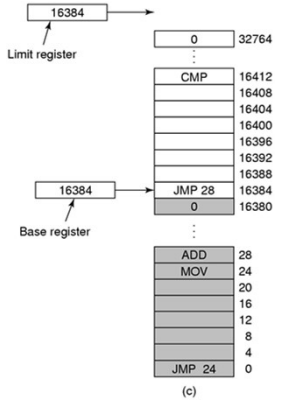
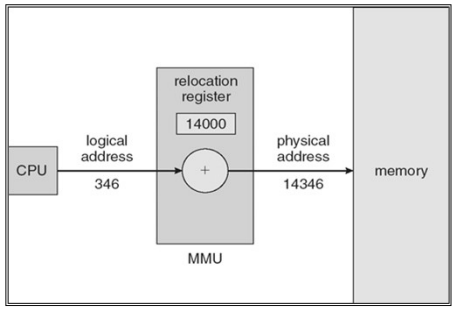


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

12

Dynamic relocation using a relocation register



13

13

Basic Memory Management
Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Use base and limit values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error

14

14

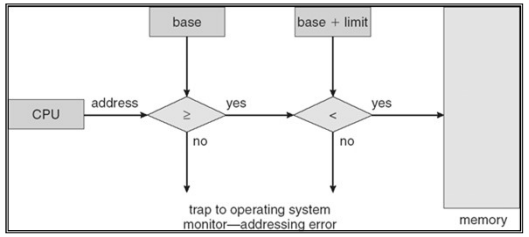
Relocation and Protection

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

15

15

Basic Memory Management
HW address protection with base and limit registers



16

16

Base and Limit Registers

- Disadvantage:
  - Need to perform an addition and a comparison on every memory reference

17

Swapping

- Many background server processes run in the system
- Physical memory not large enough to hold all programs
  - Swapping
    - Bring in and swap out programs
  - Virtual memory
    - Run programs even (ngay cả) when they are partially in memory

18

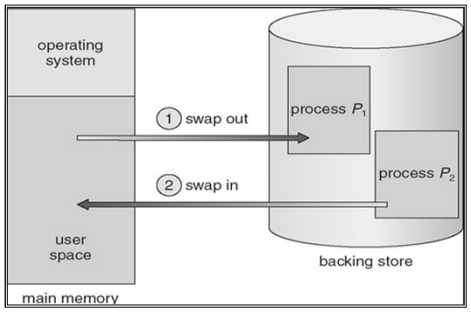
Swapping

- Problems
  - Addresses different as swaps in and out
    - Static relocation/dynamic relocation
  - Memory holes
    - Memory compaction
      - Require CPU time
      - Move 4 byes in 20ns, then 5 sec to compact 1 GB
  - How much memory allocate for a program
    - Programs tend (có xu hướng) to grow
    - Both data segment (heap) and stack

19

Swapping (1)

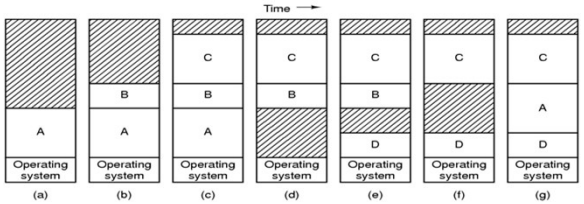
Schematic View of Swapping



20

20

Swapping (2)

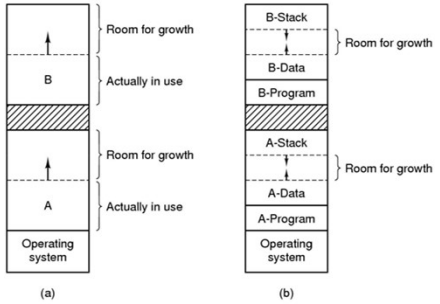


- Memory allocation changes as
  - processes come into memory
  - leave memory
- Shaded regions are unused memory
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

21

21

Swapping (3)



- (a) Allocating space for growing data segment
- (b) Allocating space for growing stack & data segment

22

22

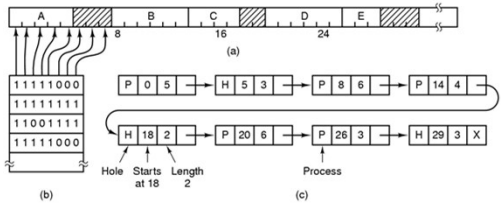
Swapping (4)
Multiple-partition allocation

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about: a) allocated partitions b) free partitions (hole)
  - There are two ways to keep track of memory usages
    - Memory Management with Bit Maps
    - Memory Management with Linked Lists

23

23

Swapping (4)
Multiple-partition allocation



Memory Management with Bit Maps

- (a) Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- (b) Corresponding bit map
- (c) Same information as a list

24

24

Swapping (5)

Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Next fit**: Start seaching the list from the place where it left off last time
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Managing free memory

- Bit maps and linked lists
- Bitmap
  - Memory is divided into allocation units (a few word to KB)
  - Corresponding to each unit, there is a bit in the bitmap
  - Hard to find a given length free space

Memory Management with Bitmaps

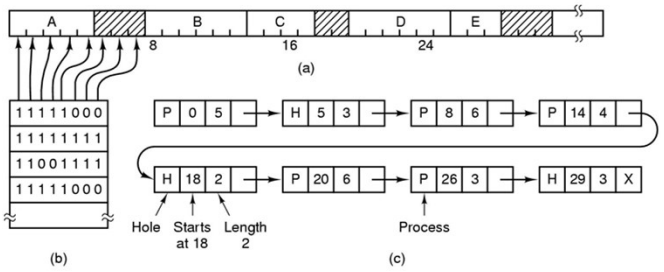


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists

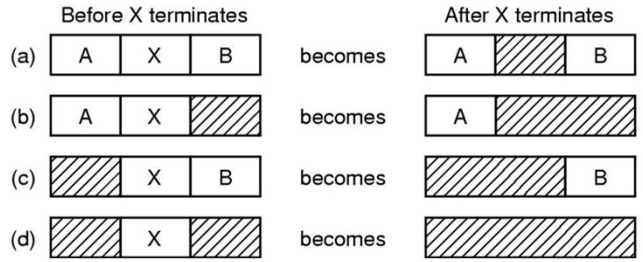


Figure 3-7. Four neighbor combinations for the terminating process, X.

Manage free memory

- Linked list
  - Double linked list
  - How to allocate free memory to programs?
    - First fit
      - quick; beginning used more often; break a large free space
    - Next fit
      - Every time from where last time used
    - Best fit
      - Search entire list, finds the hole close to the actual size
    - Worst fit
      - Finds the largest hole
    - Quick fit
      - Keeps separate queues of processes and holes

29

exercise

- In a swapping system, memory consists of the following hole sizes in memory order: 10KB，4KB，20KB，18KB，7KB，9KB，12KB and 15KB. Which hold is taken for successive segment requests of
  - 12KB
  - 10KB
  - 9KB

  For first fit? Best fit? Worst fit? And next fit?

30

Virtual Memory

- Manage bloat ware
  - Where programs are too big to fit into memory
  - Being split by programs is a bad idea (overlays)
  - Virtual memory
    - Every program has its own address space
    - The address space is divided into chunks called **pages**
    - Each page is a contiguous area and mapped to physical address
    - But, not all pages are needed to in physical memory
    - OS maps page addresses and physical addresses on the fly
    - When a needed page not in memory, OS needs to get it in
    - Every page needs relocation

31

Virtual Memory – Paging (1)

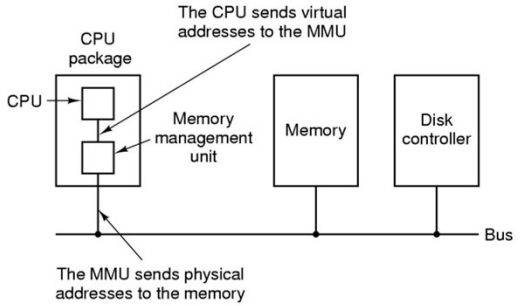


Figure 3-8. The position and function of the MMU – shown as being a part of the CPU chip (it commonly is nowadays). Logically it could be a separate chip, was in years gone by.

32

Paging (2)

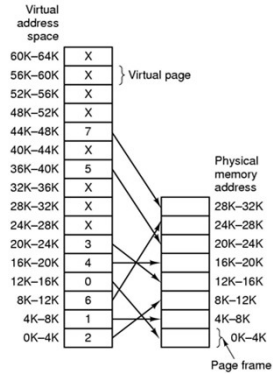


Figure 3-9. Relation between virtual addresses and physical memory addresses given by page table.

33

Paging

- MMU(memory management unit)
  - CPU: MOV REG, 0
  - MMU: MOV REG, 8192
  - CPU: MOV REG 8192
  - MMU: MOV REG 24576
  - CPU:MOV REG 20500
  - MMU:MOV REG 12308
  - CPU: MOV REG 32780
  - MMU: page fault

34

Paging (3)

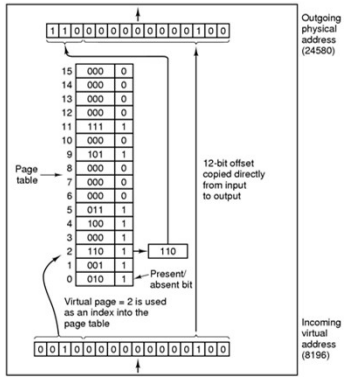


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

35

Page tables

- Virtual addresses mapping
  - Virtual address split into virtual page number and offset
  - 16-bit address: 4KB page size; 16 pages
  - Virtual page number: index to the page table
  - Purpose of page table
    - Map virtual pages onto page frames

36

Structure of Page Table Entry

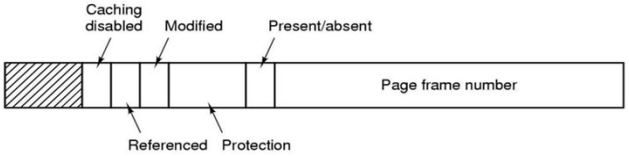


Figure 3-11. A typical page table entry.

37

Page Table Structure

- Protection
  - What kinds of access are permitted
- Modified:
  - When a page is written to (dirty)
- Referenced:
  - When a page is referenced
- Cache disabling
  - Data inconsistency

38

Speeding Up Paging

Paging implementation issues:

- The mapping from virtual address to physical address must be fast.
- If the virtual address space is large, the page table will be large. (32bit/64bit)
- Every process should have its own page table in memory

39

Speeding up paging

- To keep the page table in register?
  - No more memory access needed during process execution
  - But unbearably expensive
- To keep the page table entirely in memory?
  - Each process has its own page table
  - Page table is kept in memory
  - How many memory access needed to perform a logical memory access?

40

Speed up paging

- Effective memory-access time,
  time needed for every data/instruction access
  - Two time memory-access time; reduces performance by half
  - Access the page table & Access the data/instruction
- Solution:
  - A special fast-lookup hardware cache called **associative registers** or **translation look-aside buffers (TLBs)**

41

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

Figure 3-12. A TLB to speed up paging.

42

TLB

- TLB is usually inside MMU and consists of a small number of entries
- When received a virtual address
  - MMU first check to see if its virtual pager number is in TLB;
  - if it's, there is no need to visit page table; if not, evict one entry from TLB and replaces it with the page table entry

43

Effective Access Time

- Associative Lookup = ε time unit;
  memory cycle time = t time unit;
  Hit ratio = α
- Effective Access Time (EAT)

  $EAT = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$

  $= 2t + \varepsilon - t\alpha$
- If ε(20 ns), t(100 ns), α1(80%), α2(98%):
  - TLB hit: 20+100=120 ns
  - TLB miss: 20+100+100=220 ns
  - EAT1 = 120*0.8 + 220 * 0.2 = 140 ns
  - EAT2 = 120*0.98 + 220 * 0.02 = 122 ns

44

Page Table for Large Memory

- Address space: 32bit
- Page size: 4KB
- Page Numbers: 20bit, 1M pages
- 32bit per page entry, then needs 4MB to store a page table
- Not to mention 64bit system

45

Multilevel page table

- 32bit virtual memory divided into three parts
  - 10bit PT1, 10bit PT2, 12bit offset
- Multilevel page table
  - Not to have all page tables in memory all the time
  - Page tables are stored in pages, too
  - Example: a program has 4G address space, it needs 12M to run: 4M code; 4M data; 4M stack

46

Multilevel Page Tables

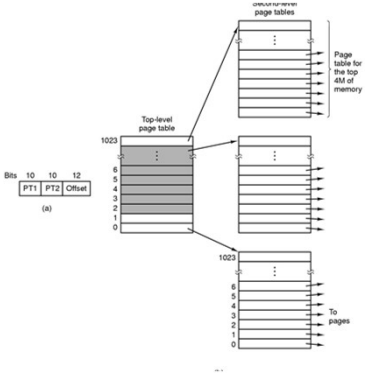


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

47

?

- A computer has 32-bit virtual addresses and 4-KB pages. The program and data together fit in the lowest page (0-4095) the stack fits in the highest page. How many entries are needed in the page table if traditional paging is used? How many page table entries are needed for 2-level paging, with 10 bits in each part?

48

Inverted Page Table

- When virtual address spaces is much larger than physical memory
- Inverted page table: entry per page frame rather than per page of virtual address space
- Search is much harder
  - TLB
  - Hash
- Inverted page table is common on 64-bit machines

49

Inverted Page Tables

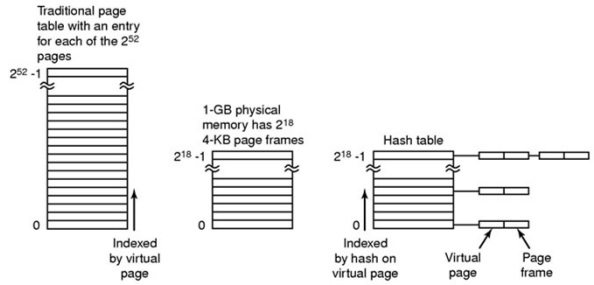


Figure 3-14. Comparison of a traditional page table with an inverted page table.

50

?

- Array A[1024, 1024] of integer, each row is stored in one page
- Program 1

```
for j := 1 to 1024 do
        for i := 1 to 1024 do
                A[i,j] := 0;
```

  - 1024 × 1024 page faults
- Program 2

```
for i := 1 to 1024 do
        for j := 1 to 1024 do
                A[i,j] := 0;
```

  - 1024 page faults

51

Page 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

52

Impact (tác động) of page fault

Page fault time = 25ms
ma = 100ns
With page miss rate as p:

EAT = 100(1-p)+25×10^6×p
= 100 + 24,999,900p

If p=1/1000, then EAT =25,099.9 ns
If needs EAT <110ns, then 100+24,999,900p<110
that is p <10/24,999,900<10/25,000,000
=1/2,500,000
=4×10^{-7}
Page fault rage p must be smaller than 4×10^{-7}

53

Page replacement

- When a page fault occurs, some page must be evicted from memory
- If the evicted page has been modified while in memory, it has to be write back to disk
- If a heavily used page is moved out, it is probably that it will be brought back in a short time
- How to choose one page to replace?

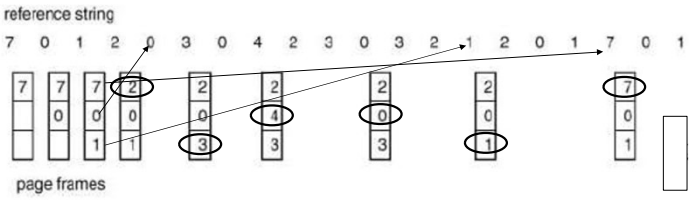
54

Optimal Page replacement

- Easy to describe but impossible to implement
  - Each page be labelled with the number of instructions that will be executed before that page is first referenced
  - The page with the highest label be removed
  - OS won't know when each page will be reference next
  - Be used to compare the performance of realizable algorithms

55

Optimal Page replacement



Page fault 9 times，replacement 6 times

56

Not Recently Used

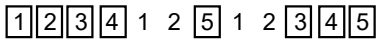
- Replacing based on page usages
- Pages have R and M bit
  - When a process is started up, both page bits are set to 0, periodically, R bit is cleared
  - Four classes can be formed
    - Class 0: not referenced, not modified
    - Class 1: not referenced, modified
    - Class 2: referenced, not modified
    - Class 3: referenced, modifed

C	R	M
0	0	0
1	0	1
2	1	0
3	1	1

57

Not Recently Used (NRU)

- Example
  - 4 page frames
  - Reference string



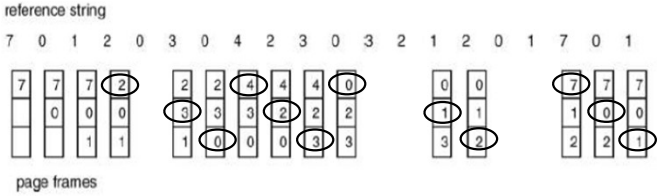
  - 8 page faults
- Pros
  - Implementable
- Cons
  - Require scanning through reference bits and modified bits

58

58

First-In-First-Out (FIFO) Algorithm

- Replace **"the oldest one"**
- Simple，with unsatisfactory (THƯỜNG)performance

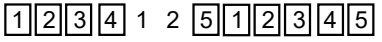


Page fault 15，replacement 12

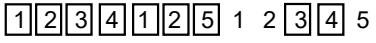
59

More Frames → Fewer Page Faults? Belady's Anomaly

- Consider the following with 4 page frames
  - Algorithm: FIFO replacement
  - Reference string:
  - 10 page faults



- Same string with 3 page frames
  - Algorithm: FIFO replacement
  - Reference string:
  - **9 page faults!**

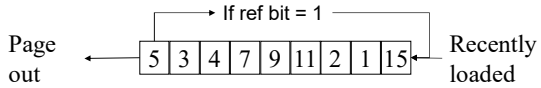


- This is so called "Belady's anomaly" (Belady, Nelson, Shedler 1969)

60

60

FIFO with 2nd Chance



- Algorithm
  - Check the reference-bit of the oldest page
  - If it is 0, then replace it
  - If it is 1, clear the referent-bit, put it to the end of the list (as if it had just been loaded), and continue searching
- Example
  - 4 page frames
  - Reference string:
  - 8 page faults

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

- Pros
  - Simple to implement
- Cons
  - The worst case may take a long time (moving pages around on the list)

61

61

Clock

- 2nd chance is inefficient:
  - It constantly (luôn luôn) moves pages around in its list
- An alternative (sự lựa chọn):
  - To keep all the page frames on a circular list in the form of a clock

62

The Clock Page Replacement Algorithm

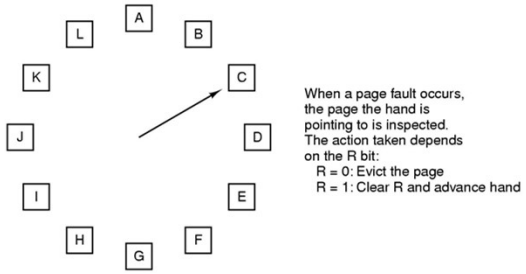
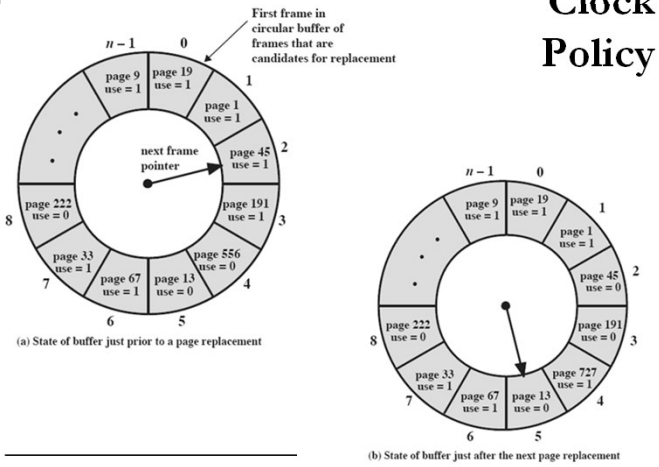


Figure 3-16. The clock page replacement algorithm.

63

Clock Policy



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

64

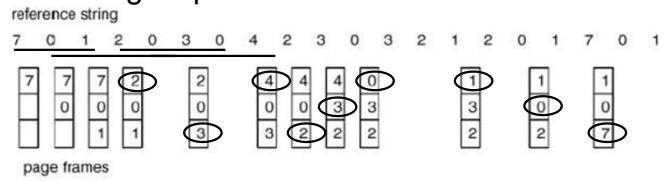
LRU

- An approximation to the optimal algorithm:
  - Pages have not been used for ages will probably remain unused for a long time
  - When a page fault occurs, throw out the page that has been unused for the longest time
- to implement LRU
  - To record time or use a linked list of all pages in memory;
  - Or use hardware counter or a matrix

65

LRU algorithm

- LRU = Least Recently Used
- Replace the page that has not been used for the longest period of time.



Page fault: 12
Replacement: 9

66

Ex.

- If FIFO page replacement is used with four page frames and ten pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

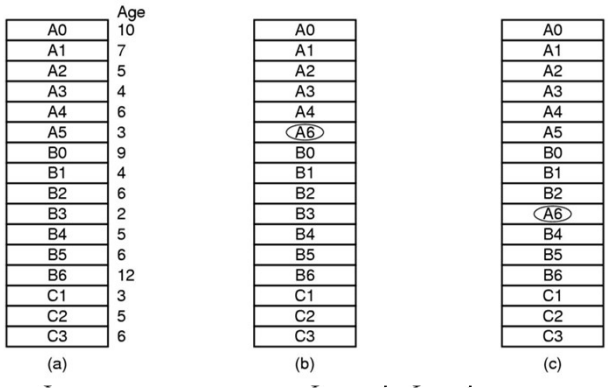
67

Page Replacement Algorithm Summary

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
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

68

Local versus Global Allocation Policies (1)



(a) Original configuration. (b) Local page replacement. (c) Global page replacement.

69

Local versus Global Allocation Policies (2)

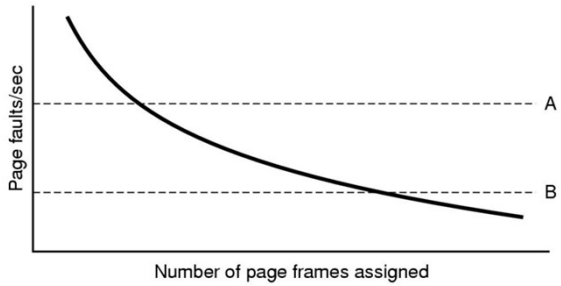


Figure 3-24. Page fault rate as a function of the number of page frames assigned.

70

Separate Instruction and Data Spaces

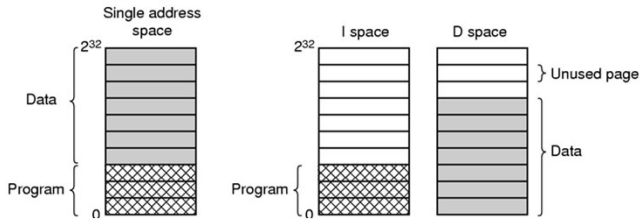


Figure 3-25. (a) One address space. (b) Separate I and D spaces.

71

Shared Pages

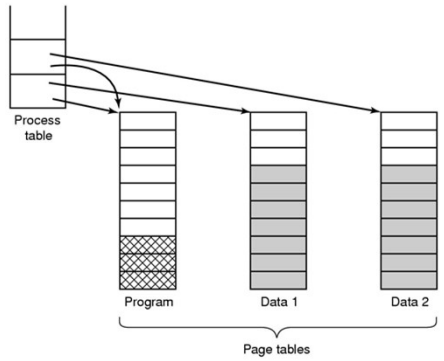


Figure 3-26. Two processes sharing the same program sharing its page table.

72

Shared Libraries

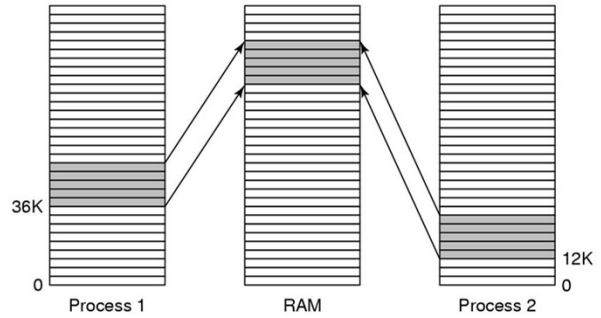


Figure 3-27. A shared library being used by two processes.

73

Page Fault Handling (1)

- The hardware traps to the kernel, saving the program counter on the stack.
- An assembly code routine is started to save the general registers and other volatile information.
- The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed.
- 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

74

Page Fault Handling (2)

- If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
- 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.
- When disk interrupt indicates page has arrived, page tables updated to reflect position, frame marked as being in normal state.

75

Page Fault Handling (3)

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

76

Instruction Backup

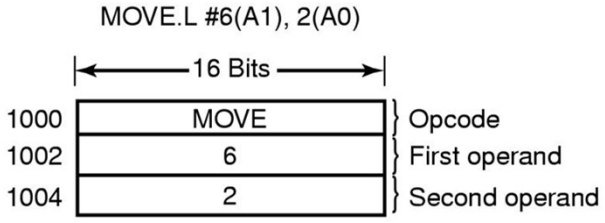


Figure 3-28. An instruction causing a page fault.

77

Backing Store (1)

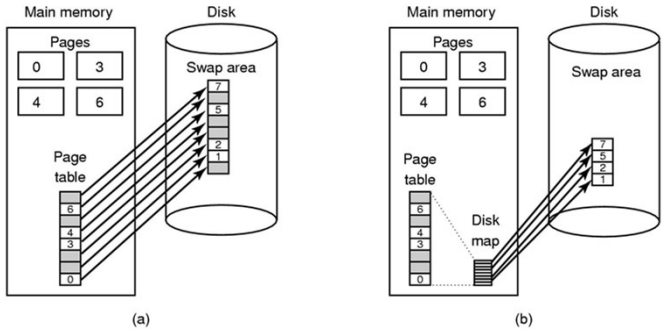


Figure 3-29. (a) Paging to a static swap area.

78

Backing Store (2)

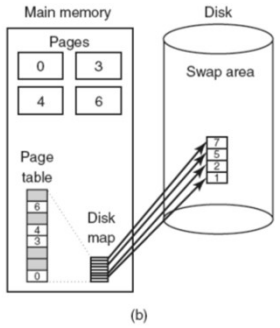


Figure 3-29. (b) Backing up pages dynamically.

79

Separation of Policy and Mechanism (1)

Memory management system is divided into three parts:

- A low-level MMU handler.
- A page fault handler that is part of the kernel.
- An external pager running in user space.

80

Separation of Policy and Mechanism (2)

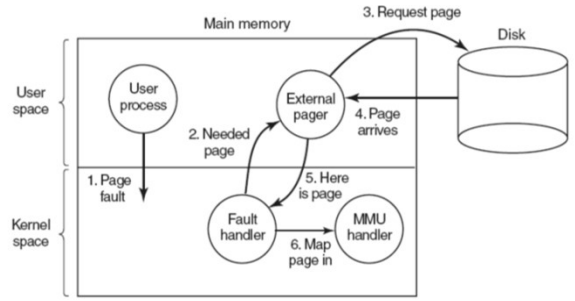


Figure 3-30. Page fault handling with an external pager.

81

Segmentation

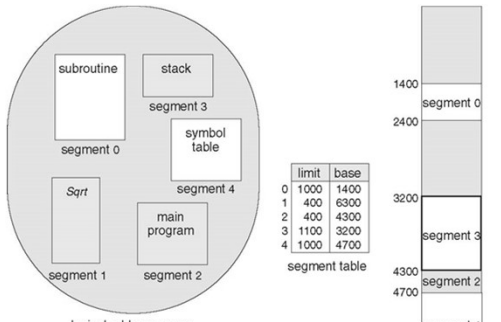
- Segmentation allows the programmer to view memory as consisting of multiple address spaces or **segments**

Advantages:

- simplifies handling of growing data structures
- allows programs to be altered and recompiled independently
- Lends (thích hợp) itself to **sharing** data among processes
- lends itself to **protection**

82

Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

83

Segment Organization

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

84

Address Translation

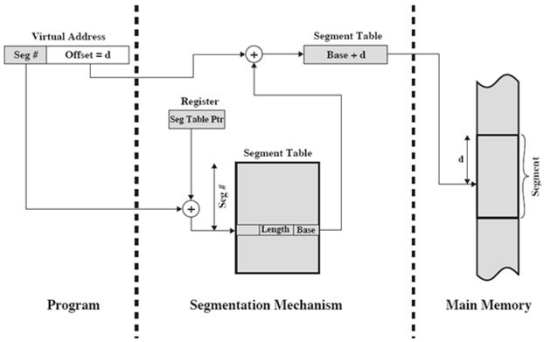


Figure 8.12 Address Translation in a Segmentation System

85

Combined Paging and Segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

- Segmentation is visible to the programmer
- Paging is transparent to the programmer

86

Address Translation

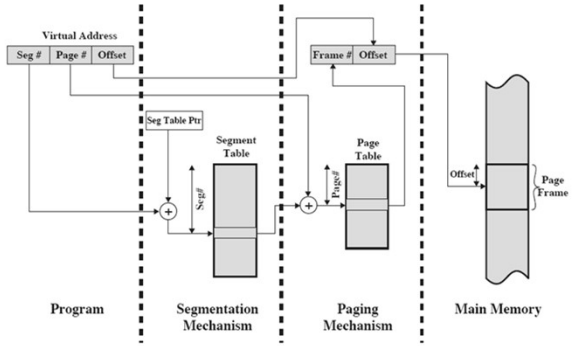
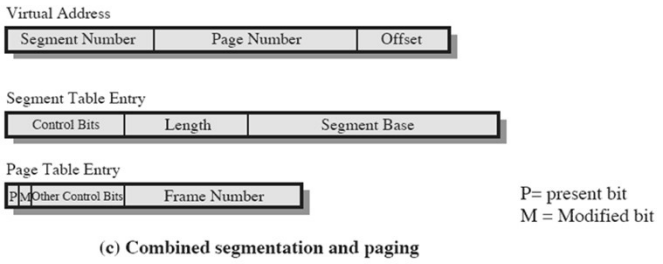


Figure 8.13 Address Translation in a Segmentation/Paging System

87

Combined Segmentation and Paging



(c) Combined segmentation and paging

88

Segmentation with Paging: MULTICS (1)

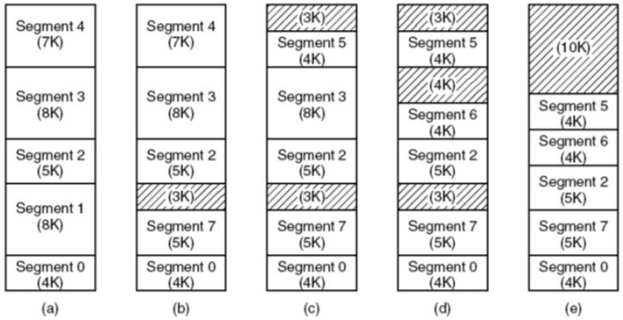


Figure 3-34. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

89

Segmentation with Paging: MULTICS (2)

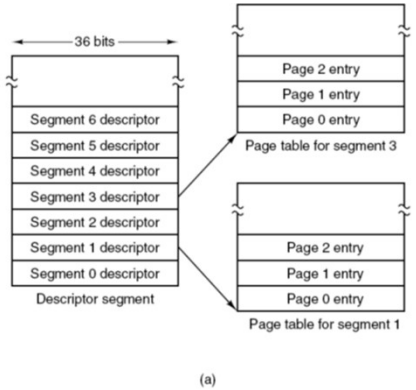


Figure 3-35. The MULTICS virtual memory. (a) The descriptor segment points to the page tables.

90

Segmentation with Paging: MULTICS (5)

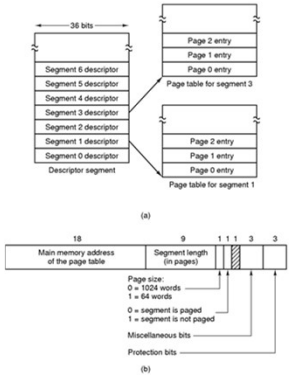


Figure 3-35. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

91

Segmentation with Paging: MULTICS (6)

When a memory reference occurs, the following algorithm is carried out:

- The segment number used to find segment descriptor.
- Check is made to see if the segment's page table is in memory.
  - If not, segment fault occurs.
  - If there is a protection violation, a fault (trap) occurs.

92

Segmentation with Paging: MULTICS (7)

- Page table entry for the requested virtual page examined.
  - If the page itself is not in memory, a page fault is triggered.
  - If it is in memory, the main memory address of the start of the page is extracted from the page table entry
- The offset is added to the page origin to give the main memory address where the word is located.
- The read or store finally takes place.

93

Segmentation with Paging: MULTICS (8)

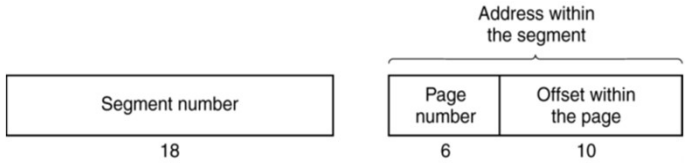


Figure 3-36. A 34-bit MULTICS virtual address.

94

Segmentation with Paging: MULTICS (9)

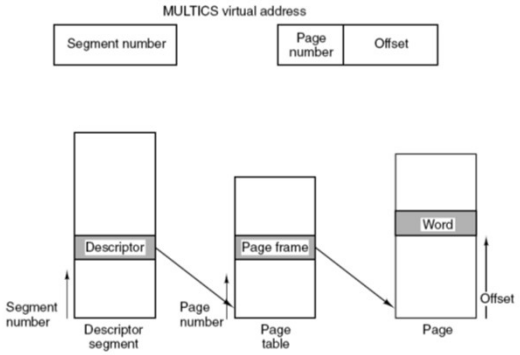


Figure 3-37. Conversion of a two-part MULTICS address into a main memory address.

95

Segmentation with Paging: MULTICS (10)

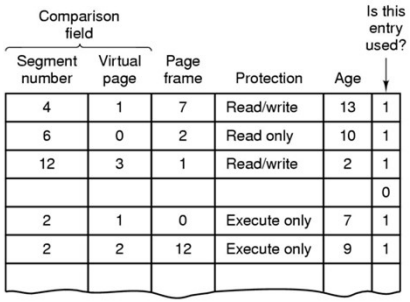


Figure 3-38. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

96

Segmentation with Paging: The Pentium (1)

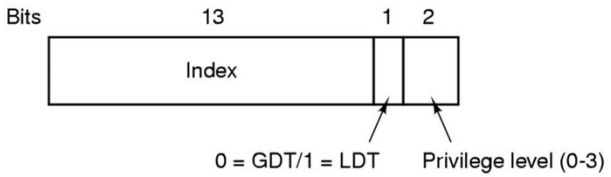


Figure 3-39. A Pentium selector.

97

Segmentation with Paging: The Pentium (2)

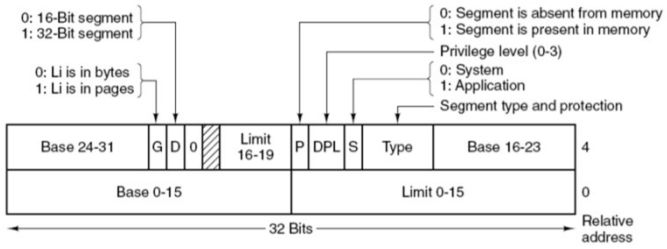


Figure 3-40. Pentium code segment descriptor. Data segments differ slightly.

98

Segmentation with Paging: The Pentium (3)

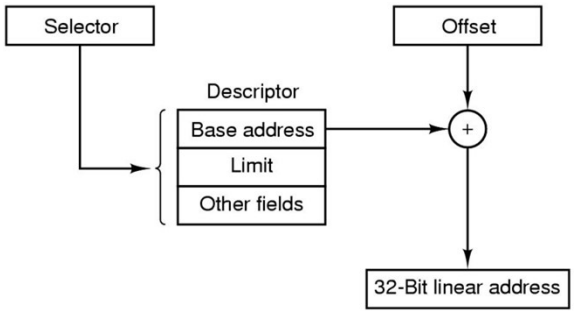


Figure 3-41. Conversion of a (selector, offset) pair to a linear address.

99

Segmentation with Paging: The Pentium (4)

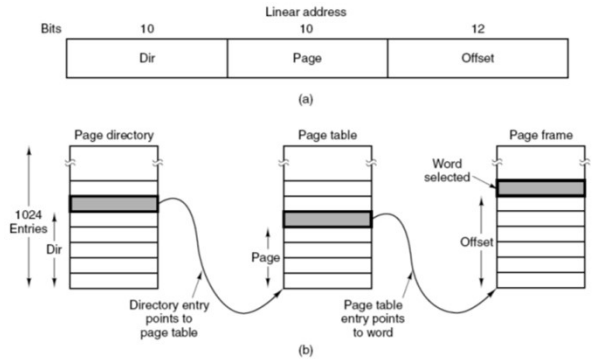


Figure 3-42. Mapping of a linear address onto a physical address.

100

Segmentation with Paging: The Pentium (5)

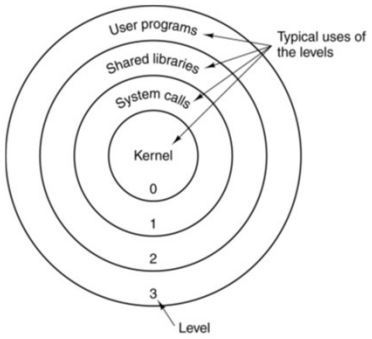


Figure 3-43. Protection on the Pentium.

101

Linux Memory Management

- Shares many characteristics with Unix
- Is quite complex

Two main aspects

- process virtual memory
- kernel memory allocation

102

Linux Virtual Memory

- Three level page table structure:

Page directory	Page middle directory	Page table
process has a single page directory	may span multiple pages	may also span multiple pages
each entry points to one page of the page middle directory	each entry points to one page in the page table	each entry refers to one virtual page of the process
must be in main memory for an active process		

103

Address Translation

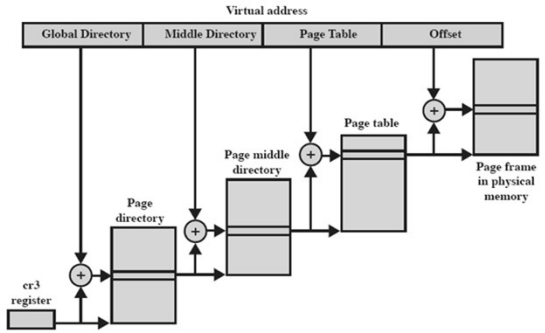


Figure 8.25 Address Translation in Linux Virtual Memory Scheme

104

Linux Page Replacement

- Based on the clock algorithm
- The use bit is replaced with an 8-bit age variable
  - incremented each time the page is accessed
- Periodically decrements the age bits
  - a page with an age of 0 is an “old” page that has not been referenced is some time and is the best candidate for replacement
- A form of least frequently used policy

105

Kernel Memory Allocation

- Kernel memory capability manages physical main memory page frames
  - primary function is to allocate and deallocate frames for particular uses

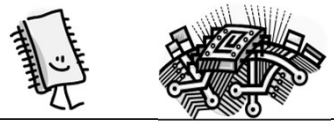
Possible owners of a frame include:
• user-space processes • dynamically allocated kernel data • static kernel code • page cache

- A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages
- Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes
- Slab allocation
  - used by Linux to accommodate small chunks

106

Windows Memory Management

- Virtual memory manager controls how memory is allocated and how paging is performed
- Designed to operate over a variety of platforms
- Uses page sizes ranging from 4 Kbytes to 64 Kbytes

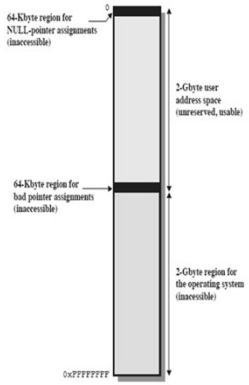


107

Windows Virtual Address Map

- On 32 bit platforms each user process sees a separate 32 bit address space allowing 4 Gbytes of virtual memory per process
  - by default half is reserved for the OS
- Large memory intensive applications run more effectively using 64-bit Windows
- Most modern PCs use the AMD64 processor architecture which is capable of running as either a 32-bit or 64-bit system

108



32-Bit Windows Address Space

Figure 8.26 Windows Default 32-bit Virtual Address Space

109

Windows Paging

- On creation, a process can make use of the entire user space of almost 2 Gbytes
- This space is divided into fixed-size pages managed in contiguous regions allocated on 64 Kbyte boundaries
- Regions may be in one of three states:

available reserved committed

110

Resident Set Management System

- Windows uses variable allocation, local scope
- When activated, a process is assigned a data structure to manage its working set
- Working sets of active processes are adjusted depending on the availability of main memory

111

Summary

- Memory management requirements
  - relocation
  - protection
  - sharing
  - logical organization
  - physical organization
- Paging
- Memory partitioning
  - fixed partitioning
  - dynamic partitioning
  - buddy system
  - relocation
- Segmentation

112

Summary

- Desirable (Khao khát) to:
  - maintain as many processes in main memory as possible
  - free programmers from size restrictions (sự giới hạn) in program development
- With virtual memory:
  - all address references are logical references that are **translated at run time** to real addresses
  - a process can be broken up into pieces
  - two approaches are paging and segmentation
  - management scheme requires both hardware and software support

113