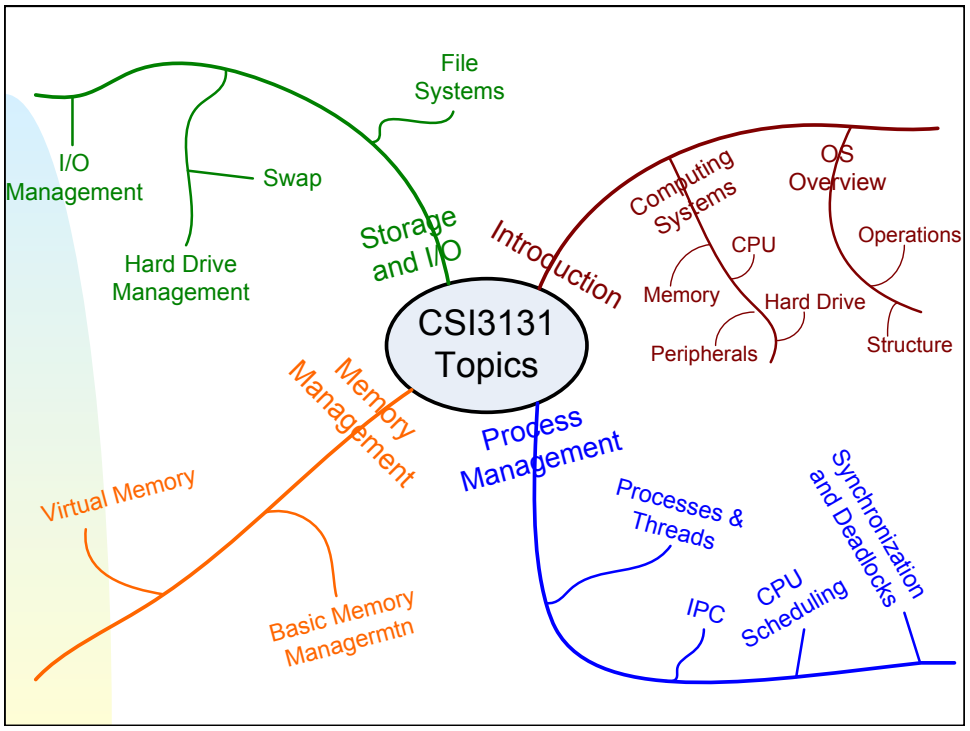
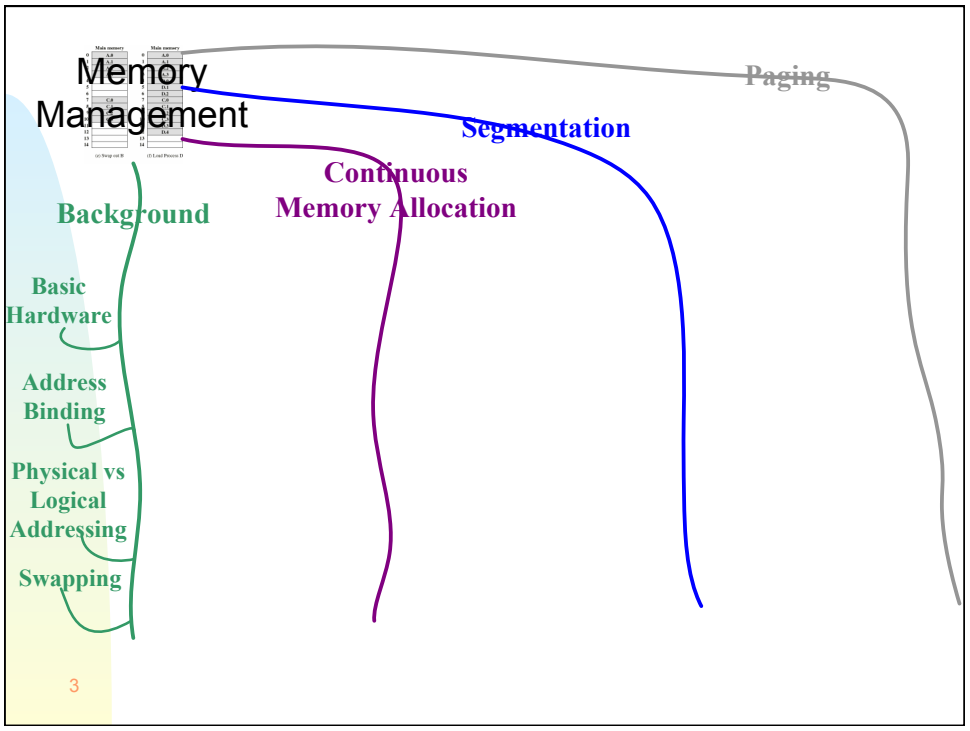


Module 7: Memory Management

Reading: Chapter 8

- **To provide a detailed description of various ways of organizing memory hardware.**
- **To discuss various memory-management techniques, including paging and segmentation.**

2



Memory Management

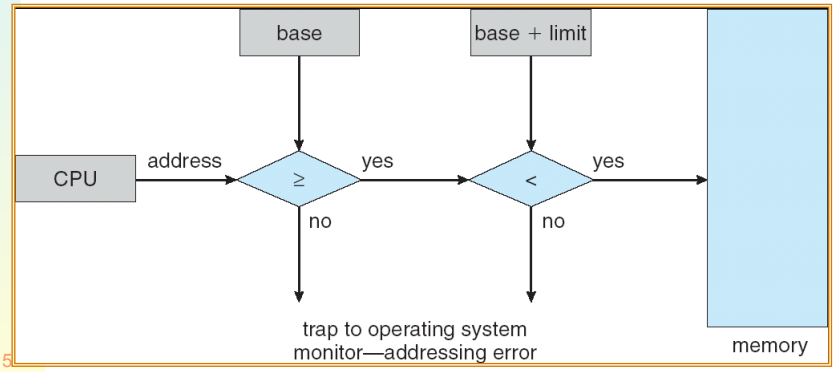
Challenge:

- **To accommodate multiple processes in main memory…**
  - **Allocate and deallocate the memory for them**
  - **Make sure they do not corrupt each other's memory**
  - **Make sure that all of this is done efficiently and transparently**
  - **Not concerned with how addresses are generated.**
- **Main requirements**
  - Relocation.
  - Sharing.
  - Allocation.
  - Protection

4

Basic Hardware

- CPU accesses main memory for execution of instructions
- For protection base and limit registers are used to trap illegal accesses to memory.
- Only OS allowed to change base and limit registers.
- All is done in hardware for efficiency.

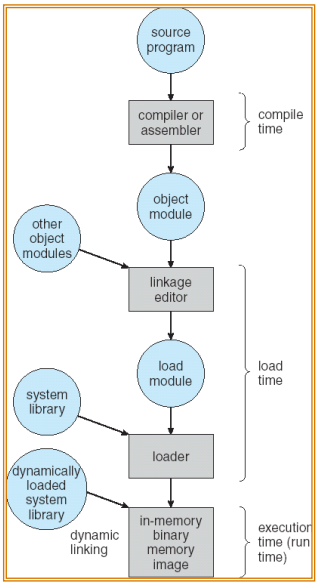


5

Address Binding

When are addresses of instructions and data bound to memory addresses?

- Compile time: **Compiler generates *absolute code***
  - **Can run only from fixed memory location, must recompile the code if the memory location changes**
- Load time: **Compiler generates *relocatable code,* loader sets the absolute memory references at load time**
  - **Once loaded, it cannot be moved**
- Execution time: **Binding is done in execution time, needs hardware support (MMU, base and limit registers)**
  - **Can be relocated in run-time**
  - **The most flexible, and the most commonly used**
  - **We will discuss how to efficiently implement it, and what kind of hardware support is needed**



6

Logical versus Physical Address Space

- **The crucial idea is to distinguish between a** logical address **and a** physical address
- **A** physical address **(absolute address) specifies a physical location in the main memory**
  - **These addresses occur on the address bus**
- **A** logical address (or virtual address) **specifies a location in the process (or program)**
  - **This location is independent of the physical structure/ organization of main memory**
  - **Compilers and assemblers produce code in which all memory references are logical addresses**
- **Relocation and other MM requirements can be satisfied by clever mapping of logical to physical addresses**
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes;**
- **Logical and physical addresses differ in execution-time address-binding scheme**

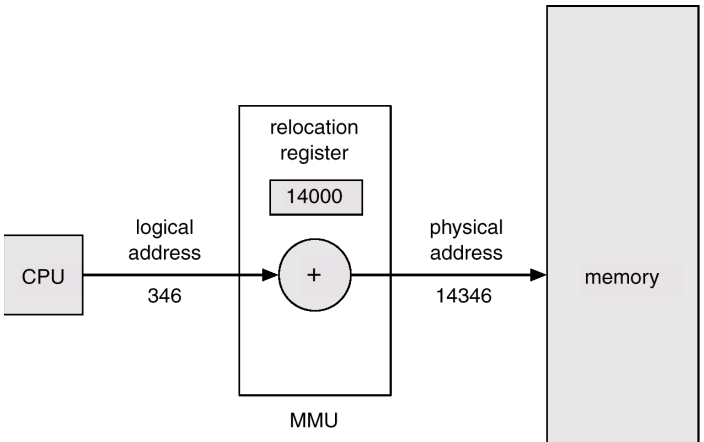
7

Simple Address Translation

- **A** relative address **is an example of logical address in which the address is expressed as a location relative to some known point in the program (ex: the beginning)**
- **Program modules are loaded into the main memory with all memory references in relative form**
- **Physical addresses are calculated "on the fly" as the instructions are executed**
- **For adequate performance, the translation from relative to physical address must by done by hardware**

8

Translating logical addresses → physical addresses



MMU: memory management unit

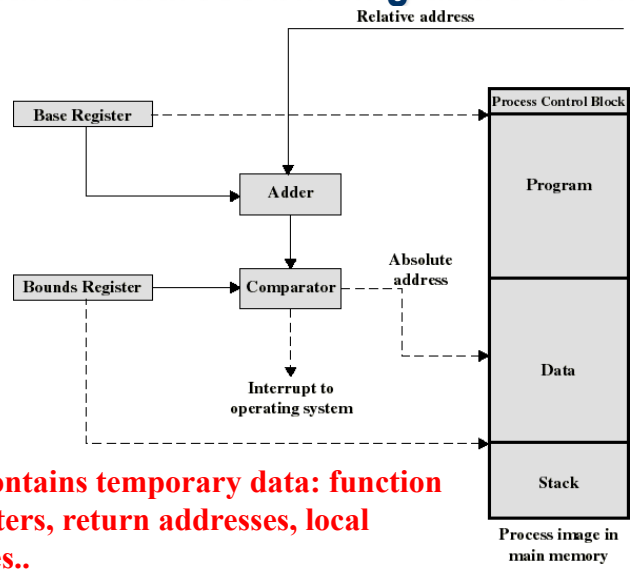
9

Simple example of hardware translation of addresses

- When a process is assigned to the running state, a base (also called *relocation*) register is loaded with the starting physical address of the process
- The limit (also called *bounds*) register is loaded with the process's ending physical address (can also be the size of the process).
- A relative address is compared with the limit register, and if it is OK, it is added to the base register to obtain the physical address
- This provides hardware protection: each process can only access memory within its process image
  - Modifying base and limit registers is a privileged instruction
- If the program is relocated, the base register is set to the new value and all references work as before

10

Dynamic address binding and checking



Stack contains temporary data: function parameters, return addresses, local variables..

11

Data: global variables

Dynamic Loading

- By default, a routine is not loaded
- It is loaded only when it is called for the first time
  - Better memory-space utilization; unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required, implemented through program design

12

Dynamic Linking

- Linking postponed until execution time
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
  - If the library is updated/fixed, the new version is automatically loaded without the need to touch the executable file
  - Careful with library versions, forward/backward compatibility…

13

Swapping

Run out of memory?

- Move process out of main memory

Where?

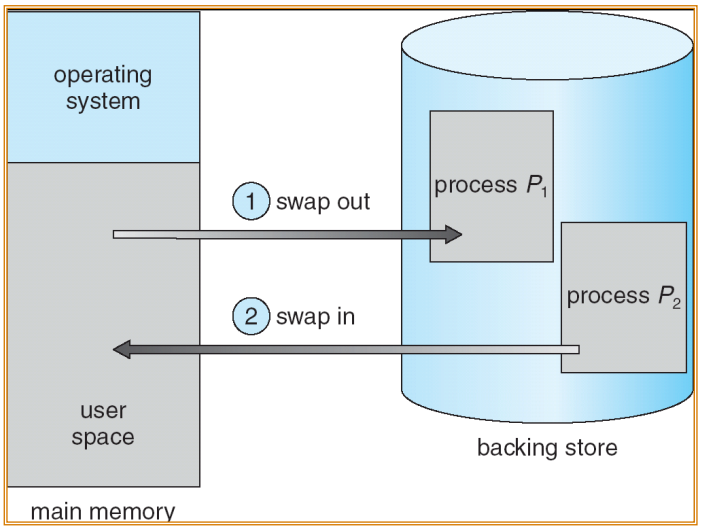
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

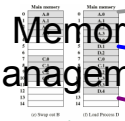
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Needs intelligent decisions, if serious swapping starts, you can go for a coffee

14

Schematic View of Swapping



15



Memory Management

Paging

Segmentation

Continuous Memory Allocation

Background

Basic Hardware

Address Binding

Physical vs Logical Addressing

Swapping

Mapping and Protection

Memory Allocation

Fragmentation

16

Contiguous Allocation

What do we mean by contiguous allocation?

- **Each process occupies a contiguous block of physical memory**

Where is the OS kernel?

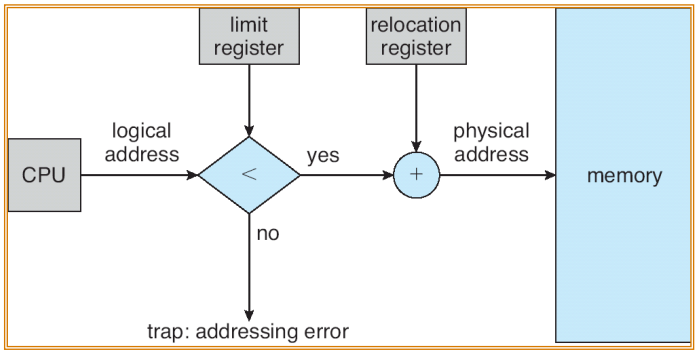
- **usually held in low memory, together with the interrupt vector**

With contiguous allocation, a base **and a** limit **register are sufficient to describe the address space of a process**

Fixed Partitioning **or** Dynamic Partitioning **can be used**

17

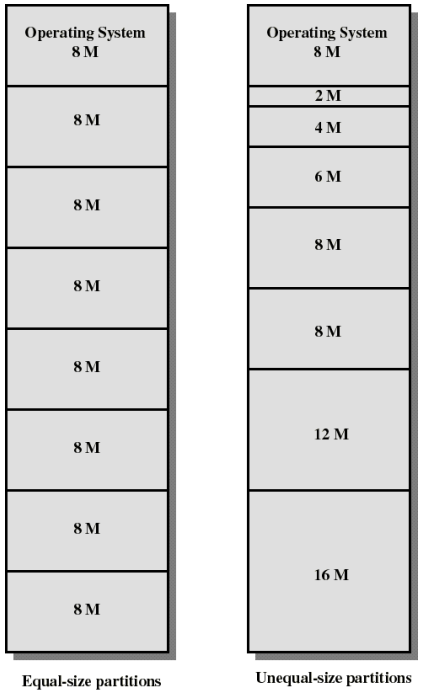
Hardware Support for Base/Relocation and Limit Registers



18

Memory Allocation: Fixed Partitioning

- Partition main memory into a set of non overlapping regions called partitions
- Partitions can be of equal or unequal sizes
- any process whose size is less than or equal to a partition size can be loaded into the partition
- if all partitions are occupied, the operating system can swap a process out of a partition



Equal-size partitions

Unequal-size partitions

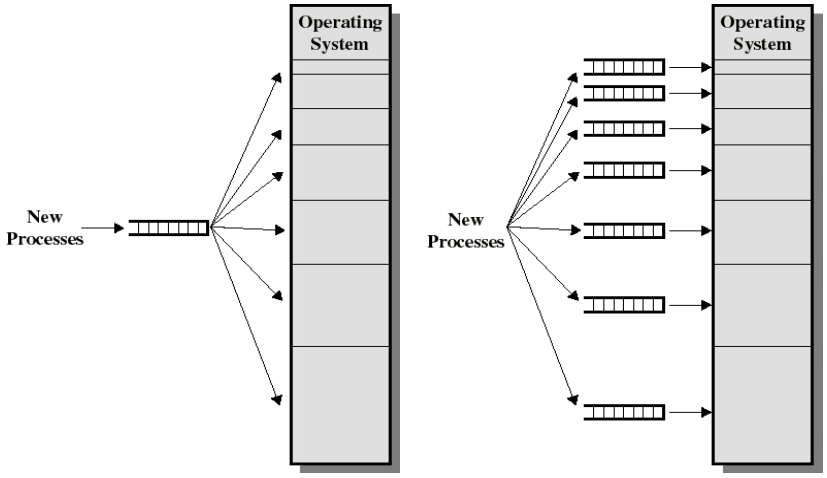
19

Placement Algorithm with Partitions

- Equal-size partitions
  - If there is an available partition, a process can be loaded into that partition
    - because all partitions are of equal size, it does not matter which partition is used
  - If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process

20

Placement Algorithm with Partitions – unequal sizes



21

Fixed Partitioning – Problems

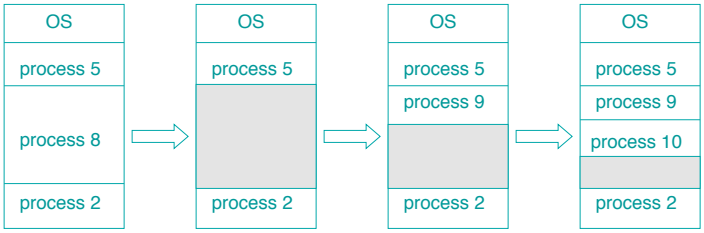
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.
- Unequal-size partitions lessens these problems but they still remain...
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)

22

Memory Allocation: Dynamic Allocation

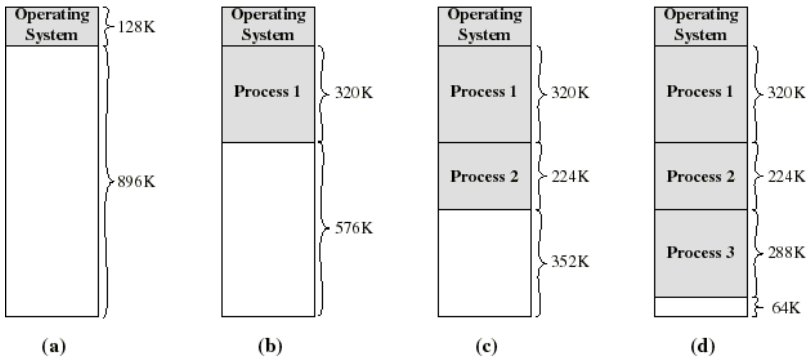
Idea: Forget partitions, put a new process into a large enough hole of unused memory.

- **The physical memory consists of allocated partitions and *holes* – blocks of available memory between already allocated.**
- **Initially, there is one huge hole containing all free memory**
- **Operating system maintains information about: a) allocated partitions b) free partitions (holes)**



23

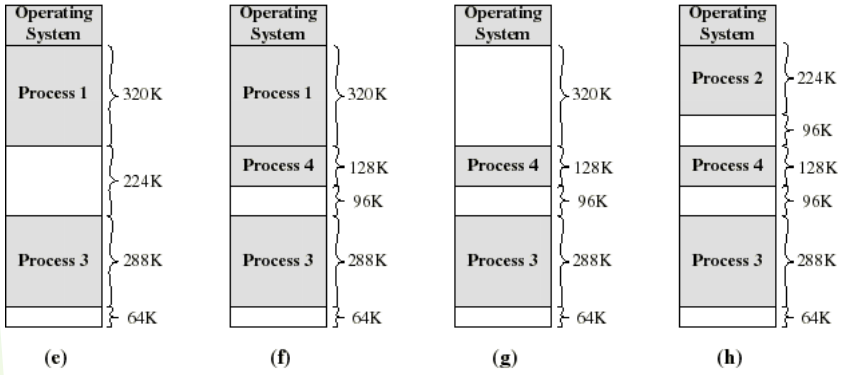
Dynamic Allocation: an example



- **A hole of 64K is left after loading 3 processes: not enough room for another process**
- **Eventually each process is blocked. The OS swaps out process 2 to bring in process 4**

24

Dynamic Allocation: an example



- another hole of 96K is created
- Eventually each process is blocked. The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...

25

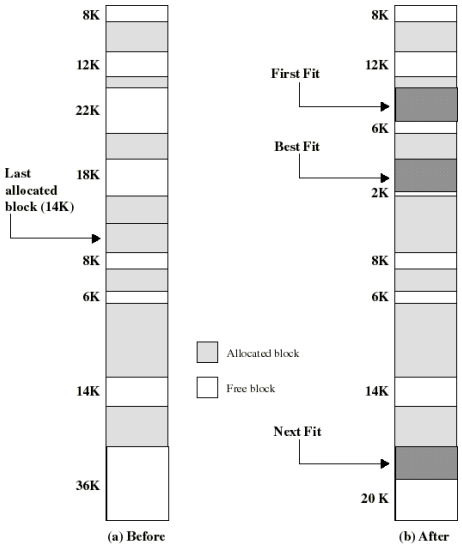
Placement Algorithm in Dynamic Allocation

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: **Allocate the first hole that is big enough, starting from the last allocated partition**
- 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 and next-fit in terms of speed and storage utilization

26



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

27

Fragmentation

What happens after some time of allocation and deallocation?

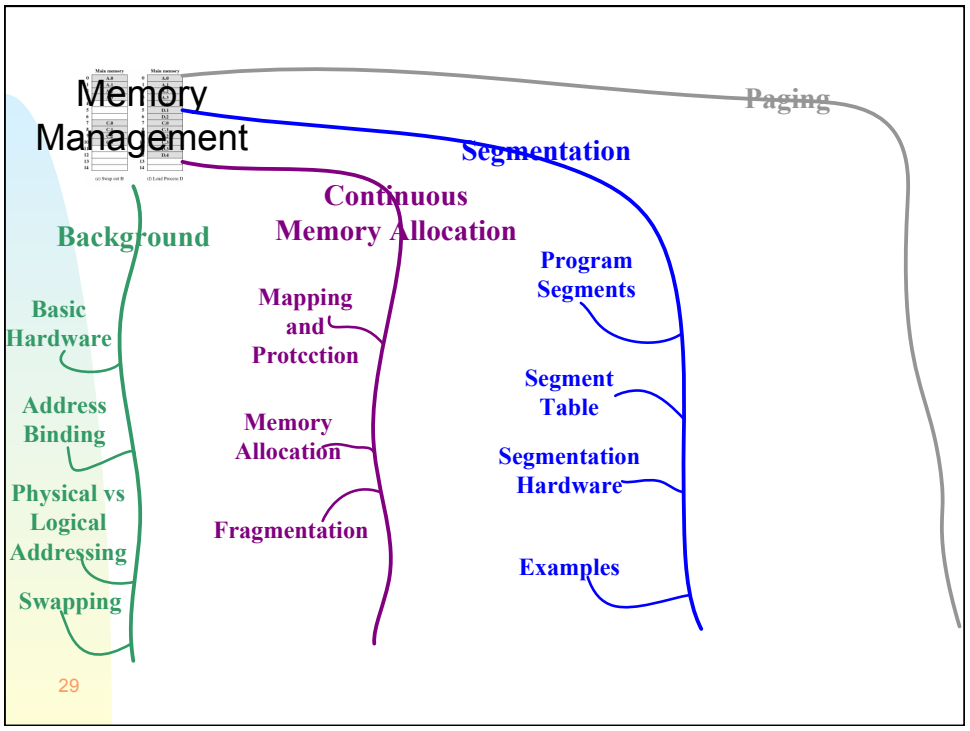
- **Plenty of small holes nobody can fit into**

- External Fragmentation **– total memory space exists to satisfy a request, but it is not contiguous**
- **Reduce external fragmentation by** compaction
  - **Shuffle memory contents to place all free memory together in one large block**
  - **Compaction is possible *only* if relocation is dynamic, and address binding is done at execution time**

What happens if we allocate more then requested?

- **e.g. we allocate only in 1k blocks, or add the remaining small hole to the process's partition**
- Internal Fragmentation

28



Non-contiguous Allocation

OK, contiguous allocation seems natural, but we have the fragmentation problem.

Also: **how do we grow/shrink the address space of a process?**

Idea: **If we can split the process into several non-contiguous chunks, we would not have these problems**

Two principal approaches:

- **Segmentation**
- **Paging**
- **Can be combined**

30

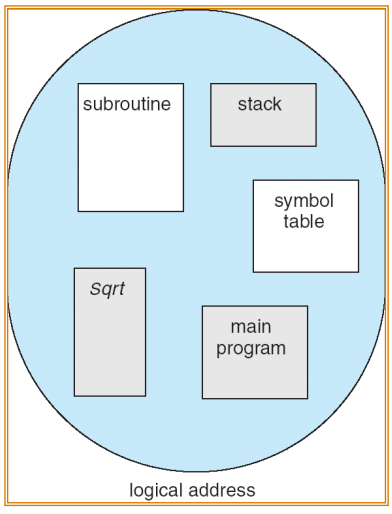
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  main program,
  procedure,
  function,
  method,
  object,
  global variables,
  common block,
  stack,
  symbol table, arrays

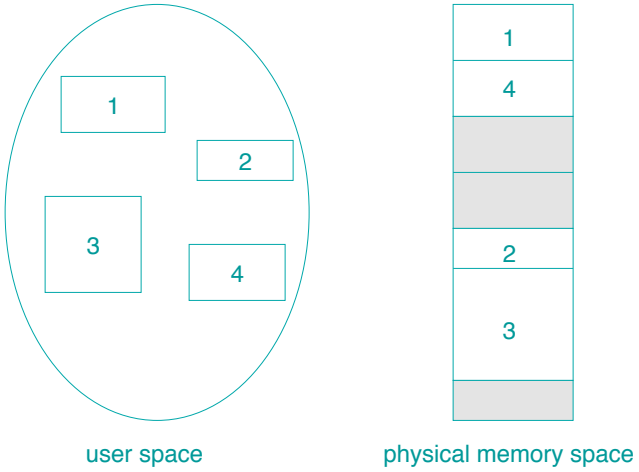
31

User's View of a Program



32

Logical View of Segmentation



33

Segmentation Architecture

- How do we logically address a memory location?
  - Segment number + offset within segment:
    <segment-number, offset>
- What information do we need to store in order to be able to translate logical address into physical?
  - For each segment:
    - Start of the segment (*base*)
    - Length of the segment (*limit*)
- Where do we store this information?
  - Segment table
- What do we need to know about segment table?
  - Where it starts? *Segment-table base register (STBR)*
  - How long is it? *Segment-table length register (STLR)*
    - segment number *s* is legal if *s* < STLR

34

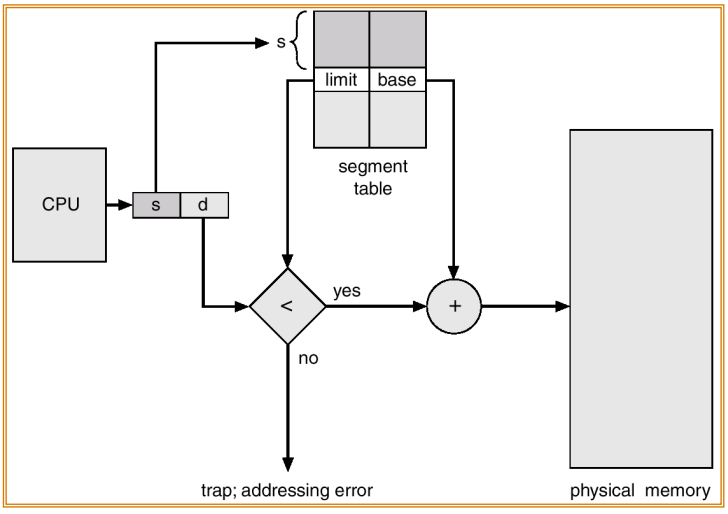
Segmentation Architecture (Cont.)

- Relocation.
  - dynamic
  - change the base pointer in the segment table entry
- Sharing.
  - shared segments
  - same segment table entries
- Allocation.
  - first fit/best fit
  - external fragmentation
- Protection
  - protection bits associated with each segment
  - code sharing occurs at segment level

35

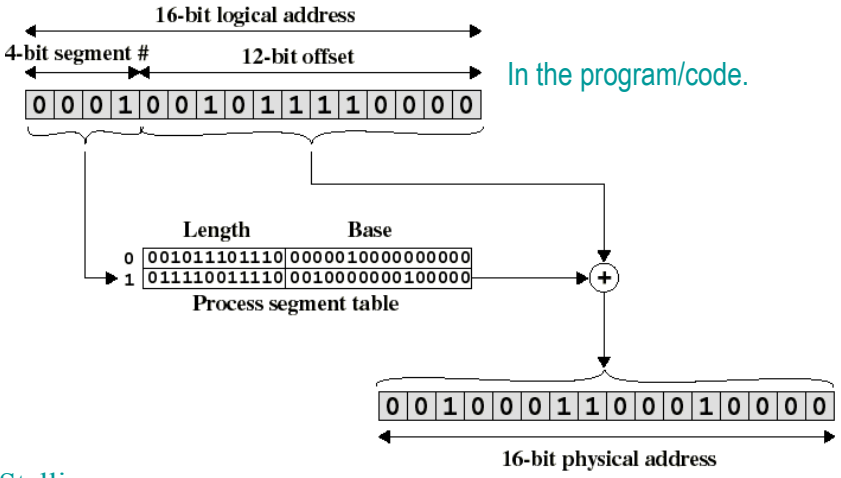
Segmentation Hardware

How to calculate physical address with segmentation?



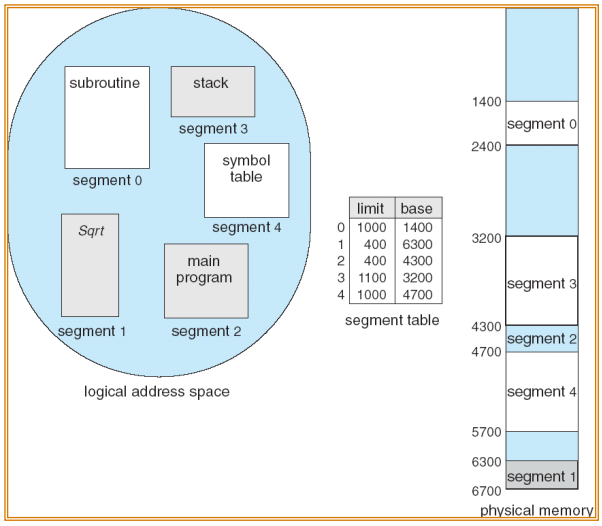
36

Details on address translation (in hardware).

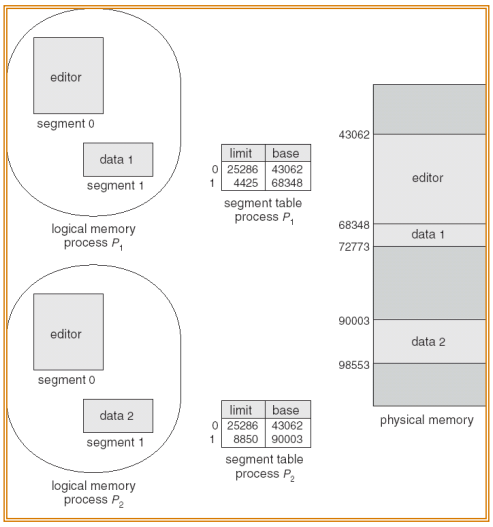


Stallings

Example of Segmentation



Sharing of Segments



39

Segmentation and protection

- Each entry in the segment table contains protection information
  - Segment length
  - User privileges on the segment: read, write, execution.
    - If at the moment of address translation, it is found that the user does not have access permissions → interruption
    - This info can vary from user to user (i.e. process to process), for the same segment.

limit	base	read, write, execute?

40

Discussion of simple segmentation

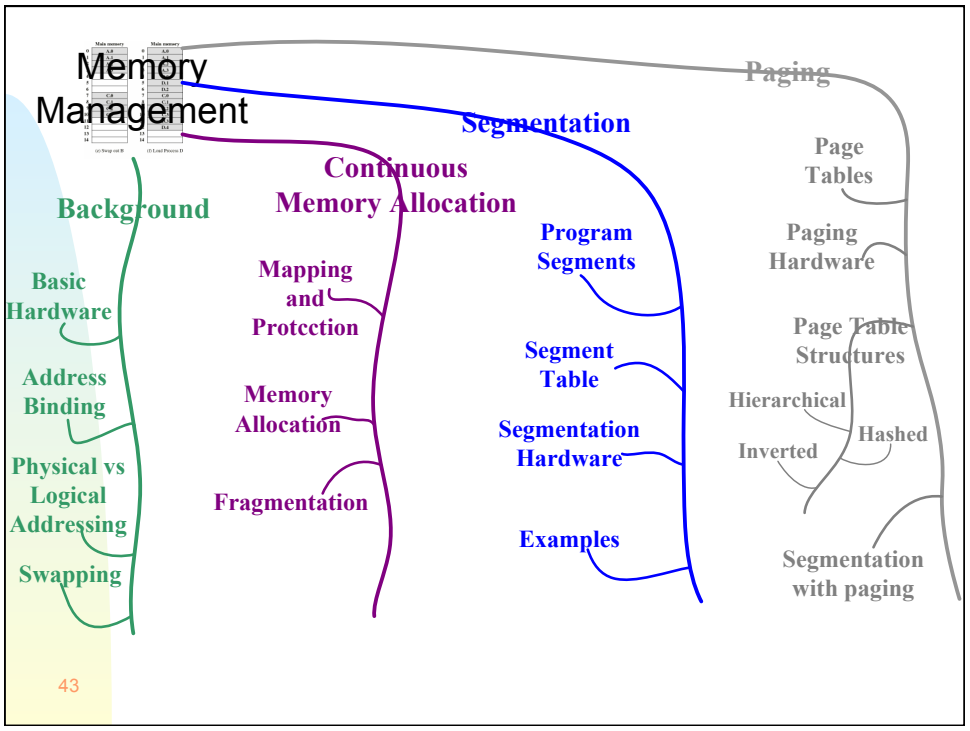
- Advantages: the unit of allocation is
  - Smaller than the entire program/process
  - A logical unit known by the programmer
  - Segments can change place in memory
  - Easy to protect and share the segments (in principle).
- Disadvantage: the problems of dynamic partitioning:
  - External fragmentation has not been eliminated:
    - Holes in memory, compression?
- Another solution is to try to simplify the mechanism by using small units of allocation of equal size:
  - PAGING

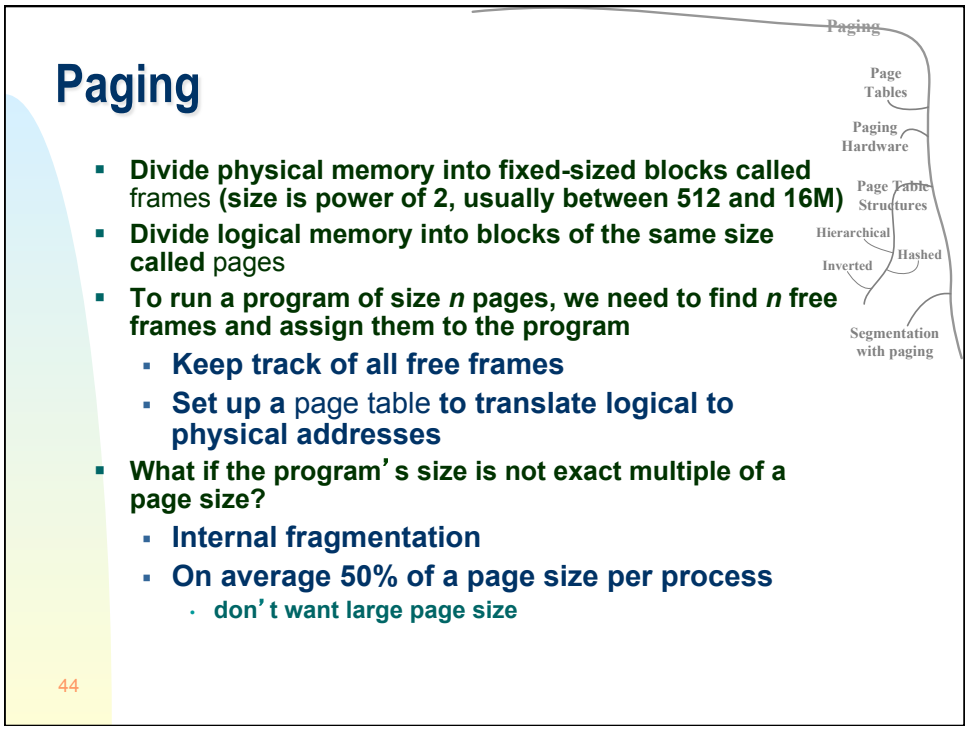
41

Segmentation versus pagination

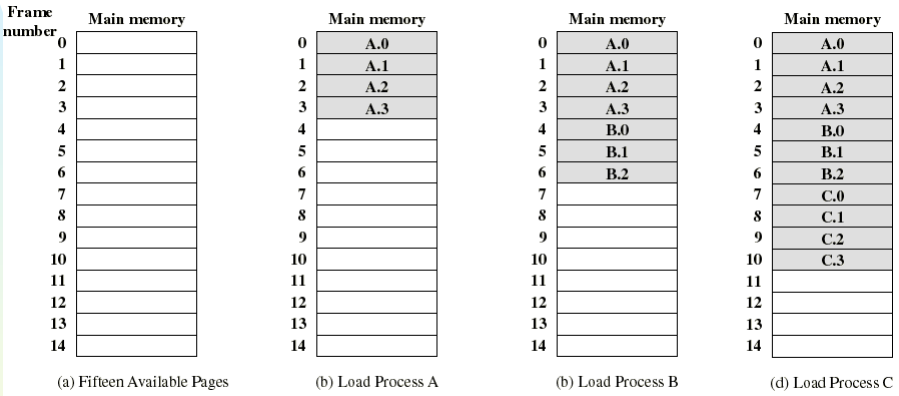
- The problem with segmentation is that the unit of allocation of memory (the segment) is of variable length.
- Paging uses units of memory allocation of fixed size, and eliminates this problem.

42


Memory Management
Background
Basic Hardware
Address Binding
Physical vs Logical Addressing
Swapping
Continuous Memory Allocation
Mapping and Protcction
Memory Allocation
Fragmentation
Segmentation
Program Segments
Segment Table
Segmentation Hardware
Examples
Paging
Page Tables
Paging Hardware
Page Table Structures
Hierarchical
Hashed
Inverted
Segmentation with paging
43


Paging
Divide physical memory into fixed-sized blocks called frames (size is power of 2, usually between 512 and 16M)
Divide logical memory into blocks of the same size called pages
To run a program of size n pages, we need to find n free frames and assign them to the program
Keep track of all free frames
Set up a page table to translate logical to physical addresses
What if the program's size is not exact multiple of a page size?
Internal fragmentation
On average 50% of a page size per process
don't want large page size
Page Tables
Paging Hardware
Page Table Structures
Hierarchical
Hashed
Inverted
Segmentation with paging
44

Example of process loading



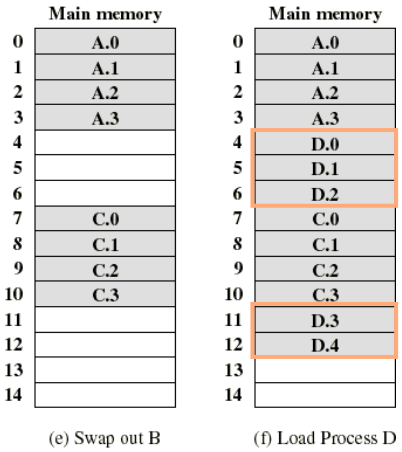
(a) Fifteen Available Pages (b) Load Process A (b) Load Process B (d) Load Process C

- Now suppose that process B is swapped out

45

Example of process loading (cont.)

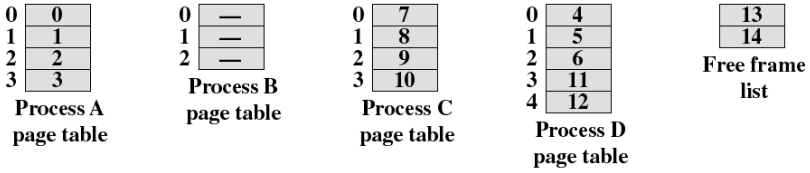
- When process A and C are blocked, the pager loads a new process D consisting of 5 pages
- Process D does not occupy a contiguous portion of memory
- There is no external fragmentation
- Internal fragmentation consist only of the last page of each process



(e) Swap out B (f) Load Process D

46

Page Tables



- The OS now needs to maintain a page table for each process; page= logical memory block
- Each entry of a page table consist of the frame (physical memory block) number where the corresponding page is physically located
- The page table is indexed by the page number to obtain the frame number
- A free frame list, available for pages, is maintained

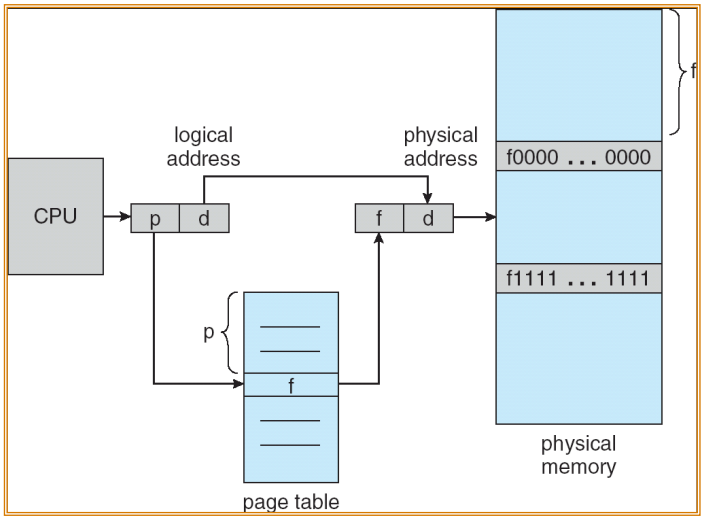
47

Address Translation Scheme

- Address (logical) generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains number of each frame in physical memory
  - *Page offset (d)* – combined with frame number, defines the physical memory address that is sent over the address bus

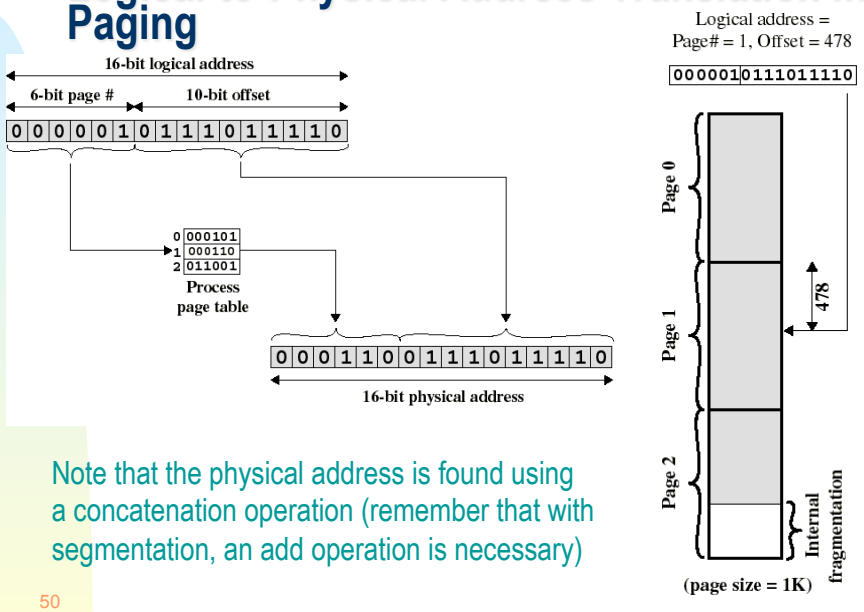
48

Address Translation in Paging



49

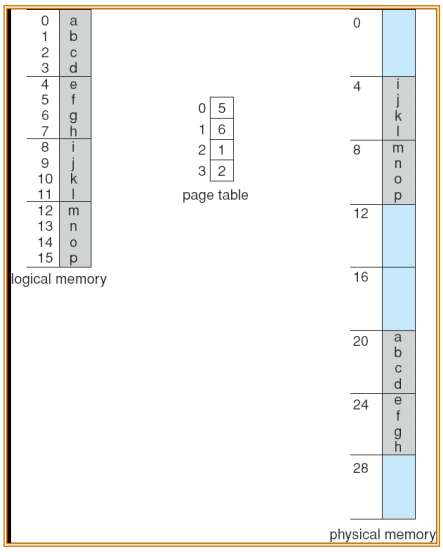
Logical-to-Physical Address Translation in Paging



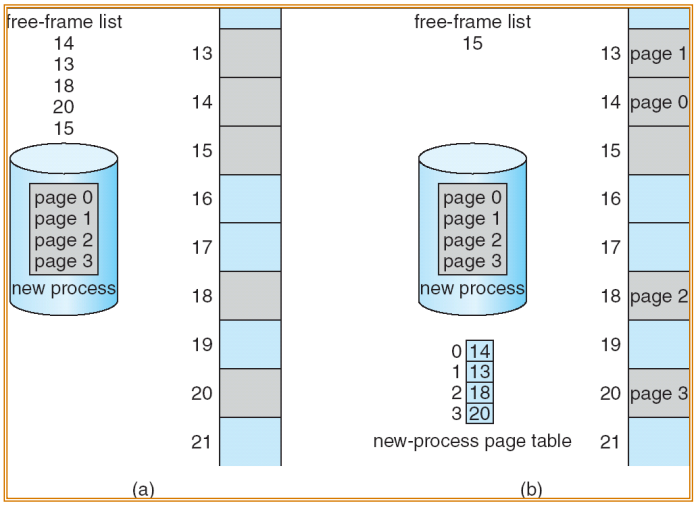
Note that the physical address is found using a concatenation operation (remember that with segmentation, an add operation is necessary)

50

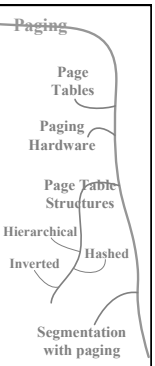
Paging Example



Free Frames



Implementation of Page Table



Where are page tables stored?

- **Ideally – special fast registers on the CPU**
  - **But page tables can be really huge**
- **In reality – in main memory**
  - ***Page-table base register* (PTBR) points to the page table**
  - ***Page-table length register* (PRLR) indicates the page table size**
- **What happens on each memory reference?**
  - **we have to read page table to figure out address translation**
  - **every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.**

53

Implementation of Page Table

Two memory accesses per each memory reference is too much!

What to do?

Idea: **Cache page table entries**

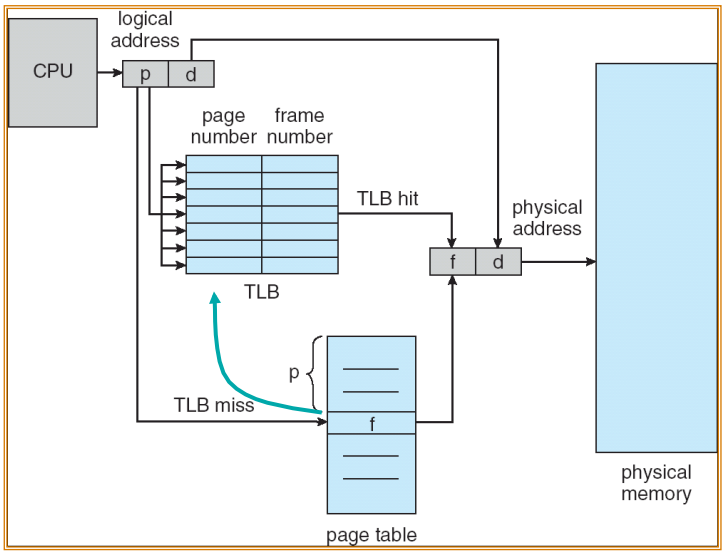
- **special fast-lookup hardware cache called** translation look-aside buffer (TLB)

How does TLB work?

- **associative memory – checks all entries in parallel whether they contain a pair (page #, frame#) for the needed page#**
- **If hit, we got the frame# and saved one memory access**
- **If miss, still need to go the memory to look-up the page table**

54

Paging Hardware With TLB



55

Effective Access Time

- Associative Lookup = ε time units
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

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

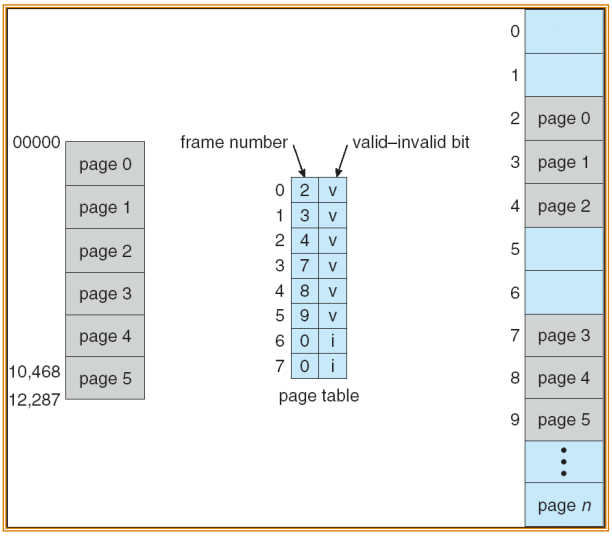
56

Memory Protection

- We can associate several bits with each page
  - read only/read-write
  - valid-invalid
  - these bits are stored for each entry of the page table
- Checking for accessing only process's memory
  - If page table size is fixed so that it stores the whole process's addressable memory
    - valid-invalid bits
  - If page table size corresponds to the memory the process is really using
    - Compare page# with *page-table length register* (PTLR)

57

Valid (v) or Invalid (i) Bit In A Page Table



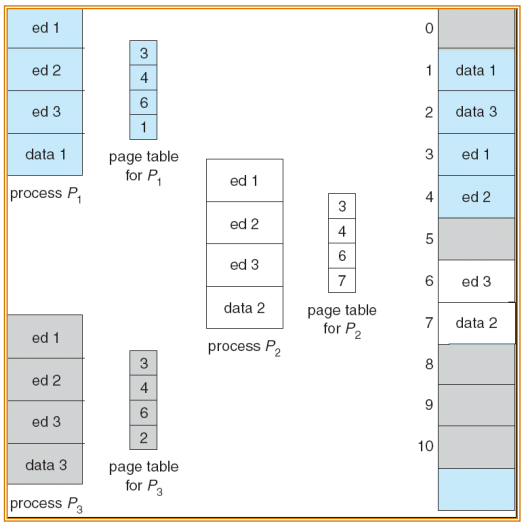
58

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Private code and data
  - Each process keeps a separate copy of private code and data

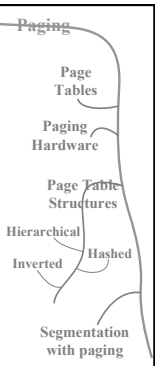
59

Shared Pages Example



60

Page Table Structure



How big can a page table be?

- **32-bit address space, 4kb page -> 2^{20} pages = 1M page entries!**
- **Better come-up with something smart!**

- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

61

Hierarchical Page Tables

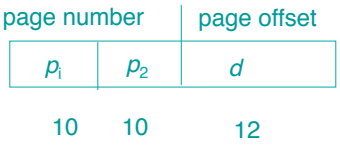
Idea: **The page table itself is paged**

- **A simple technique is a two-level page table:**
  - **The outer page table entries point to the pages of the real page table**
- **Sometimes (i.e. SPARC), three-level page tables are used**

62

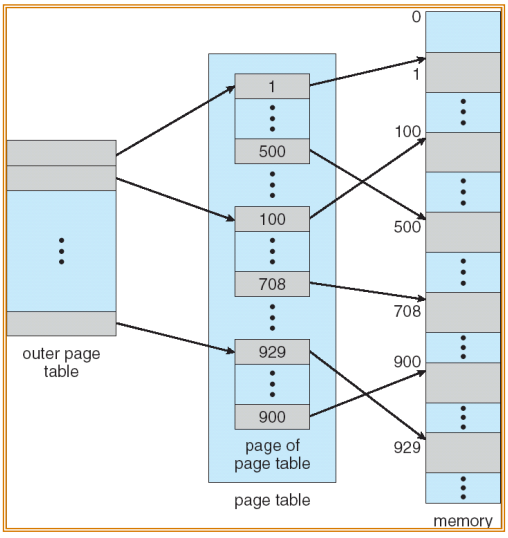
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



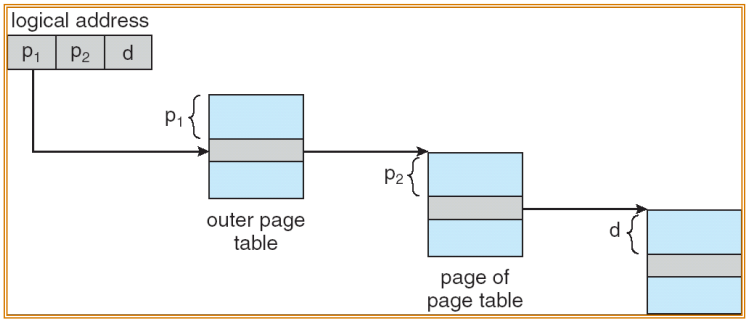
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Page-Table Scheme



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



- In the case of the two-level paging, the use of the TLB becomes even more important to reduce the multiple memory accesses to calculate the physical address.

65

Hashed Page Tables

Assume the address space is 64 bits

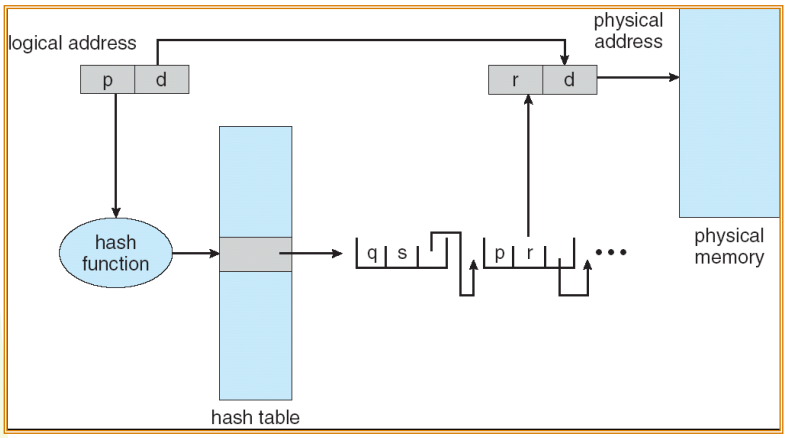
How many level of page hierarchy would we need?

- With 4k pages we have 2^{52} pages
- Assume 10 bits per level = 1K entries
- 5 levels are not enough!
- (one page fits only 1k entries)
- That is not feasible, what can we do?

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

66

Hashed Page Table



67

Inverted Page Table

We can have plenty of processes…

… and each one of them has big page table.

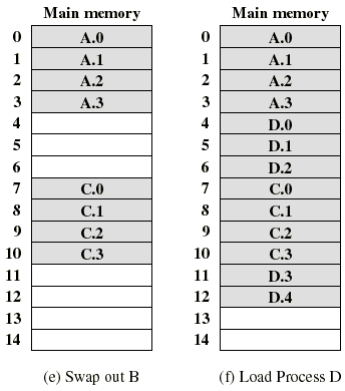
processes may not fit into memory but page table entries are not removed

wasting memory on their page tables…

and same page can be listed several times

Idea: **have only one entry for each physical frame of memory**

- **Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page**



68

Inverted Page Table

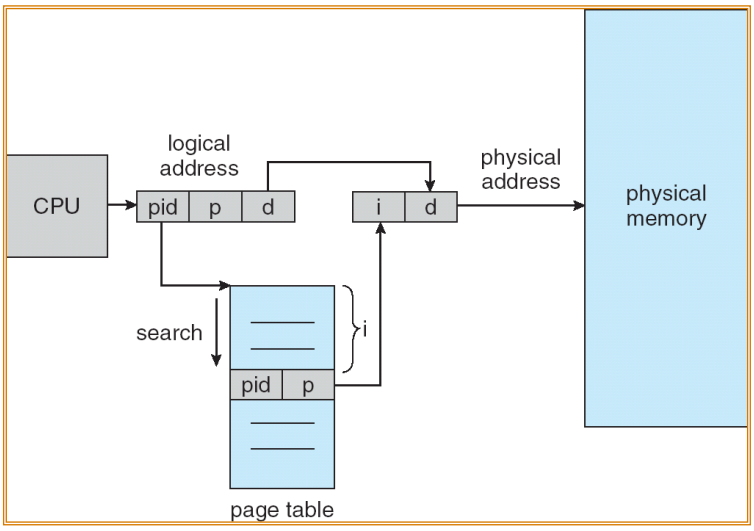
Now we need only one entry per physical frame, not per each process's logical page

But how do we translate from page # to frame #?

- We have to search the inverted page table until we find match for the page # and process ID
- That can be really slow….
- What to do?
  - Use hash table to limit the search to one — or at most a few — page-table entries
  - Shared memory is a problem

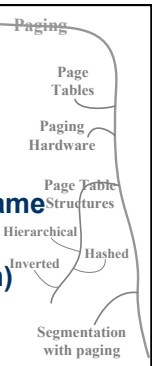
69

Inverted Page Table Architecture



70

Segmentation versus Pagination

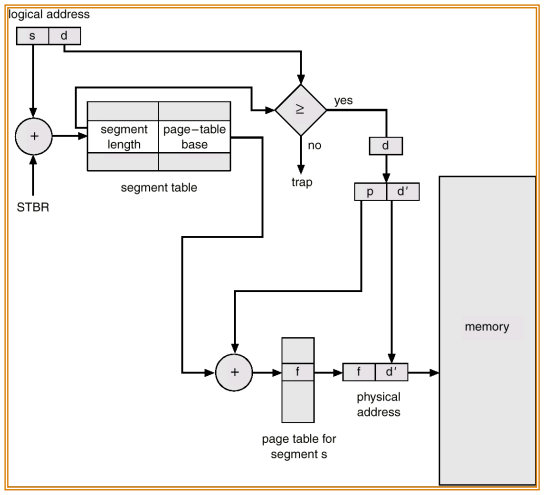


- **Paging**
  - Not visible to the user
  - Suffers from internal fragmentation (but only ½ frame per process)
  - Address translation is simple (uses concatenation)
- **Segmentation**
  - Based on the user view
  - Since segment is a logical unit for protection and sharing, these functions easier with segmentation.
  - Requires more expensive material for address translation (requires addition).
- **Fortunately segmentation and paging can be combined.**
  - Divide segments into pages.

71

MULTICS Address Translation Scheme

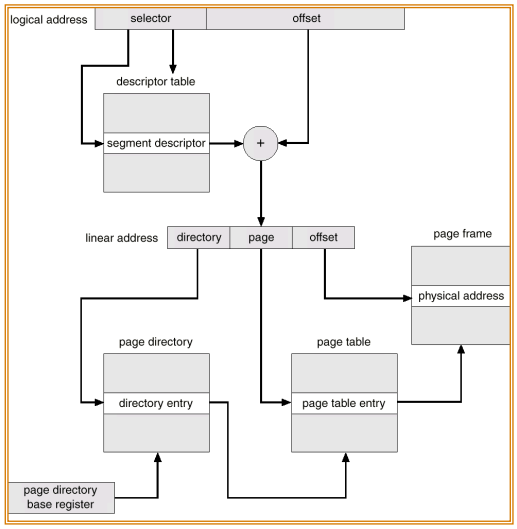
Segment page entry contains pointer to the page table of that segment



72

Intel 30386 Address Translation

Intel 30386 has segmentation with two level hierarchical paging

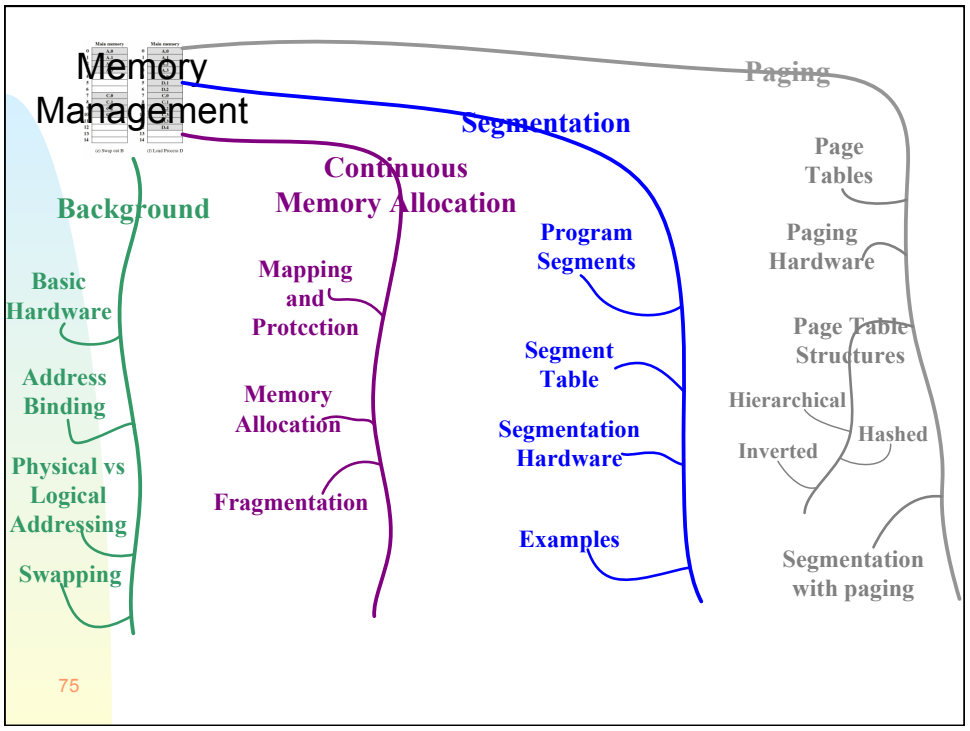


73

Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

74


Memory Management
Background
Basic Hardware
Address Binding
Physical vs Logical Addressing
Swapping
Continuous Memory Allocation
Mapping and Protcction
Memory Allocation
Fragmentation
Segmentation
Program Segments
Segment Table
Segmentation Hardware
Examples
Paging
Page Tables
Paging Hardware
Page Table Structures
Hierarchical
Hashed
Inverted
Segmentation with paging
75