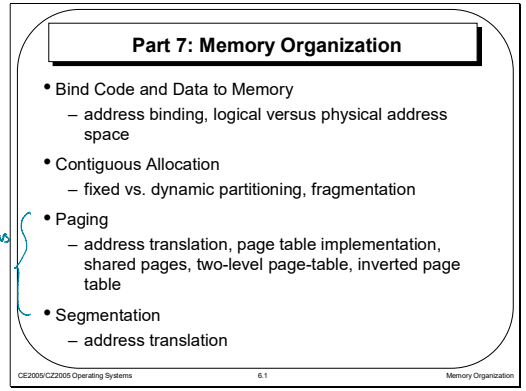
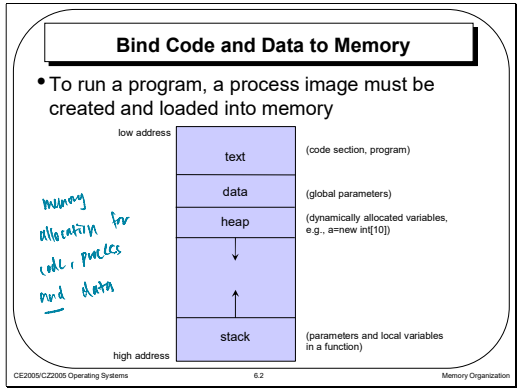
Slide 1



Effective memory management is vital in a multiprogramming system. To improve CPU utilization, multiple processes need to be loaded into memory to ensure a reasonable supply of ready processes to consume available CPU time.

We begin this chapter by understanding how a process gets executed. Next, we approach the technology of memory management by looking at a variety of schemes that have been used.

Slide 2

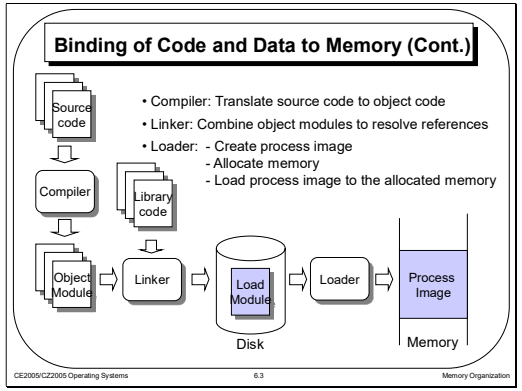


How is memory allocated for a process image?

Logical view of a process in memory (also called process image) – not just code and data, but also include dynamic memory space (stack and heap) required during process execution

It defines logically the space a process is able to access (or address) during its execution. So, this is called logical address space of a process. Address reference generated when executing an instruction always refers to this logical address space and is called logical address.

Slide 3

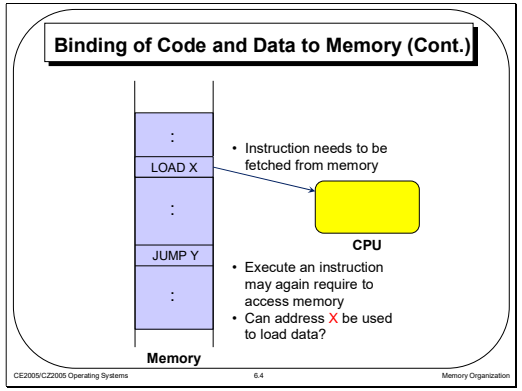


How is a process image generated and loaded?

Loader is part of OS

A program may contain multiple source code files. Source code files are compiled into object files. The linker combines these object files into a single binary executable (i.e., the load module on disk). During the linking phase, other object files or libraries may be included as well, such as the standard C or math library. A loader is used to load the binary executable file into memory where its is eligible to run on a CPU core.

Slide 4



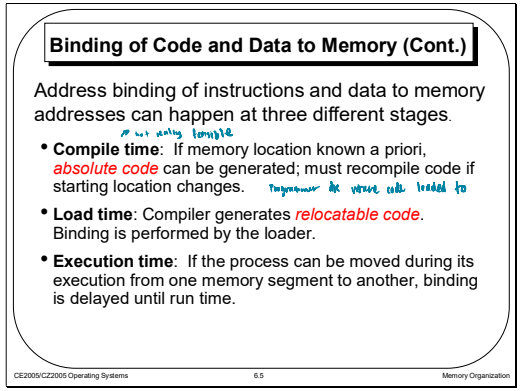
As a process executes, it accesses instructions and data in memory

Instruction execution cycle

CPU must deal with memory references ~~within the program~~ generated. Branch instructions contain an address to reference the instruction to be executed next. Data reference instructions contain the address of the byte or word of data referenced.

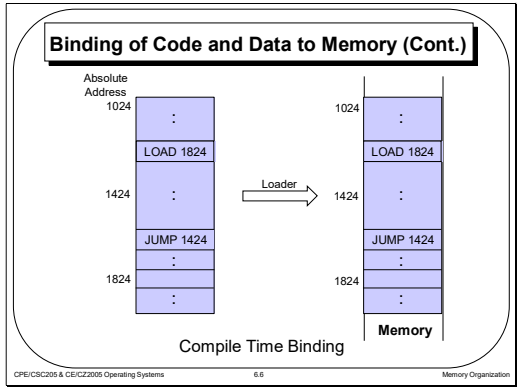
Address generated while executing an instruction may not be used directly to access physical memory, depending on when address binding is performed.

Slide 5

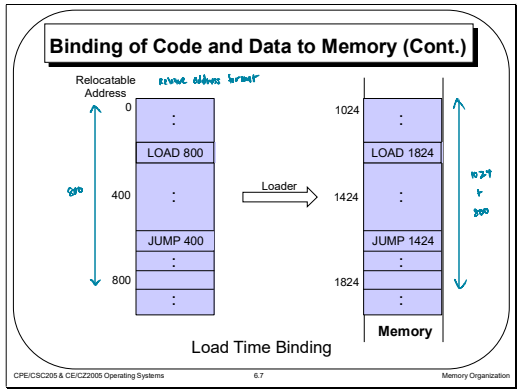


Address binding refers to the action of binding (or translating) the addresses in process image to the physical memory addresses (or absolute addresses).

Slide 6



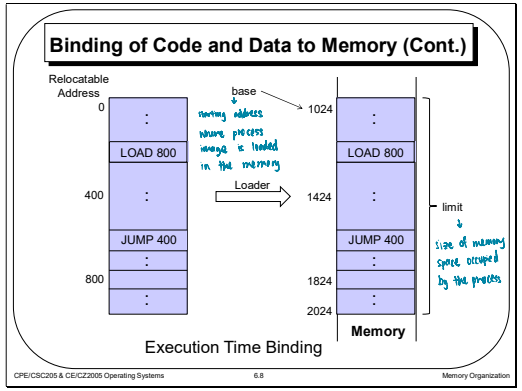
Slide 7



Compiler generates code in re-locatable address format, i.e., address format that is relative to the beginning of the address space (also called relative address).

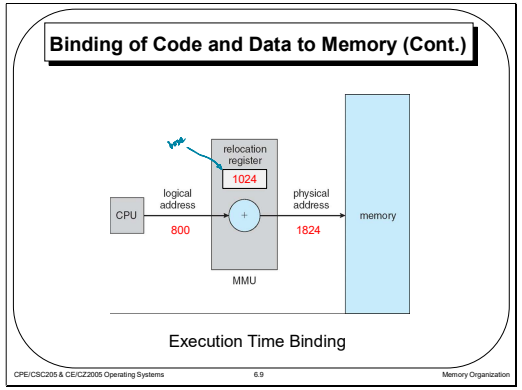
Loader will translate the re-locatable address to absolute address.

Slide 8

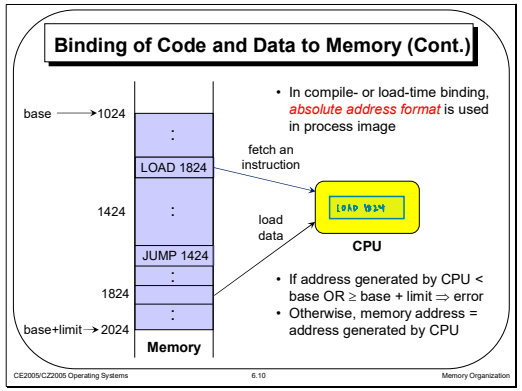


Base – starting address where process image is loaded in the memory.
Limit – size of the memory space occupied by the process.

Slide 9



Slide 10

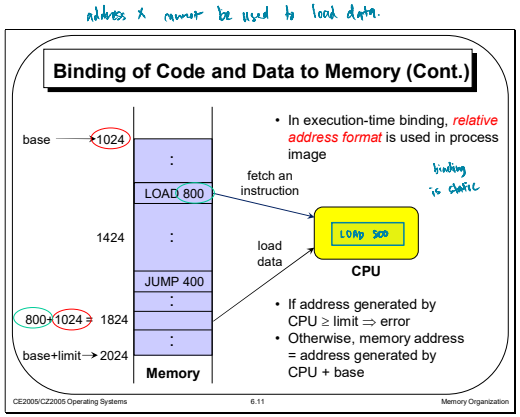


For compile-time and load-time binding, absolute address is used directly to access memory.
So, the binding of logical address space to the physical memory is static. Once process image is loaded, it can't be moved in the memory.

In a multiprogramming system, the available memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in physical memory at the time of execution of his or her program. In addition, we would like to be able to:
• Perform memory compaction
• Swap in and out process
• Provide virtual memory support

Hence, we are not interested in compile- or load-time binding.

Slide 11



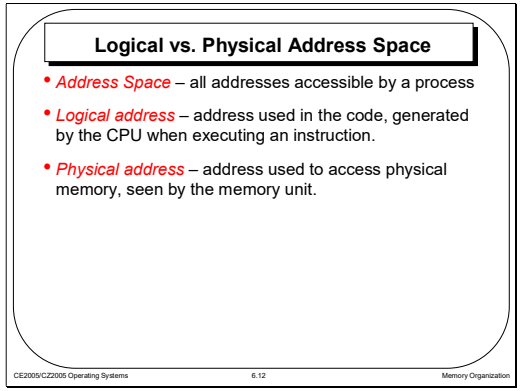
Relative address – address format that is relative to the beginning of the process image.
Base – starting address where process image is loaded in the memory.

In execution-time binding, address generated by CPU needs to be translated to the one used to access memory and this translation depends on memory allocation method used.

The address translation is done by MMU and supported by hardware – recall base and limit registers.

Execution-time binding is dynamic, process image can be moved in the memory. In the following lecture, we assume execution-time binding.

Slide 12



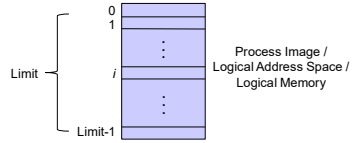
Process executes in logical address space and it never sees physical address. There is one logical address space per process.

To access physical memory, logical address may need to be translated into physical address.

Slide 13

Logical vs. Physical Address Space

- Logical address space can be viewed as a linear, or one-dimensional, continuous address space consisting of a sequence of bytes
- For execution time binding, logical address space always starts from 0.



Slide 14

How to Allocate Memory among Processes

- Two approaches:
  - *Contiguous Allocation*

    The logical address space of a process remains contiguous in physical memory
    - Fixed Partitioning
    - Dynamic Partitioning

  - *Non-contiguous Allocation*

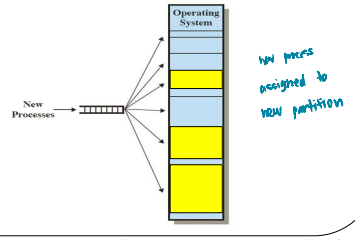
    A process logical address space is scattered over different regions in physical memory

Slide 15

Contiguous Allocation

- *Fixed Partitioning*

  Memory is partitioned into regions with fixed boundaries



• The number of partitions specified at system generation time limits the number of active processes in the system.

• Because partition sizes are preset at system generation time, small processes will not utilize partition space efficiently. Any process, no matter how small, occupies an entire partition. This approach is not really practical. It is difficulty to determine the number of partitions and size of each partition since the memory requirements of all processes are in general unknown beforehand.

Slide 16

Contiguous Allocation (Cont.)

- *Dynamic Partitioning*
  - *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
    - allocated partitions
    - free partitions (hole)

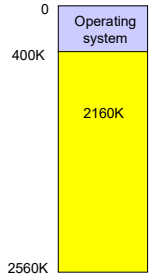
Initially, there is a single hole.

Memory allocation status may change while processes are created (allocate memory) and terminate (de-allocate memory).

Eventually, as time goes, memory contains a list of holes of vary size.

Slide 17

Contiguous Allocation (Cont.)

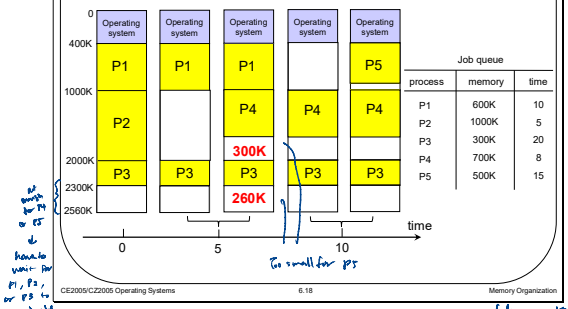


Job queue

process	memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

Slide 18

Contiguous Allocation (Cont.)



With dynamic partitioning, the partitions are of variable length and number. When a process is brought into physical memory, it is allocated exactly as much memory as it requires and no more.

Slide 19

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.
- **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.

Given memory requirement of a process, dynamic storage allocation policies determine which one to use amongst a list of free holes.

Slide 20

Fragmentation

- *External fragmentation*: when enough total memory space exists to satisfy a request, but it is not contiguous. Happens outside a partition.
- *Internal fragmentation*: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Variable partitioning -> external fragmentation
Fixed partitioning -> internal fragmentation

Slide 21

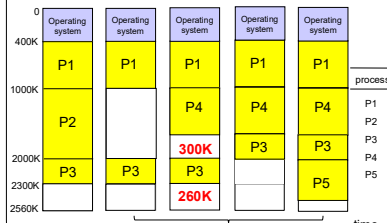
Fragmentation (Cont.)

- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place free memory together in one large block.
  - Compaction is possible only if re-locatable address format is used in process image and binding is done during execution-time.

From time to time, the OS shifts the processes so that all of the free memory is together in one block. The difficulty with compaction is that it is a time-consuming procedure and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a process from one region to another in memory without invalidating the memory references in the program.

Slide 22

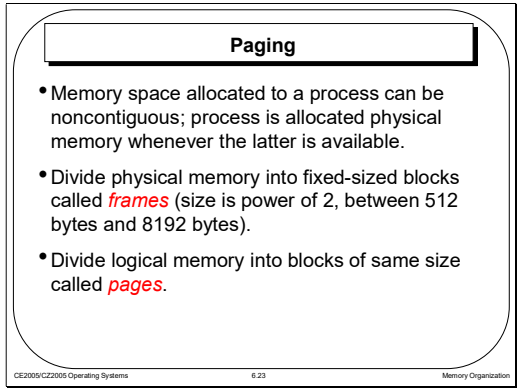
Fragmentation (Cont.)



Job queue

process	memory	time
P1	600K	10
P2	1000K	5
P3	300K	20
P4	700K	8
P5	500K	15

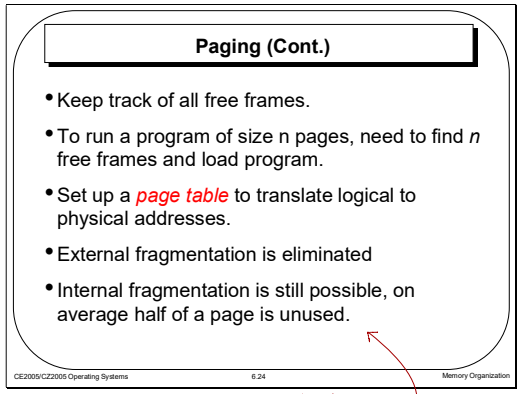
Slide 23



Memory space allocated to a process is not contiguous.

This is how paging works.

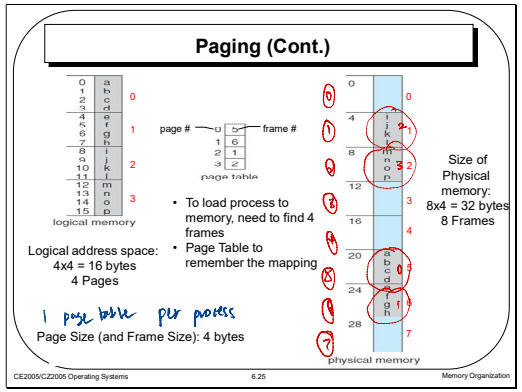
Slide 24



→ Internal fragmentation still possible, last page may not occupy the entire frame.
↳ logical address space cannot be divided into the exact number of pages

The operating system maintains a **page table** for each process. The page table shows the frame location for each page of the process.

Slide 25



Example: LOAD 0111 (7)
load "h" from logical address 0111

"h" is stored in physical location 11011 (27)

Translate "0111" to "11011"

To do address translation, need to know the logical address is in which page and the offset within the page

Slide 26

Address Translation Scheme

- Logical address contains:
  - *Page number (p)* – used as an index into a *page table* entry which contains *frame number* in physical memory.
  - *Page offset (d)* – combined with frame number to define the physical memory address that is sent to the memory unit.

1 page 1 process

1 TLB for all processes

Slide 27

Page size: 1024 bytes → 2^{10} bytes

Size of page table entry: 4 bytes

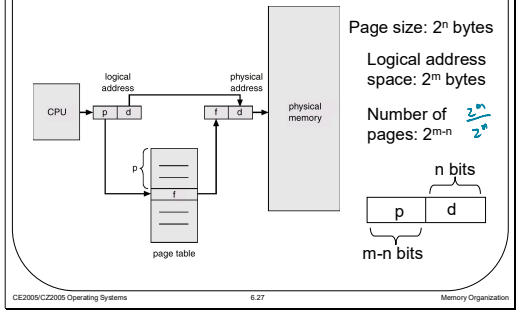
Logical address space: 2^{32} bytes

$\frac{\text{Logical address space}}{\text{Page size}}$ = no. of pages

no. of pages × page table entry size

$= \frac{2^{32}}{2^{10}} \times 4 = 2^{20} \times 2^2 = 2^{22} = 4$ MB

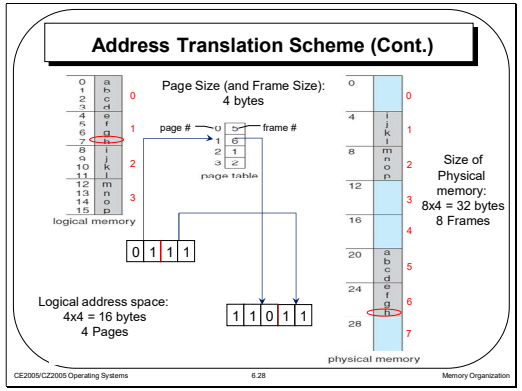
Address Translation Scheme (Cont.)



Logical address: m bits. There are 2^m bytes in logical address space.
Page size: 2^n. There are 2^m/2^n pages in logical address space.
(For the convenience of addressing, page size is always power of 2.)
Logical address is divided into two parts: lower n bits for page number and higher (m-n) bits for page number.

This figure suggests a hardware implementation. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address. Typically, the page number field is longer than the frame number field.

Slide 28



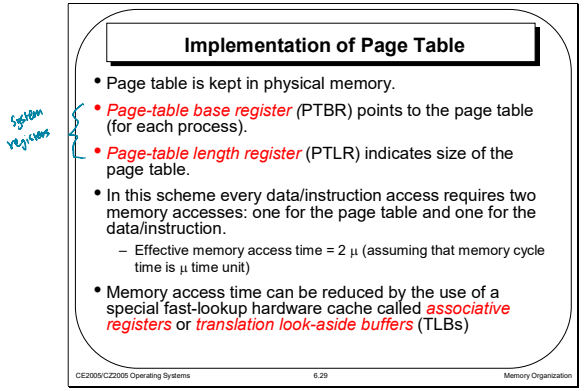
Example: LOAD 0111 (7)
load "h" from logical address 0111

"h" is stored in physical location 11011 (27)

Translate "0111" to "11011"

To do address translation, need to know the logical address is in which page and the offset within the page

Slide 29

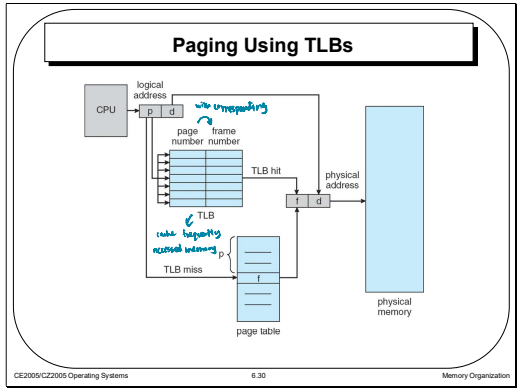


PAGE TABLE STRUCTURE The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers. Instead, it must be in physical memory to be accessed.

With paging, the logical-to-physical address translation is still done with hardware support. Now the CPU must know how to access the page table of the current process -> page-table base register and page-table length register.

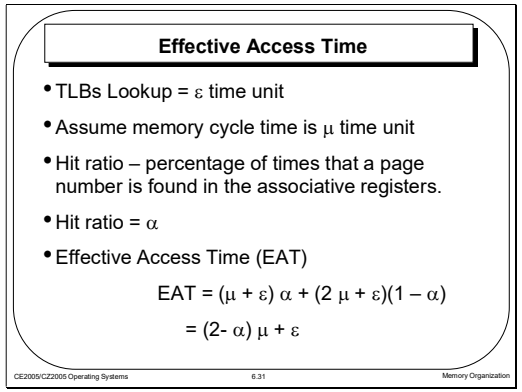
In principle, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special high-speed cache for page table entries, usually called a translation lookaside buffer (TLB).

Slide 30

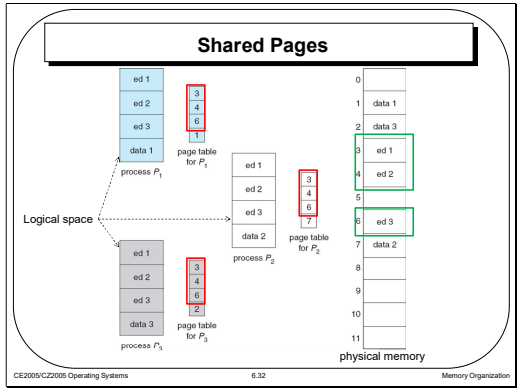


Essentially, TLB are associate registers which support parallel search. Typically, there are 64 to 1024 entries in TLB.

Slide 31



Slide 32



Sharing pages will reduce the memory usage.

In this example, three processes execute the same editor but on different data. Each process has its own logical address space, but they contain the same editor code (i.e., pages ed1, ed2, and ed3). So, instead of loading three copies of the editor code, only one copy needs to be loaded in the memory and this copy can be shared amongst 3 processes (by mapping ed1, ed2, and ed3 in each process's logical address space to the corresponding editor pages loaded in the memory as shown in the slide). This is only possible if code is read-only code (re-entrant code).

Shared code

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

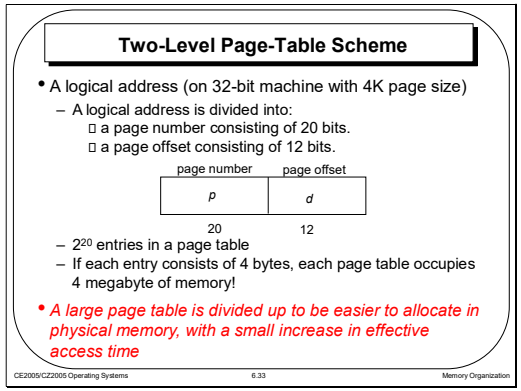
Shared code must appear in same location in the logical address space of all processes

Private code and data

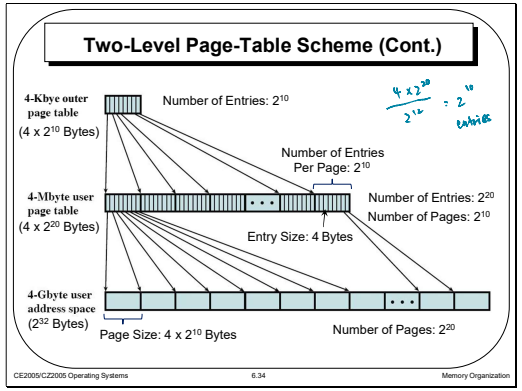
Each process keeps a separate copy of the private code and data

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

Slide 33



Slide 34



This figure shows an example of a two-level scheme typical for use with a 32-bit address.

If we assume byte-level addressing and 4-kbyte (2^12) pages, then the 4-Gbyte (2^32) virtual address space is composed of 2^20 pages.

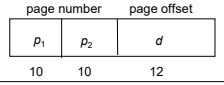
If each of these pages is mapped by a 4-byte page table entry, we can create a user page table composed of 2^20 entries, requiring 4 Mbytes (2^22).

This huge user page table, occupying 2^10 pages, can be kept in virtual memory and mapped by an outer page table with 2^10 entries, occupying 4 Kbyte (2^12) of physical memory.

Slide 35

Two-Level Page-Table Scheme (Cont.)

- Since the page table is paged, the page number is further divided into:
  - p_1: an index into the outer page (second level) table
    - The outer page table contains $(4 \times 2^{20}) / 4K = 2^{10}$ number of entries -> 10 bits for p_1
  - p_2: is an index into a page of the inner (first level) page table
    - Each page in the page table contains$(4 \times 2^{10}) / 4 = 2^{10}$ number of entries -> 10 bits for p_2
- Thus, a logical address is as follows:

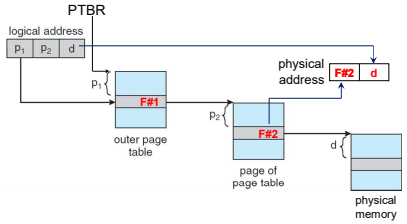


Slide 36

learn at slides animation file

Two-Level Page-Table Scheme (Cont.)

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



This figure shows the steps involved in address translation for this scheme. The outer page table always remains in physical memory. The first 10 bits of a virtual address are used to index into the outer page table to find an entry for a page of the page table. If that page is not in physical memory, a page fault occurs. If that page is in physical memory, then the next 10 bits of the virtual address index into the page table to find the entry for the page that is referenced by the virtual address.

Multiple-level paging is used in many systems, e.g., Intel IA-32 (Pentium) and IA-64 (Itanium).

Slide 37

Inverted Page Table

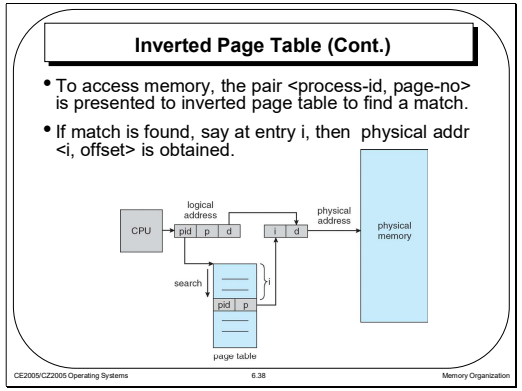
- Usually each process has its own page table. Hence the system could have many page tables, consuming substantial memory space
  - The page table size is proportional to that of the logical address space.
- **Alternative**: have a single table with one entry for each physical frame, as <process-id, page-no>. This is called an *Inverted Page Table*
- Logical address: <process-id, page-no, offset>
- Increases search time: table sorted by physical address but lookups occur on logical address

A drawback of the type of page tables that we have been discussing is that their size is proportional to that of the logical address space and there is one page-table for each process.

An alternative approach is the use of an inverted page table structure.

Entries in the inverted page table are ordered by the frame number order. Each entry contains <pid, page number> tuple. pid is needed because the inverted page table is shared by all the processes – the OS needs to know the page mapped to the frame belongs to which process.

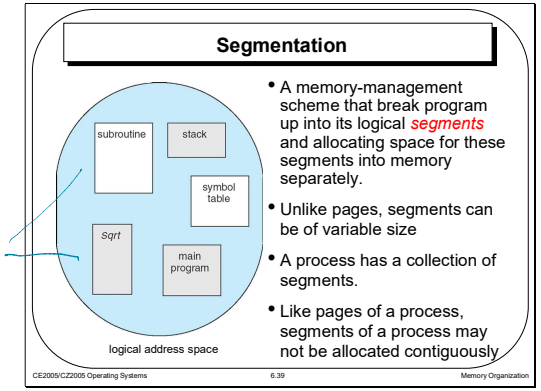
Slide 38



Process id is added to logical address. Given the pair <pid, p>, the inverted page table is searched to locate a matching entry. If entry i is a matching entry (that is, it contains <pid, p>), the page p of process pid is mapped to frame i.

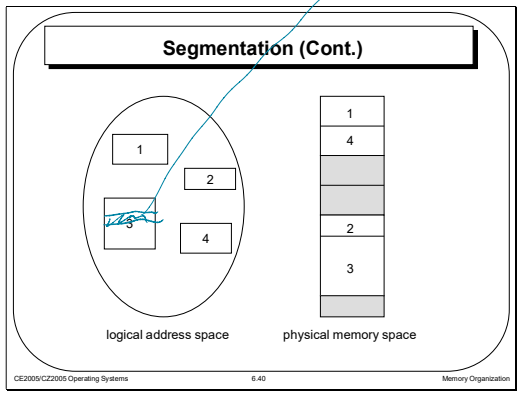
There is overhead in locating an entry, since entries are arranged in the frame number order (rather in the page number order as in conventional page table).

Slide 39

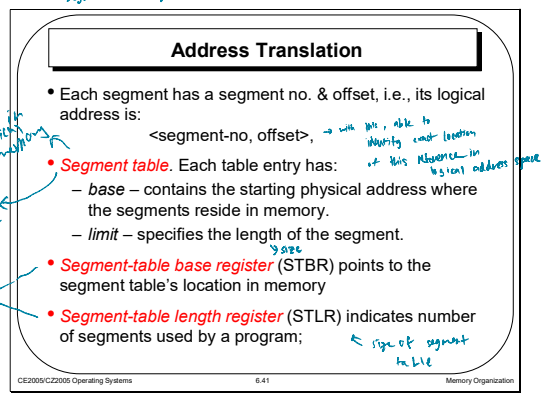


Logical address space is viewed as a collection of segments. A segment is a logical unit such as:
main program, function, local variables, global variables, stack, symbol table, arrays

Slide 40



Slide 41


Address Translation
• Each segment has a segment no. & offset, i.e., its logical address is:
<segment-no, offset>,
• Segment table. Each table entry has:
– base – contains the starting physical address where the segments reside in memory.
– limit – specifies the length of the segment.
• Segment-table base register (STBR) points to the segment table's location in memory
• Segment-table length register (STLR) indicates number of segments used by a program;
CE2005/CZ2005 Operating Systems
6.41
Memory Organization

This slide shows logical address space of a process with 4 segments. The 4 segments are loaded to different places in the physical memory.

Segmentation has a number of advantages to the paging scheme:

•**It simplifies the handling of growing data structures**.

- With segmented virtual memory, the dynamic data structure (e.g., heap) can be assigned its own segment, and the operating system will expand or shrink the segment as needed.
- If a segment that needs to be expanded is in physical memory and there is insufficient room, the operating system may move the segment to a larger area of physical memory, if available, or swap it out. The enlarged segment would be swapped back in at the next opportunity.

[2. **It allows programs to be altered and recompiled independently,**

- without requiring the entire program to be relinked and reloaded.]

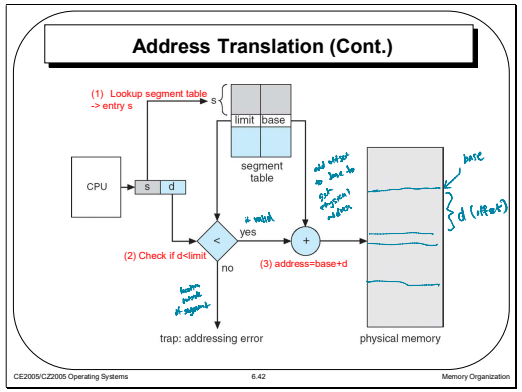
3. **It lends itself to sharing among processes**.

- Shared code/data can be put in a segment.

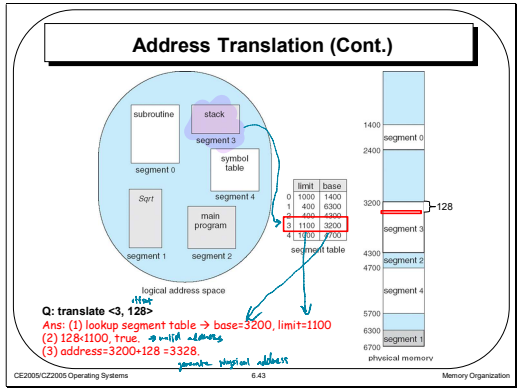
[4. **It lends itself to protection.**

- Access right can be easily defined for each segment.]

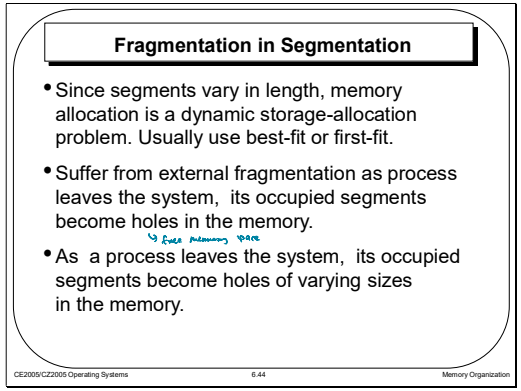
Slide 42



Slide 43

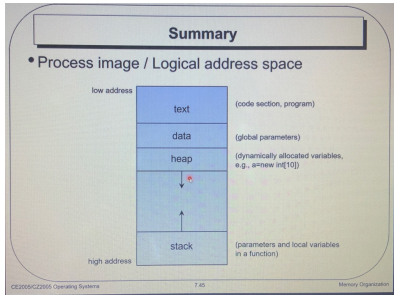


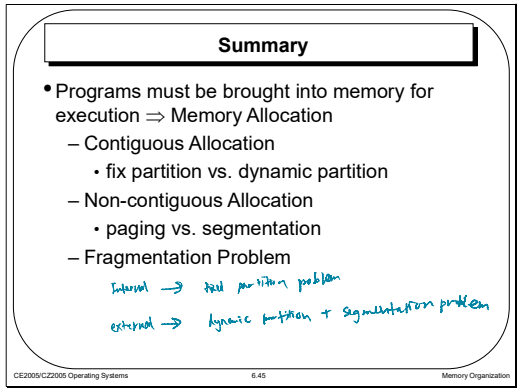
Slide 44



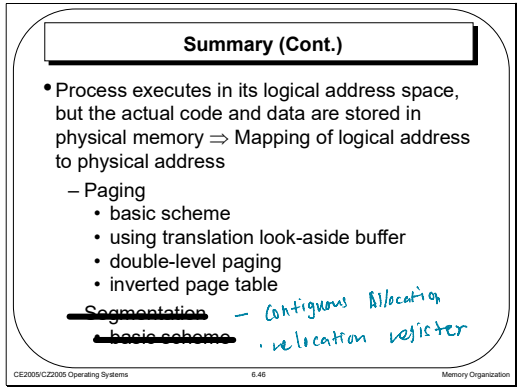
Memory allocation of a segmentation is similar to dynamic partitioning. Various dynamic storage allocation strategies (e.g., first-fit, best-fit, and worst-fit) can be used. The difference, compared to dynamic partitioning, is that a process image may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces (segments), the external fragmentation should be less.

Slide 45



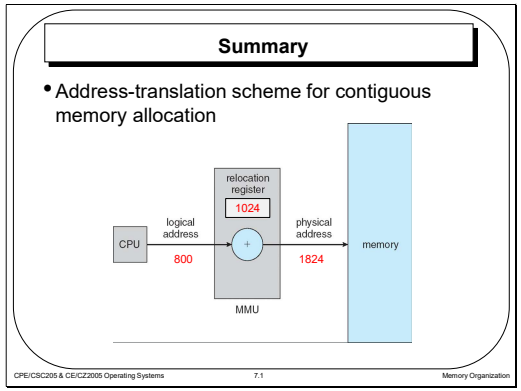


Slide 46

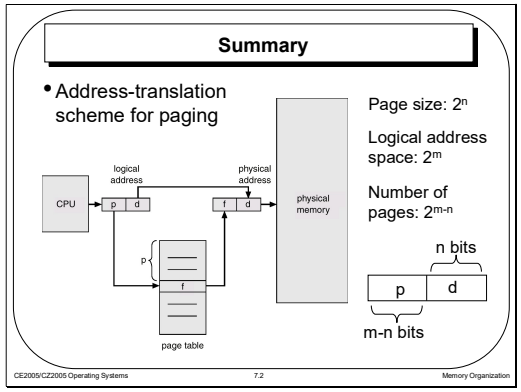


Slide 47

fixed or dynamic partitioning

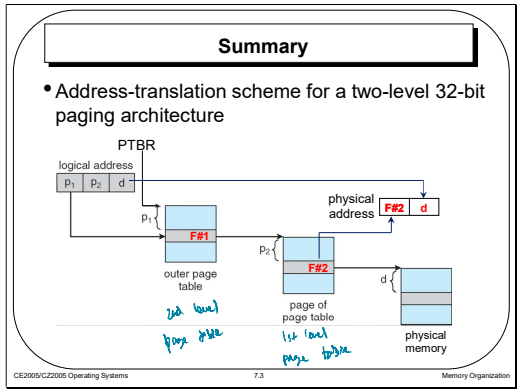


Slide 48



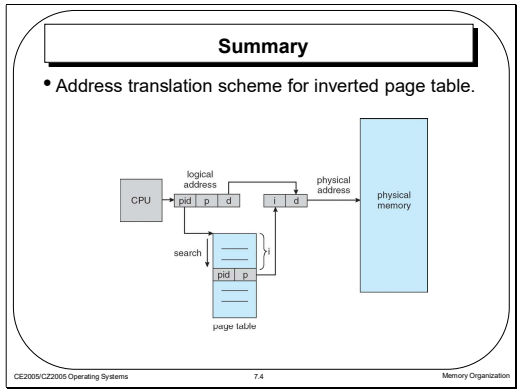
This figure suggests a hardware implementation. When a particular process is running, a register holds the starting address of the page table for that
process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address. Typically, the page number field is longer than the frame number field.

Slide 49



This figure shows the steps involved in address translation for this scheme. The outer page table always remains in main memory. The first 10 bits of a virtual address are used to index into the outer page table to find an entry for a page of the page table. If that page is not in main memory, a page fault
occurs. If that page is in main memory, then the next 10 bits of the virtual address index into the page table to find the entry for the page that is referenced by the virtual address.

Slide 50



Process id is added to logical address. Given the pair <pid, p>, the inverted page table is searched to locate a matching entry. If entry i is a matching entry (that is, it contains <pid, p>), the page p of process pid is mapped to frame i.

There is overhead in locating an entry, since entries are arranged in the frame number order (rather in the page number order as in conventional page table).

Slide 51