Memory Management

Process

- One or more threads of execution
- Resources required for execution
  - Memory (RAM)
    - Program code (“text”)
    - Data (initialised, uninitialised, stack)
    - Buffers held in the kernel on behalf of the process
  - Others
    - CPU time
    - Files, disk space, printers, etc.

Some Goals of an Operating System

- Maximise memory utilisation
- Maximise CPU utilization
- Minimise response time
- Prioritise “important” processes

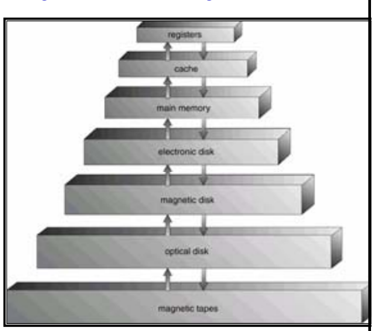
- Note: Conflicting goals $\Rightarrow$ tradeoffs
  - E.g. maximising CPU utilisation (by running many processes) increases system response time.

Memory Management

- Keeps track of what memory is in use and what memory is free
- Allocates free memory to process when needed
  - And deallocates it when they don’t
- Manages the transfer of memory between RAM and disk.

Memory Hierarchy

- Ideally, programmers want memory that is
  - Fast
  - Large
  - Nonvolatile
- Not possible
- Memory manager coordinates how memory hierarchy is used.
  - Focus usually on RAM $\Leftrightarrow$ Disk

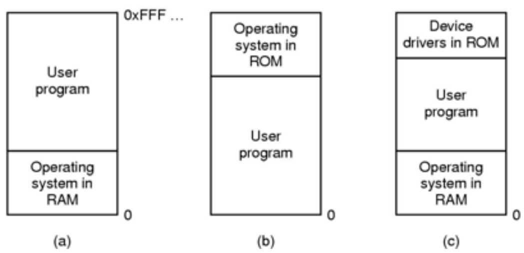


Memory Management

- Two broad classes of memory management systems
  - Those that transfer processes to and from disk during execution.
    - Called swapping or paging
  - Those that don’t
    - Simple
    - Might find this scheme in an embedded device, phone, smartcard, or PDA.

Basic Memory Management

Monoprogramming without Swapping or Paging



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

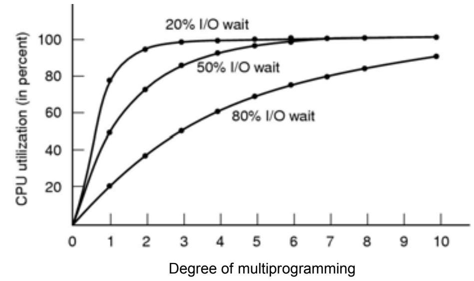
Monoprogramming

- Okay if
  - Only have one thing to do
  - Memory available approximately equates to memory required
- Otherwise,
  - Poor CPU utilisation in the presence of I/O waiting
  - Poor memory utilisation with a varied job mix

Idea

- Subdivide memory and run more than one process at once!!!!
  - Multiprogramming, Multitasking

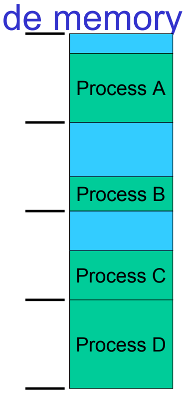
Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

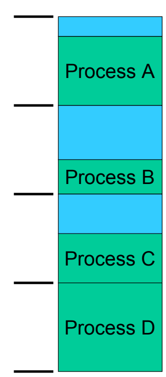
Problem: How to divide memory

- One approach
  - divide memory into fixed equal-sized partitions
  - Any process <= partition size can be loaded into any partition



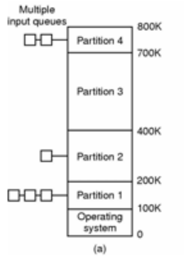
Simple MM: Fixed, equal-sized partitions

- Any unused space in the partition is wasted
  - Called internal fragmentation
- Processes smaller than main memory, but larger than a partition cannot run.

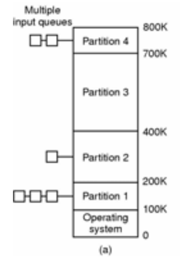


Simple MM: Fixed, variable-sized partitions

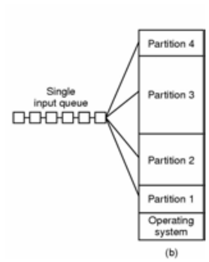
- Multiple Queues:
  - Place process in queue for smallest partition that it fits in.



- Issue
  - Some partitions may be idle
    - Small jobs available, but only large partition free



- Single queue, search for any jobs that fits
    - Small jobs in large partition if necessary
  - Increases internal memory fragmentation



Fixed Partition Summary

- Simple
- Easy to implement
- Poor memory utilisation
  - Due to internal fragmentation
- Used on OS/360 operating system (OS/MFT)
  - Old mainframe batch system
- Still applicable for simple embedded systems

Dynamic Partitioning

- Partitions are of variable length
- Process is allocated exactly what it needs
  - Assume a process knows what it needs

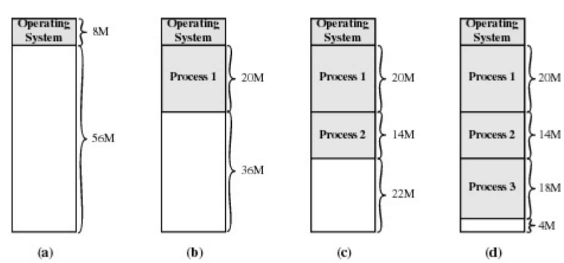


Figure 7.4 The Effect of Dynamic Partitioning

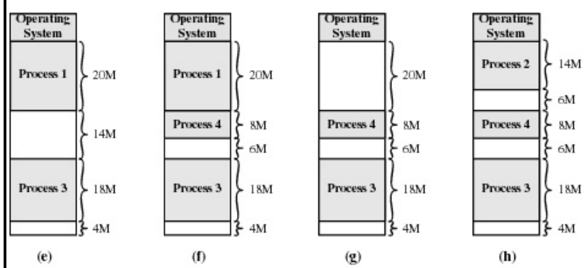


Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning

- In previous diagram
  - We have 16 meg free in total, but it can't be used to run any more processes requiring > 6 meg as it is fragmented
  - Called *external fragmentation*
- We end up with unusable holes
- 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 is done at execution time.

Recap: Fragmentation

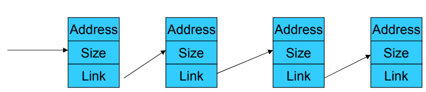
- **External Fragmentation**:
  - The space wasted external to the allocated memory regions.
  - Memory space exists to satisfy a request, but it is unusable as it is not contiguous.
- **Internal Fragmentation:**
  - The space wasted internal to the allocated memory regions.
  - allocated memory may be slightly larger than requested memory; this size difference is wasted memory internal to a partition.

Dynamic Partition Allocation Algorithms

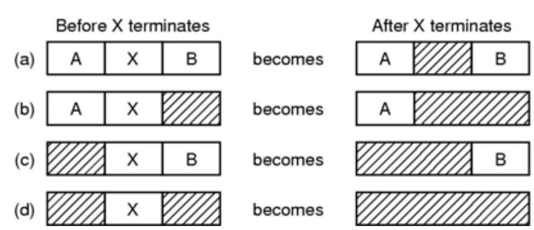
- Basic Requirements
  - Quickly locate a free partition satisfying the request
  - Minimise external fragmentation
  - Efficiently support merging two adjacent free partitions into a larger partition

Classic Approach

- Represent available memory as a linked list of available "holes".
  - Base, size
  - Kept in order of increasing address
    - Simplifies merging of adjacent holes into larger holes.



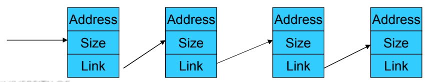
Coalescing Free Partitions with Linked Lists



Four neighbor combinations for the terminating process X

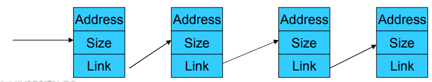
Dynamic Partitioning Placement Algorithm

- First-fit algorithm
  - Scan the list for the first entry that fits
    - If greater in size, break it into an allocated and free part
    - Intent: Minimise amount of searching performed
  - Generally results in many processes loaded, and holes at the front end of memory that must be searched over when trying to find a free block.
  - May have lots of unusable holes at the beginning.
    - External fragmentation
  - Tends to preserve larger blocks at the end of memory



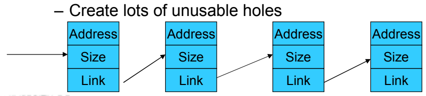
Dynamic Partitioning Placement Algorithm

- Next-fit
  - Like first-fit, except it begins its search from the point in list where the last request succeeded instead of at the beginning.
    - Spread allocation more uniformly over entire memory
      - More often allocates a block of memory at the end of memory where the largest block is found
    - The largest block of memory is broken up into smaller blocks



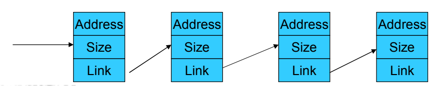
Dynamic Partitioning Placement Algorithm

- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Poor performer
    - Has to search complete list
    - Since smallest block is chosen for a process, the smallest amount of external fragmentation is left
      - Create lots of unusable holes



Dynamic Partitioning Placement Algorithm

- Worst-fit algorithm
  - Chooses the block that is largest in size (worst-fit)
    - Idea is to leave a usable fragment left over
  - Poor performer
    - Has to search complete list
    - Still leaves many unusable fragments



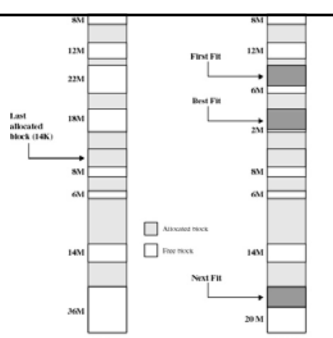


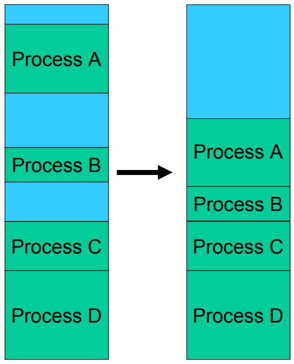
Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Dynamic Partition Allocation Algorithm

- Summary
  - First-fit and next-fit are generally better than the others and easiest to implement
- Note: Used rarely these days
  - Typical in-kernel allocators used are *lazy buddy*, and *slab* allocators
    - Might go through these later in session (or in extended)

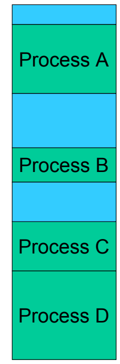
Compaction

- We can reduce external fragmentation by compaction
  - Only if we can relocate running programs
  - Generally requires hardware support



Issues with Dynamic Partitioning

- We have ignored
  - Relocation
    - How does a process run in different locations in memory?
  - Protection
    - How do we prevent processes interfering with each other



Example Logical Address-Space Layout

- Logical addresses refer to specific locations within the program
- Once running, these address must refer to real physical memory
- When are logical addresses bound to physical?

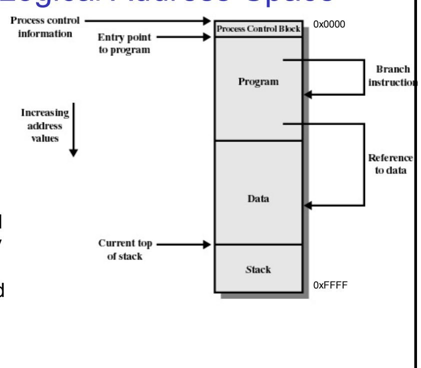
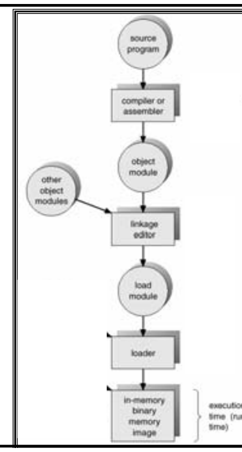


Figure 7.1 Addressing Requirements for a Process

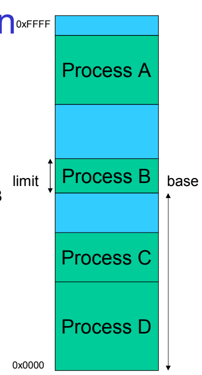
When are memory addresses bound?

- Compile/link time
  - Compiler/Linker binds the addresses
  - Must know "run" location at compile time
  - Recompile if location changes
- Load time
  - Compiler generates *relocatable* code
  - Loader binds the addresses at load time
- Run time
  - Logical compile-time addresses translated to physical addresses by *special hardware.*

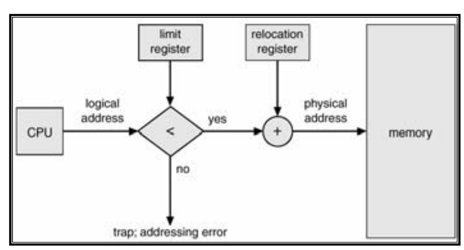


Hardware Support for Runtime Binding and Protection

- For process B to run using logical addresses
  - Need to add an appropriate offset to its logical addresses
    - Achieve relocation
    - Protect memory "lower" than B
  - Must limit the maximum logical address B can generate
    - Protect memory "higher" than B



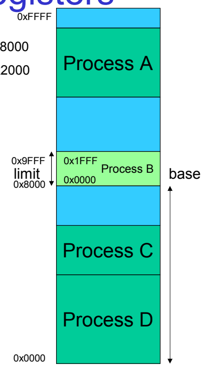
Hardware Support for Relocation and Limit Registers



36

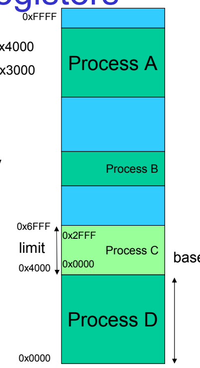
Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch



Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch

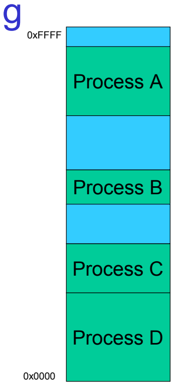


Base and Limit Registers

- Cons
  - Physical memory allocation must still be contiguous
  - The entire process must be in memory
  - Do not support partial sharing of address spaces

Timesharing

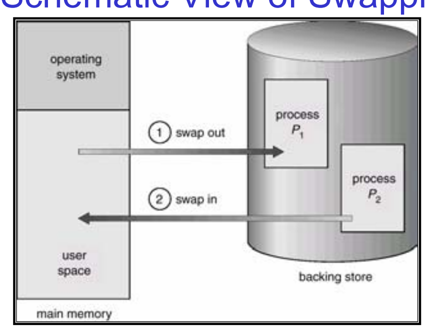
- Thus far, we have a system suitable for a batch system
  - Limited number of dynamically allocated processes
    - Enough to keep CPU utilised
  - Relocated at runtime
  - Protected from each other
- But what about timesharing?
  - We need more than just a small number of processes running at once



Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Can prioritize – lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
  - slow

Schematic View of Swapping



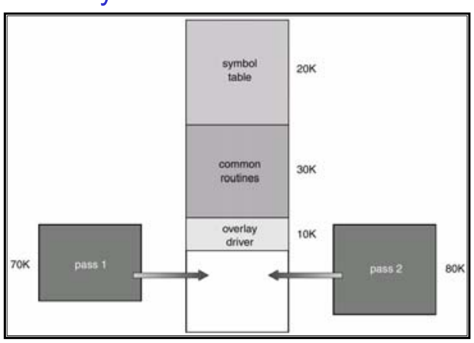
So far we have assumed a process is smaller than memory

- What can we do if a process is larger than main memory?

Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Implemented by user, no special support needed from operating system
- Programming design of overlay structure is complex

Overlays for a Two-Pass Assembler

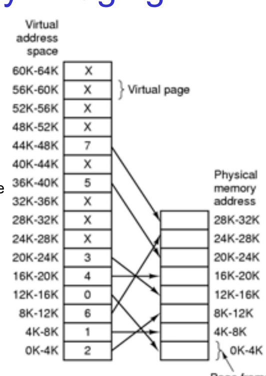


Virtual Memory

- Developed to address the issues identified with the simple schemes covered thus far.
- Two classic variants
  - Paging
  - Segmentation
- Paging is now the dominant one of the two
- Some architectures support hybrids of the two schemes

Virtual Memory - Paging

- Partition physical memory into small equal sized chunks
  - Called *frames*
- Divide each process's virtual (logical) address space into same size chunks
  - Called pages
  - Virtual memory addresses consist of a *page number* and *offset* within the page
- OS maintains a *page table*
  - contains the frame location for each page
  - Used to translate each virtual address to physical address
  - The relation between virtual addresses and physical memory addresses is given by page table
- Process's physical memory does **not** have to be contiguous



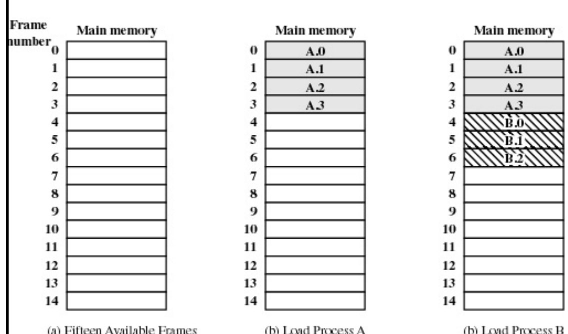
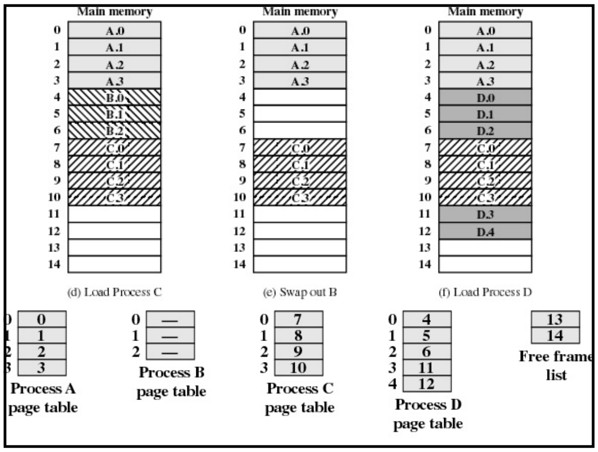


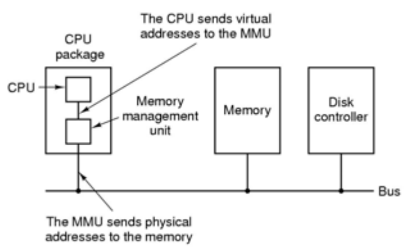
Figure 7.9 Assignment of Process Pages to Free Frames



Paging

- No external fragmentation
- Small internal fragmentation
- Allows sharing by *mapping* several pages to the same frame
- Abstracts physical organisation
  - Programmer only deal with virtual addresses
- Minimal support for logical organisation
  - Each unit is one or more pages

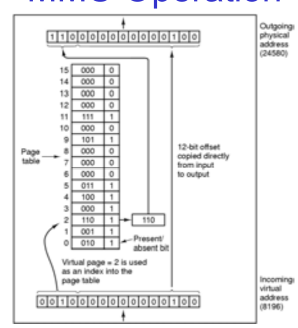
Memory Management Unit



The position and function of the MMU

MMU Operation

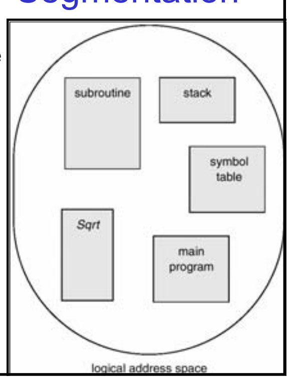
Assume for now that the page table is contained wholly in registers within the MMU



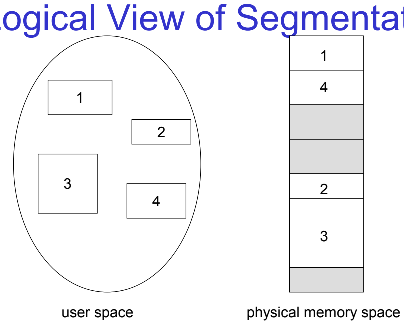
Internal operation of MMU with 16 4 KB pages

Virtual Memory - Segmentation

- Memory-management scheme that supports user's view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays



Logical View of Segmentation

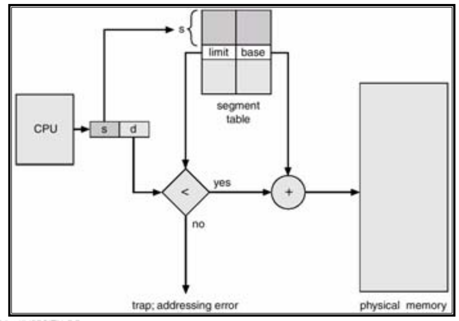


Segmentation Architecture

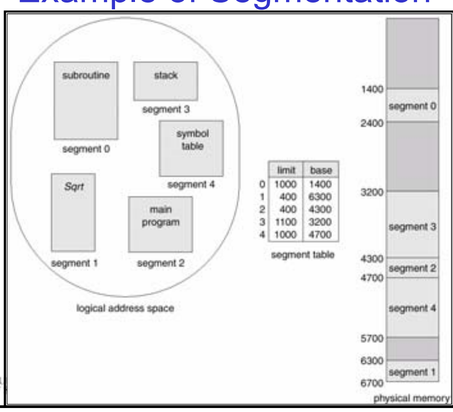
- Logical address consists of a two tuple: <segment-number, offset>,
  - Identifies segment and address with segment
- *Segment table* – each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table 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;

  segment number *s* is legal if *s* < STLR.

Segmentation Hardware



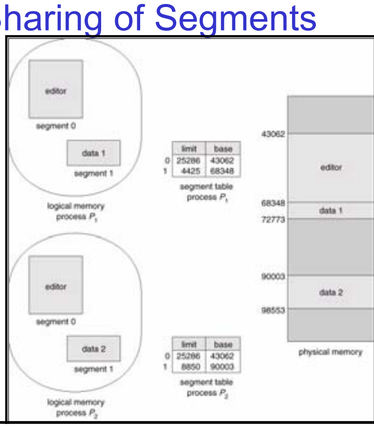
Example of Segmentation



Segmentation Architecture

- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic partition-allocation problem.
- A segmentation example is shown in the following diagram

Sharing of Segments



Segmentation Architecture

- Relocation.
  - dynamic
  - ⇒ by segment table
- Sharing.
  - shared segments
  - ⇒ same physical backing multiple segments
  - ⇒ ideally, same segment number
- Allocation.
  - First/next/best fit
  - ⇒ external fragmentation

Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Comparison of paging and segmentation