Memory Management

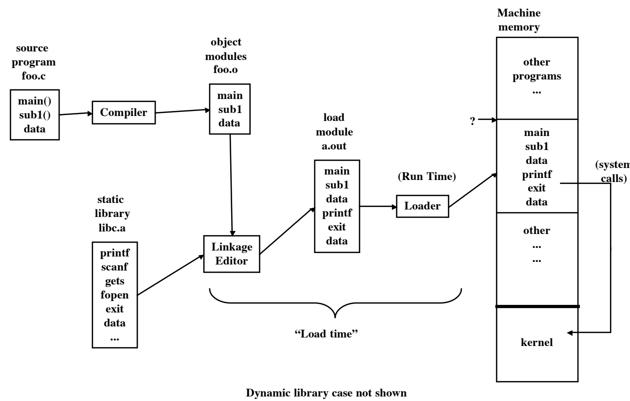
Chapter 9

Credits: Many of these slides adapted from material prepared by Profs Marchand & Logrippo, University of Ottawa

Linking and Loading

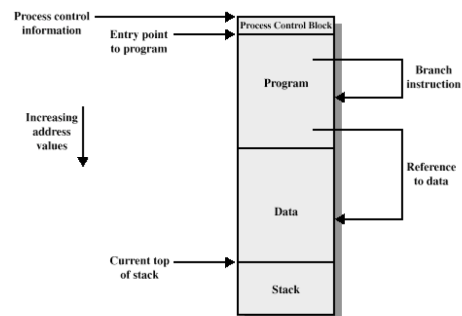
- Linking. *Bind* together the different parts of a program in order to make them into a runnable entity - Linkage editor
  - static (before execution)
  - dynamic (on demand during execution)
- Loading. Program is loaded into main memory, ready to execute. May involve address translation.
  - Absolute, relocatable
  - static, dynamic

From Source to Executable



Dynamic library case not shown

Addressing Requirements for a Process



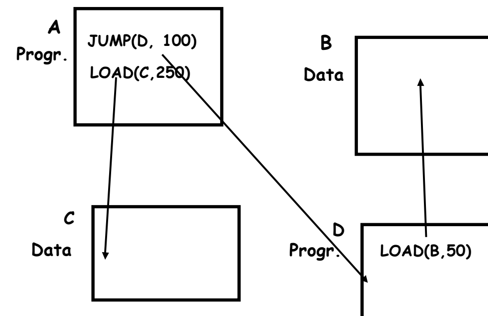
© Stallings

Binding of Instructions and Data to Memory Addresses

Address binding of instructions and data to memory addresses can happen at three different times.

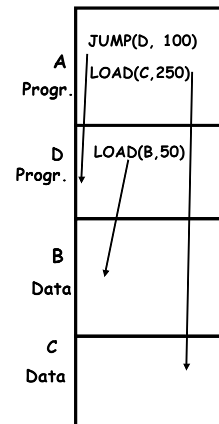
- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: If memory location is not known at compile time, compiler must generate *relocatable* code.
  - Loader knows final location and binds addresses for that location
- **Execution time**: If the process can be moved during its execution, binding must be delayed until run time. Need hardware support for address maps (e.g., *base* and *limit registers*).

Linking: Intermodule Addressing



Two- part addresses: (module name, address within module) must be translated into physical addresses

Linking



Dynamic Loading

- Routine not loaded until it is called
- 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.
- Special case: (programs too big for memory): "overlays" loaded in turn to same memory area

Dynamic Loading

- Routine not loaded until it is called
- 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.
- Special case: (programs too big for memory): "overlays" loaded in turn to same memory area

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 mapped into processes' memory address space.

Aspects of Loading

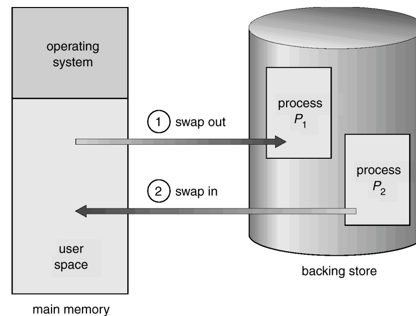
- Finding free memory for load module:
  - could be contiguous or not contiguous
- Adjust addresses in program (if required)
  - once it is known where module will be loaded

Memory Management

- A Management task carried out to accommodate multiple programs in main memory
  - If only a few programs can be in main memory, then much of the time all processes will be blocked waiting for I/O and the CPU will be idle
  - We'd like the CPU to be utilized as much of the time as possible
  - So our object is to allocate memory efficiently to pack as many programs into memory as possible

Memory Management

- In most schemes, the kernel occupies some fixed portion of main memory
- the rest of memory is shared by all the other programs
- And processes are often swapped in and out of memory, usually to different locations



Memory-Management Issues

- Relocation
- Protection
- Sharing
- Logical Organization
- Physical Organization

Relocation

- The programmer doesn't know where the program will be loaded in memory
- A program may be ***relocated*** (often) in main memory due to swapping
- This requires some kind of address translation at load or run time so the program will work no matter where it is loaded

Protection

- Processes should not reference memory locations outside their own "space"
- We cannot do this at compile time, because we do not know where the program will be loaded in memory
- This means that address references must be checked for validity at run time by hardware

Sharing

- We need a way to allow multiple processes to access a common portion of data or code without compromising protection
  - either to allow cooperating processes shared access to the same data
  - or so multiple processes can execute the same program with shared access to the same code
    - rather than use up memory by providing a separate copy for each
    - (also applies to dynamic (sharable) libraries)

Logical Organization

- Programs are packaged in modules
  - code is usually execute-only (reentrant)
  - data modules can be read-only or read/write
  - References between modules must be resolved at run time
    - Code accessing data, code accessing other code (subroutine calls, etc.)
  - Segmentation is a natural approach for all this
    - but not the only approach

Physical Organization

- Memory hierarchy: (several types of memory: from slow and large to small and fast.)
- main memory for program and data currently in use, secondary memory for swapping and paging
- moving information between levels of memory is a major concern of memory and file management (OS)
  - usually invisible to the application programmer

Physical and Virtual Memory

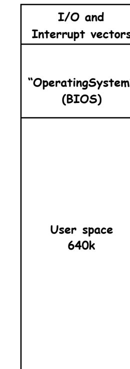
- Physical Memory: the actual main memory (RAM) of the computer
- Virtual Memory: the address space as seen by a program: can be much larger (or much smaller) than the physical memory.
  - There's a whole chapter on this.....

Simple Memory Management

- Let's assume the process image is smaller than physical memory and the program is fully loaded for execution
  - (but see *overlays*)..
- Four "simple" memory management techniques :
  - fixed partitioning
  - dynamic partitioning
  - simple paging
  - simple segmentation

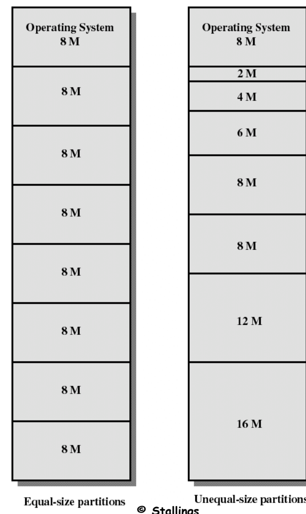
Fixed Partitioning (Single Process)

- If only one process allowed (e.g. MS-DOS):
  - one user partition
  - and one for the OS
- Not many decisions to make:
  - program either fits or it doesn't
- MS-DOS was not really an operating system!



Fixed Partitioning (Multiple Processes)

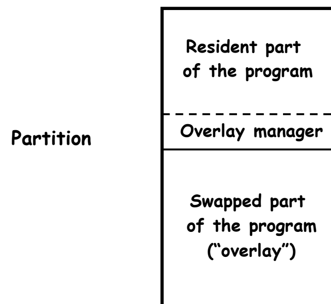
- Partition main memory into a set of non-overlapping regions called partitions
- Partitions can be of equal or unequal sizes
- ..both pictures show partitioning of 64 M main memory



Fixed Partitioning

- any program smaller than a partition can be loaded into that partition
- if no partitions are free, the operating system can swap a process out of a partition to make room
- a program may be too large to fit in a partition. The programmer would then use *overlays:*
  - when a module needed is not present the *user program* must load the missing module into a part of the program's partition, "overlaying" whatever program or data were there before

Overlays



Fixed Partitioning

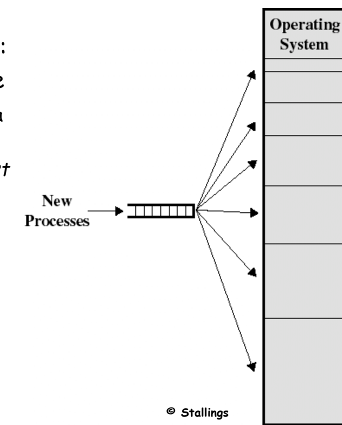
- Inefficient use of main memory. Any program, no matter how small, occupies an entire partition.
  - This leaves holes of unused memory inside the partitions: *internal fragmentation.*
- Unequal-size partitions can help
  - put small programs in small partitions
  - but there will still be holes...
- Equal-size partitions was used in IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)

"Placement Algorithm" with Partitions

- Equal-size partitions
  - If there is an available partition, a program 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 program to swap out to make room for a new program

Placement Algorithm with Partitions

- Unequal-size partitions: fed from a single queue
  - When it is time to load a process into main memory, use the *smallest available partition* that will hold the program
  - increases the level of multiprogramming
    - Does not eliminate internal fragmentation

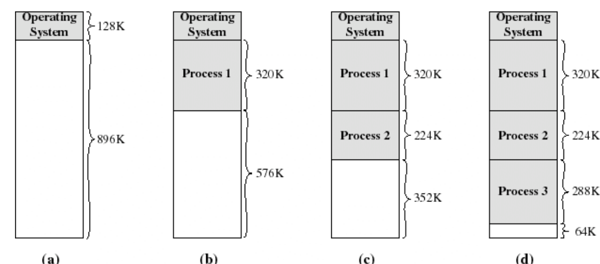


© Stallings

Dynamic Storage Allocation Problem

- Variable number of partitions, variable size (i.e. not fixed boundaries)
- Each program allocated exactly as much memory as it requires
- Eventually holes appear in main memory between the partitions
  - *This is External Fragmentation*
- Periodically, use *compaction* to shift programs so they are contiguous
  - merging free memory into one block
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks)

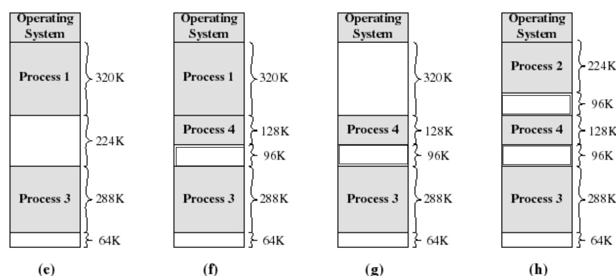
Dynamic Partitioning: an example



- Process 4 needs 128k: but after loading first 3, there is only a single hole of 64k.
- Eventually all 3 processes are blocked
- OS swaps out process 2 (closest fit) and brings in process 4 (128K)

© Stallings

Dynamic Partitioning: an example

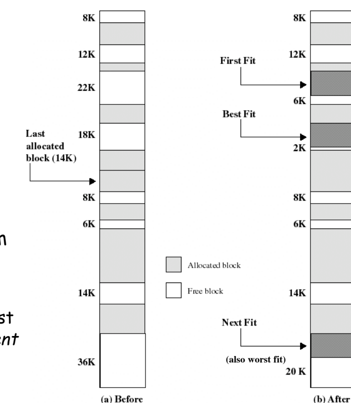


- (e,f) remove process 2, load process 4 leaves a hole of 224-128=96K (*external fragmentation*)
- (g,h) Process 1 suspended to bring in process 2 again, creating another hole of 320-224=96K...
- Left with 3 small and probably useless holes. Compaction would produce a single hole of 96+96+64=256K

© Stallings

Placement Algorithm

- Which free block to allocate to a program?
- Possible algorithms:
  - Best-fit: <u>smallest</u> hole that fits
    - creates small holes!
  - First-fit: <u>first</u> hole from beginning that fits
    - generally superior
    - Next-fit: variation: first hole after *last placement* that fits
  - Worst-fit: use largest hole
    - leaves largest leftover



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

© Stallings

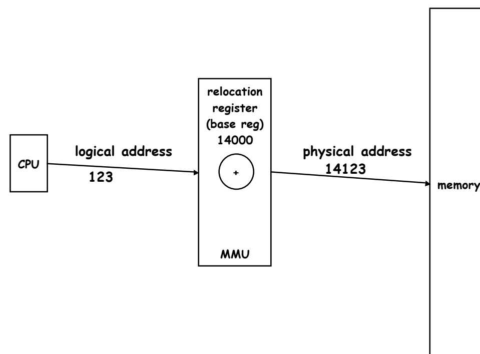
Relocation

- With swapping and compaction, a process may occupy different main memory locations during its lifetime
- So we use special hardware to perform address mapping to allow us to support relocation at run time
- We have to distinguish between *logical address* and *physical address:*

Address Terminology

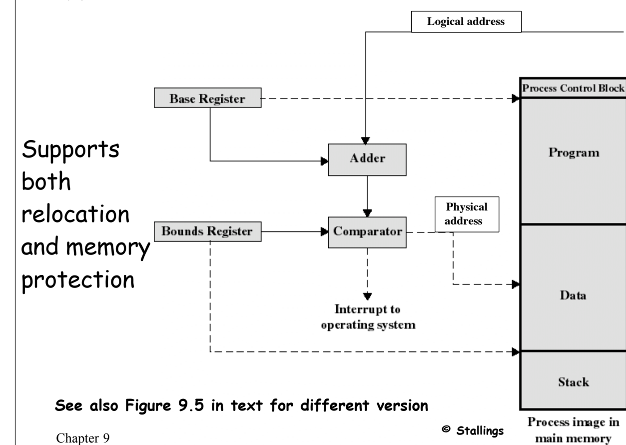
- Physical address: a physical location in main memory, from the point of view of the memory system.
- Logical address: an address as used in a program, an address as generated by the CPU when executing that program
  - i.e. an address from point of view of the CPU

Memory Management Unit (MMU)



Typical Hardware for Address Translation

Supports both relocation and memory protection



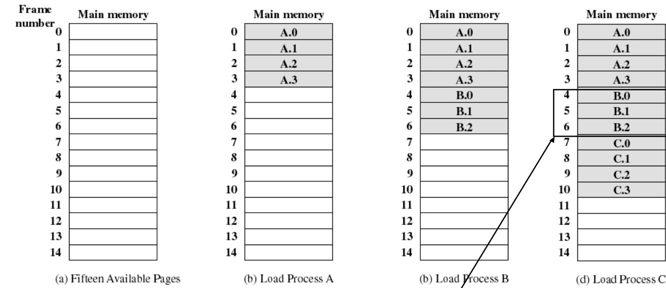
See also Figure 9.5 in text for different version

© Stallings

Paging

- Fragmentation is a significant problem with all these memory management schemes
- Let’s organize main memory into equal (small) sized chunks called *frames.*
- and divide logical address space into chunks of the same size called *pages*
- The pages of a program can thus be mapped to the available not necessarily contiguous frames in main memory,
- Result: a process can be scattered all over the physical memory!

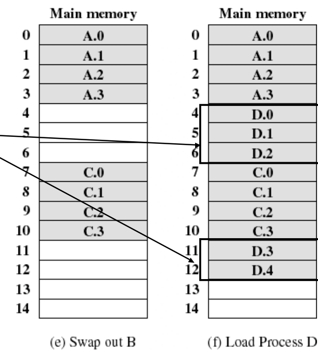
Example of Program Loading by Page



- ..now suppose that process B is swapped out

Process Loading by Page

- And load process D consisting of 5 pages
- Program D does not occupy a contiguous section of memory
- No external fragmentation!
- Internal fragmentation limited to part of the last page of each program (average 1/2 page/program).



Page Tables

- We keep a *page table* for each process
  - Contains the frame number where the corresponding page is physically located
- Page table is indexed by page number to obtain the frame number
- We also keep a Free frame list to keep track of unused frames

Main memory

Frame	Contents
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

Process A page table

Page	Frame
0	0
1	1
2	2
3	3

Process B page table

Page	Frame
0	—
1	—
2	—

Process C page table

Page	Frame
0	7
1	8
2	9
3	10

Process D page table

Page	Frame
0	4
1	5
2	6
3	11
4	12

Free frame list

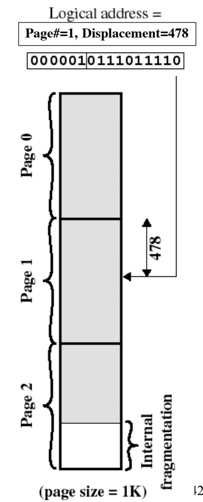
13
14

Logical Address in Paging

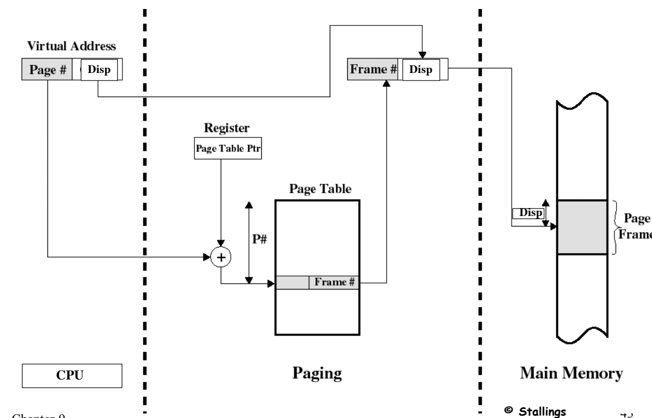
- Within each program, each logical address consists of a *page number* and an *offset* within the page
- A Memory Management Unit (MMU) is put between the CPU address bus and physical memory bus
- A MMU register holds the starting physical address of the page table of the currently running process
- Given a logical address (page number, offset) the MMU accesses the page table to obtain the physical address (frame number, offset)
- The offset is sometimes called the displacement

Logical Address in Paging

- Page size always a power of 2.
- Example: 16 bit addresses, page size = 1K, we use:
  - 10 bits for displacement and
  - 6 bits left for page number (64 pages)
- Logical address (p,d) is translated to physical address (f,d) by indexing the page table by p and appending the page displacement d to the frame number f

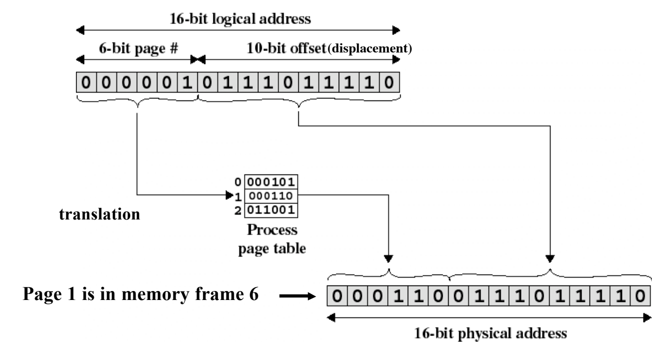


Address Translation in Paging (see also Fig 9.6)



© Stallings

Logical-to-Physical Address Translation in Paging



© Stallings

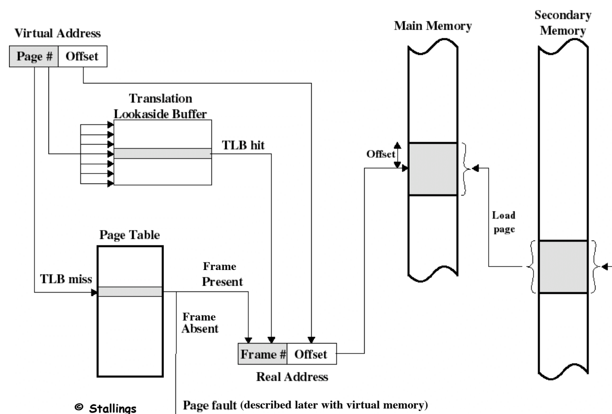
Translation Lookaside Buffer

- Because the page table is in main memory, each paged memory reference causes two physical memory accesses:
  - one to fetch the page table entry
  - one to fetch the data
- To overcome this problem a special cache is provided for page table entries
  - Translation Lookaside Buffer (TLB)
    - Contains page table entries that have been *most recently used*

Translation Lookaside Buffer

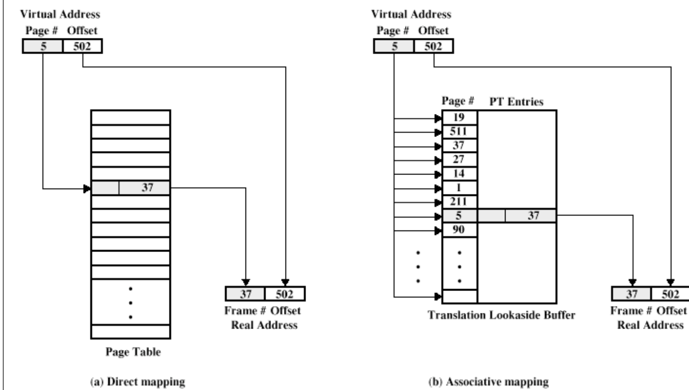
- Given a logical address, the processor examines the TLB
- If page table entry is present (a TLB hit), retrieve the frame number and the physical address is available immediately
- If page table entry is not found in the TLB (a miss), use page number to index the page table in main memory (extra memory cycle)
  - The TLB is updated to include the new page entry
  - An older entry is “bumped” to make room

Use of Translation Lookaside Buffer



© Stallings

Direct vs. Associative Mapping



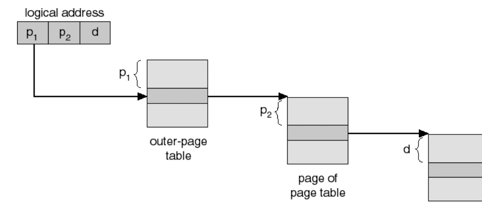
© Stallings

Page Tables and Virtual Memory

- Page tables can be very large
  - (32 - 64 bit logical addresses today!)
  - If (only) 32 bits are used (4GB) with 12 bit offset (4KB pages), a page table may have 2^{20} (1M) entries. Every entry will be at least several bytes.
- The entire page table can take up a lot of main memory.
- We may have to use a 2-level (or more) structure for the page table itself.

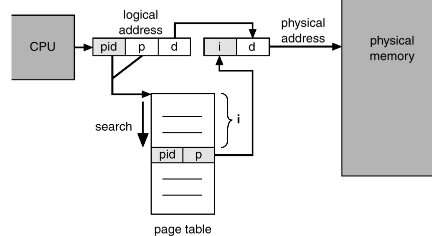
Multilevel Page Tables

- With a 2 level page table (386, Pentium), the page number is split into two numbers p1 and p2
- p1 indexes the outer page table (directory) in main memory whose entries point to a page containing page table entries for some range of virtual memory
- The second level entry is indexed by p2.
- Except for the directory (outer table), page tables entries can be swapped in and out as needed



Inverted Page Table

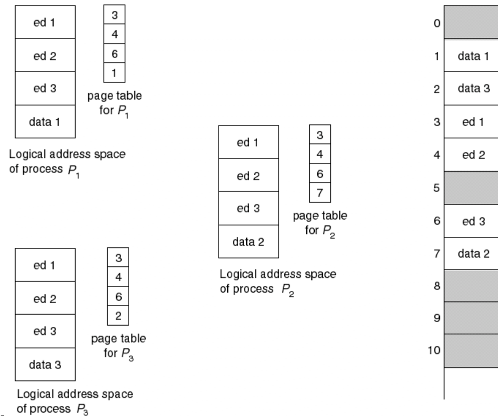
One entry per FRAME rather than one per PAGE
Search for matching page #
Hash table used to speed access,
but TLB is critical!



Sharing Pages

- A paging system can allow more than one process to share access to the same frames.
- Often used to share program code (which must be reentrant = non self-modifying)
- The page tables for multiple processes will have entries pointing to the same code frames
  - But there is only one copy of each frame in memory
- Each user will have separate data pages

Sharing Pages: a Text Editor



Segmentation

- Processes are actually organized into logical parts (segments):
  - one or more executable segments
  - one or more data segments
  - stack segment
- It is possible to organize memory in the same way.

Segmentation

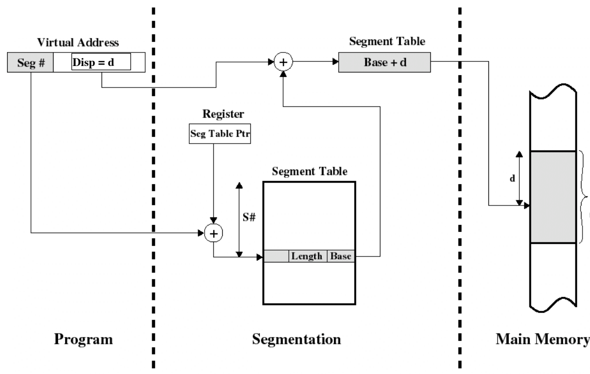
- Divide each program into unequal size blocks called segments
- When a process is loaded into main memory, its individual segments can be located anywhere
- The methods for allocating memory to segments are those we have seen so far: just replace *process* by *segment*
- Each segment has a name and a length
- Because segments are of unequal size, this is similar to *dynamic partitioning* (at the segment level).

Simple Segmentation

- Segment table per process
- For each segment:
  - starting physical address.
  - length (for protection)
- A CPU register points to the segment table
- Logical address (segment, offset) = (s,d)
- Entry s in the segment table gives base physical address k and the length l of that segment
- The physical address is k + d
  - And if d > l, the address is invalid

Address Translation in Segmentation

(See also Figure 9.18)

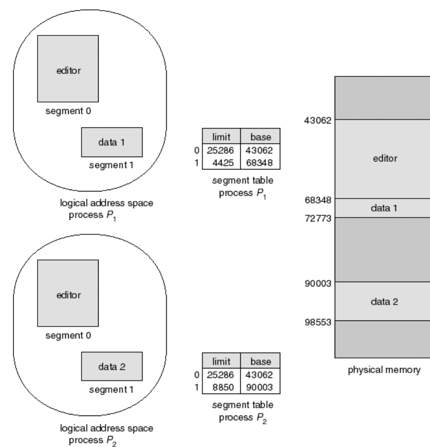


© Stallings

Sharing in Segmentation Systems

- Segment table entries of 2 different processes can point to the same physical segment
- Example: shared copy of the the code segment for the text editor
- Each user still needs its own private data segment
- More logical than sharing pages

Segment Sharing: Text Editor Example



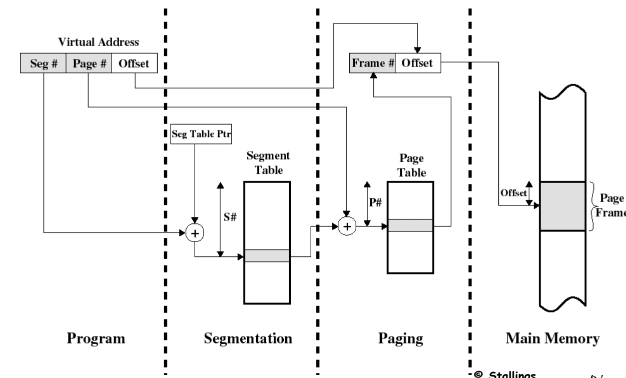
Evaluation of Simple Segmentation

- Advantage: memory allocation unit is a logically natural view of program
  - Segments can be loaded individually on demand (*dynamic linking*).
  - Natural unit for protection purposes
  - No internal fragmentation
- Disadvantage: same problems as dynamic partitioning:
  - External fragmentation
  - Unlike paging, it is not transparent to programmer
  - Requires entire segments to be resident in memory

Combining Segmentation and Paging

- Combines advantages of both
- Several combinations exist. One example:
  - Each process has:
    - one segment table
    - one page table per segment
  - Virtual address consists of:
    - segment number: index the segment table to get starting address of the page table for that segment
    - page number: index that page table to obtain the physical frame number
    - offset: to locate the word within the frame
- Segment and page tables can themselves be paged!

Address Translation in Combined Segmentation/Paging System(see also Fig 9.21)



© Stallings

Advantages of Segmentation + Paging

- Solves problems of both loading and linking.
  - Linking a new segment amounts to adding a new entry to a segment table
- Segments can grow without having to be moved in physical memory (just map more pages!)
- Protection and sharing can be done at the 'logical' segment level.