TDIU11

Operating systems

Memory Management

[SGG7/8/9] Chapter 8

Christoph Kessler, IDA,
Linköpings universitet.

Contents: Memory Management

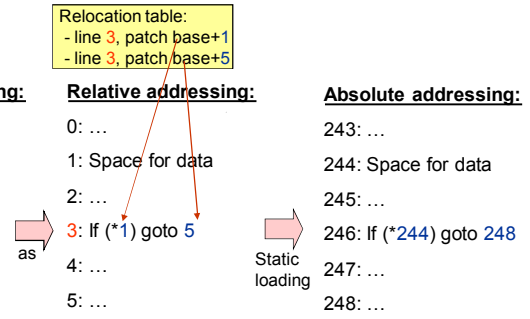
- Background
  - Relocation
  - Dynamic loading and linking
- Swapping
- Contiguous Allocation
- Paging
- Segmentation

Background

- To be run, a program must be brought into memory and placed within a process
- Long-term scheduler / Medium-term scheduler
  - Not all processes fit into memory (i.e., memory is a limited resource)
  - **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run
  - e.g., relocation

How to generate code?

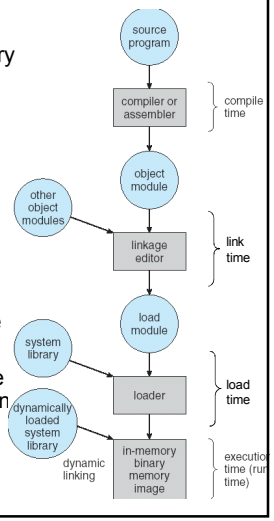
- When the compiler/assembler/linker generates code, how does it handle …
  - Absolute Jumps?
  - Adresses of global variables?



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time / Link time**:
  - If memory location known a priori, ***absolute code*** can be generated;
  - must recompile code if starting location changes
- **Load time**: Must generate ***relocatable code*** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
  - Need hardware support for address maps (e.g., *base* and *limit registers, MMU*).



Static vs. Dynamic Loading

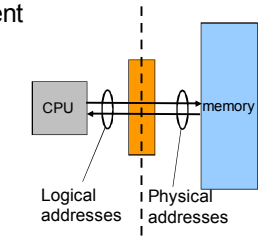
- **Static Loading:**
  - Move the compiled and linked image into program memory, patch remaining relative addresses, and start execution.
- **Dynamic loading:**
  - postpone loading of some routine until it is called
  - Better memory-space utilization
    - unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases
  - No special support from OS required
    - implemented through compiler runtime system
  - Example: Java dynamic class loading

Static vs. Dynamic Linking

- **Static linking:**
  - Copy together a program from its modules and libraries used
  - Relocation (patching) of relative addresses with new ones
  - update relocation table and table of externally visible symbols
- **Dynamic linking**:
  - "true" linking postponed until runtime
  - Small piece of code, ***stub***, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with a call to the address of the routine, and executes the routine
  - OS needed to check if routine is in processes' memory area or other accessible memory area
  - Dynamic linking is particularly useful for libraries

Logical vs. Physical Address Space

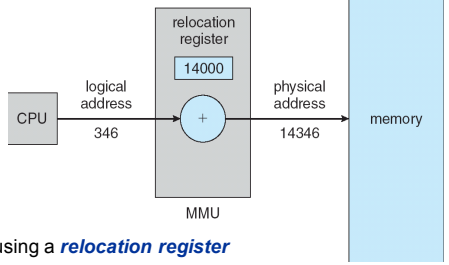
- The concept of a ***logical address space*** that is bound to a separate ***physical address space*** is central to proper memory management
  - **Logical address**
    - generated by the CPU
    - also referred to as *virtual address*
  - **Physical address**
    - address seen by the memory unit



- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical (virtual) and physical addresses differ in run-time address-binding scheme

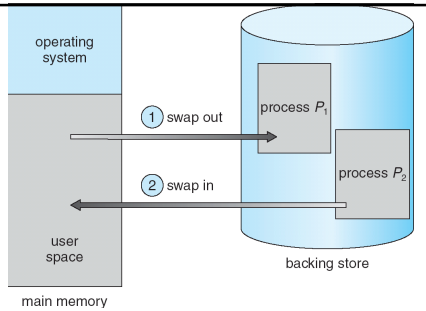
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses



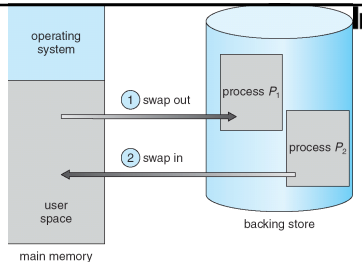
Dynamic relocation using a ***relocation register***

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

Swapping



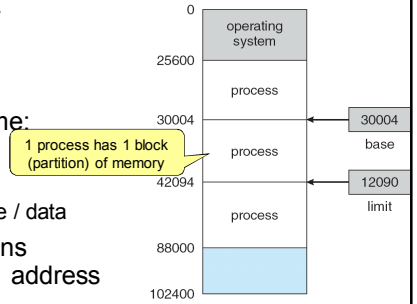
- ***Roll out, roll in***
  - swapping variant used for priority-based scheduling algorithms
  - lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time
  - total transfer time is directly proportional to the amount of memory swapped

Swapping is costly!

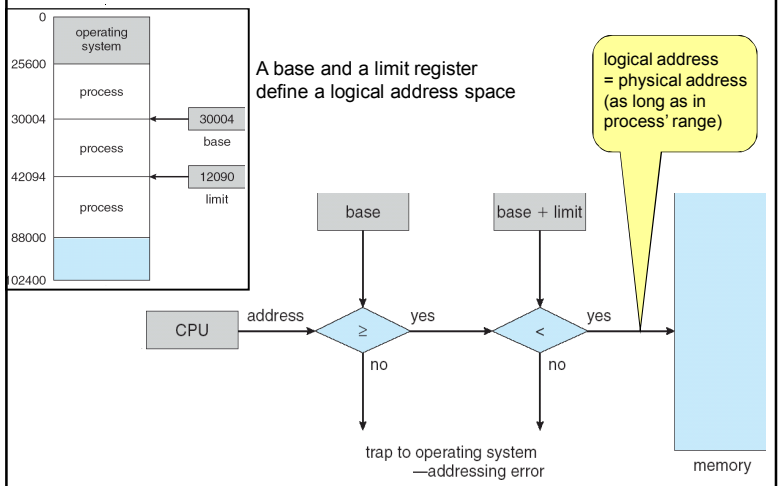
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Contiguous Allocation

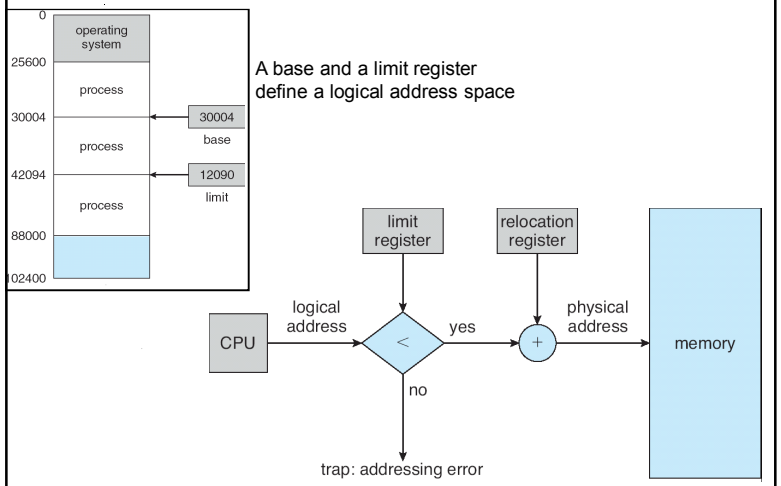
- **Dual mode**: Main memory separated into 2+ partitions:
  - Resident OS kernel, usually held in low memory (incl. interrupt vector)
  - User processes then held in high memory
- **Multi-partition allocation**
  - Relocation-register scheme:
    - protect user processes from each other,
    - and from changing OS code / data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses



HW address protection (1)
with base and limit registers



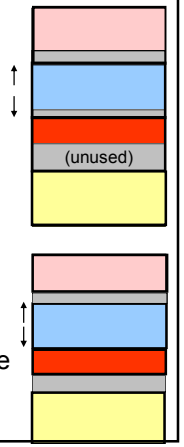
HW address protection (2)
with relocation and limit registers



Contiguous multi-partition allocation

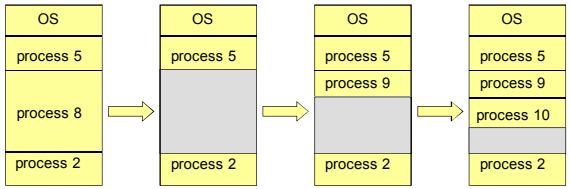
2 variants:

- **Fixed partition scheme**
  - All partitions have equal (fixed) size
    - \+ Simple model
    - – Degree of multiprogramming limited by #partitions
    - – Internal fragmentation
- **Variable partition scheme**
  - Size of loaded program dictates partition size
    - – External fragmentation



Contiguous Allocation (Cont.)

- ***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
- OS maintains information about:
  a) allocated partitions
  b) free partitions (holes)



Dynamic Storage-Allocation Problem

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

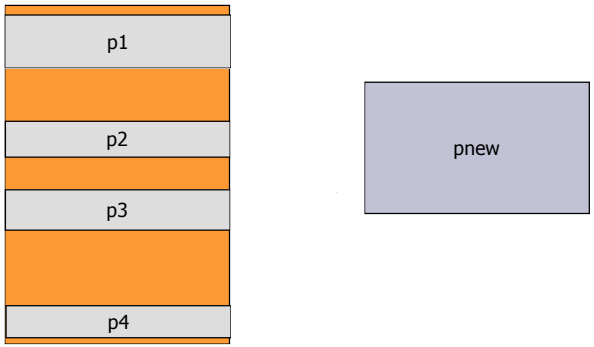
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough;
  - must search entire list, unless ordered by size.
  - Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole;
  - must also search entire list.
  - Produces the largest leftover hole.

Simulations show: First-fit and best-fit better than worst-fit in terms of storage utilization

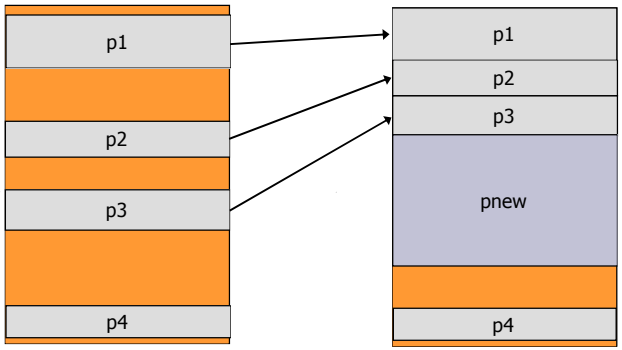
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block (also known as garbage collection)
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Example of Compacting

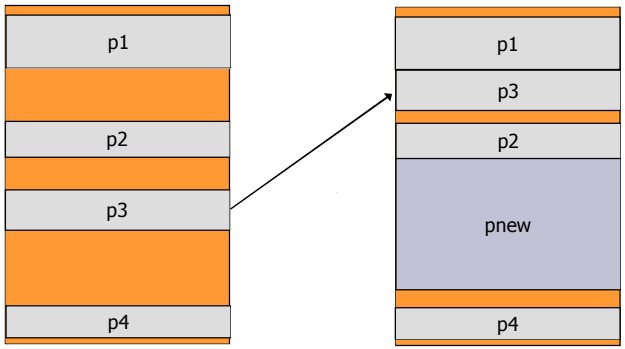


Example of Compacting: Solution 1



Move all occupied areas to one side until there is a hole large enough for pnew

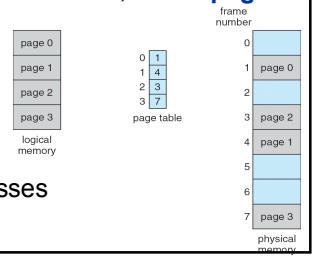
Example of Compacting: Solution 2



Search and select one (or a few) processes to move to free a hole large enough…

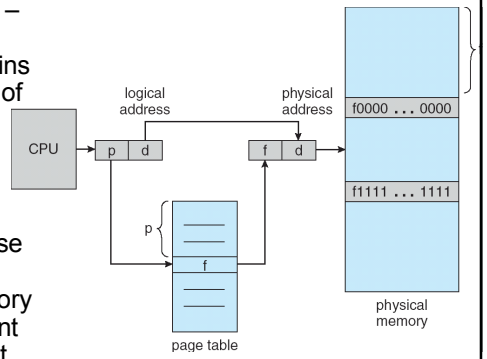
Paging

- Goals:
  - ☺ Physical address space of a *process* can be noncontiguous
  - ☺ Process is allocated physical memory whenever the latter is available – no external fragmentation
- Divide physical memory into fixed-sized blocks called **frames.** Frame size is power of 2, between 512 bytes and 8192 bytes.
- Divide logical memory into blocks of same size, called **pages**.
- Keep track of all *free frames*
- To run a program of size *n* pages, need to find *n* free frames and load program
- Set up a **page table** to translate logical to physical addresses
- ☹ Internal fragmentation

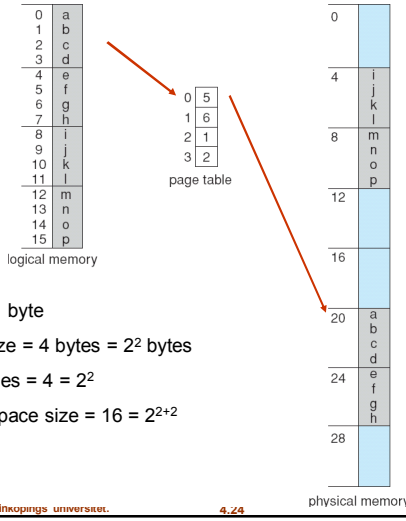


Paging: Address Translation Scheme

- Address generated by CPU is divided into:
  - ***Page number*** *(p)* – index into a ***page table*** which contains the base address of each page in physical memory
  - ***Page offset*** *(d)* – combined with base address to define the physical memory address that is sent to the memory unit



Tiny Paging Example



1 character = 1 byte

Frame/page size = 4 bytes = 2^2 bytes

Number of pages = 4 = 2^2

Logical addr. space size = 16 = 2^{2+2}

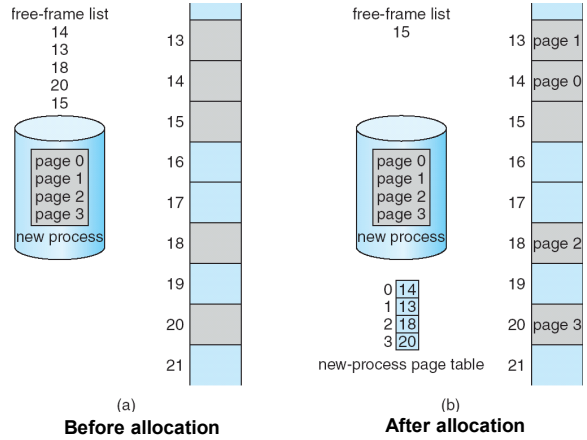
Physical address space size = 32 bytes

Paging – Remarks

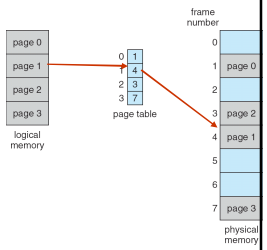
- *Internal* fragmentation with paging
  - Worst case fragmentation = 1 frame – 1 byte
  - Average fragmentation = 0.5 * frame size
- So small frame sizes are desirable?
  - But each page table entry takes memory to track
- Page sizes growing over time

- Process view and physical memory are now very different
  - By implementation, a process can only access its own memory

Allocating frames from free-frame list



Implementation of the Page Table



- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *Page-table length register* (PRLR) indicates size of the page table
- Every data/instruction access requires 2 memory accesses.
  - One for the page table and one for the data/instruction.
- Solve the two-memory-access problem
  - by using a special fast-lookup cache (in hardware):

    translation look-aside buffer (TLB)

    - Implements an **associative memory** ⟶

Associative Memory

- **Associative memory** = content-addressable memory
  - ADT Dictionary

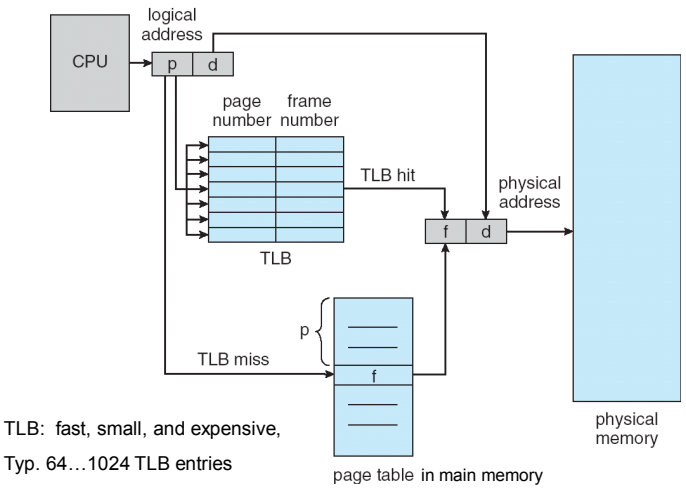
Page number	Frame number
p	*f*

TLB: Address translation (*p, f*)

- If *p* is in TLB, get frame number *f* out
- Otherwise get frame number from page table in memory

- TLB realized in hardware → parallel search for contents

Paging Hardware With TLB



TLB: fast, small, and expensive,

Typ. 64…1024 TLB entries

Effective Access Time

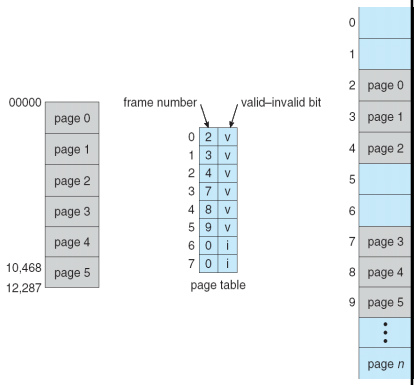
- Memory cycle time: t
- Time for associative lookup: ε
- TLB hit ratio α
  - percentage of times that a page number is found in TLB
- **Effective Access Time** (EAT):

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

Example: For t =100 ns, ε = 20 ns, α = 0.8: EAT = 140 ns

Memory Protection

- implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - **"valid"**: the associated page is in the process' logical address space, and is thus a legal page
  - **"invalid"**: the page is not in the process' logical address space

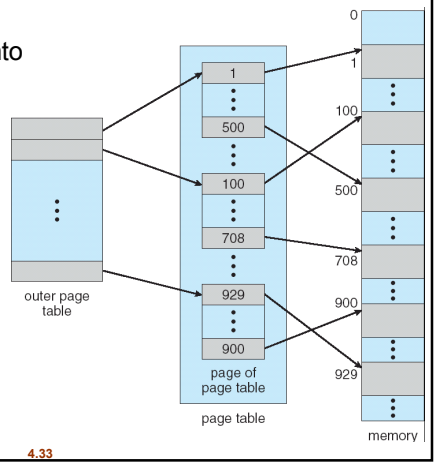


Page Table Structure

- The "large page table problem"
  - Most modern systems support large logical address spaces (2^{32} … 2^{64} bytes)
  - For 2^{32} logical addresses with a page size 2^{12} = 4096, page table has 2^{20} = 1M entries, thus needing 4 MB physical memory
- Page table structures:
  - Hierarchical Paging: "page the page table"
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table:



Two-Level Paging Example

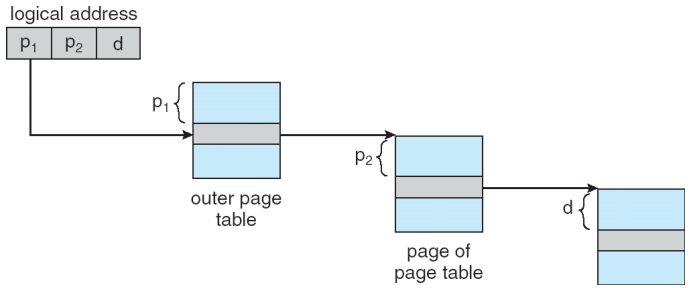
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address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

page number		page offset
p_i	p_2	d
10	10	12

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

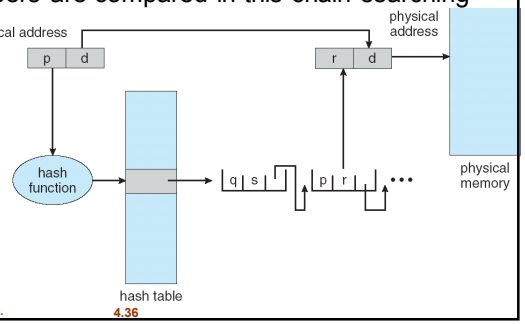
Address-Translation Scheme

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



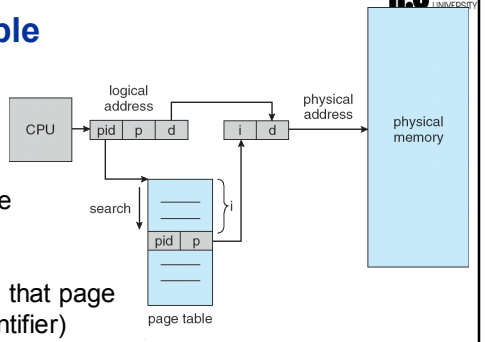
Hashed Page Tables

- Common in address spaces > 32 bits
- 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.



Inverted Page Table

- One entry for each real page of memory
- Entry: <*pid, p*>
  p: virtual address of the page stored in that real memory location,
  pid: process that owns that page (as address space identifier)

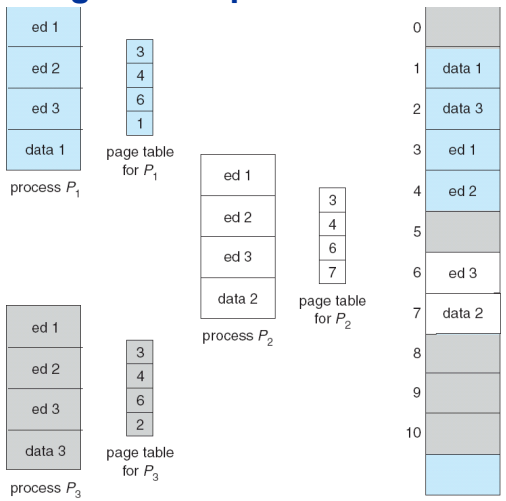


- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
- Examples: UltraSPARC, PowerPC, Itanium

Shared Pages – Easy with paged memory!

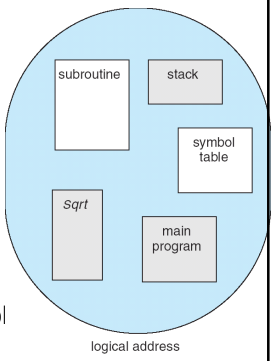
- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
- Shared pages are hard to implement when inverted page tables are used
  - Only 1 virtual page entry for every physical page

Shared Pages Example

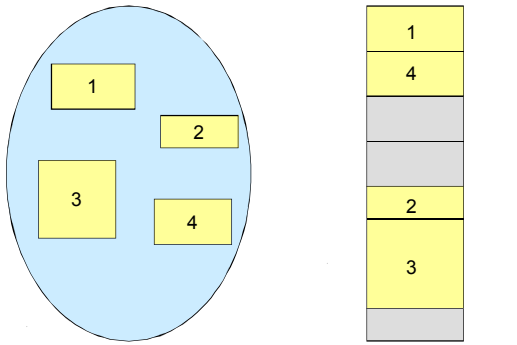


Segmentation

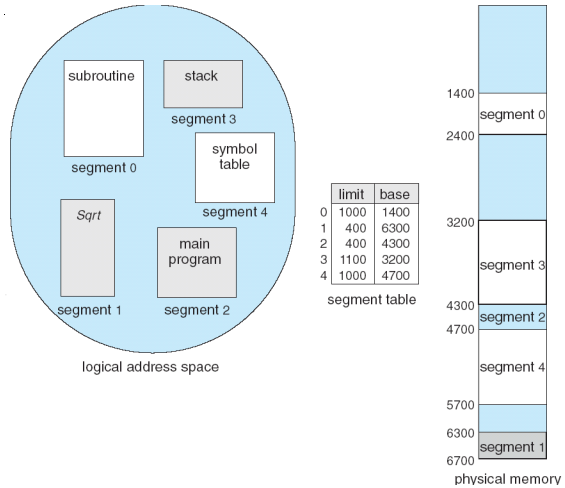
- Memory-management scheme that supports a *user view of memory*:
- A program is a collection of segments.
- A **segment** is a logical unit such as:
  - main program, procedure, function, method,
  - object, local variables, global variabl
  - common block, stack,
  - symbol table, arrays
- Idea: allocate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



Logical View of Segmentation



Example of Segmentation

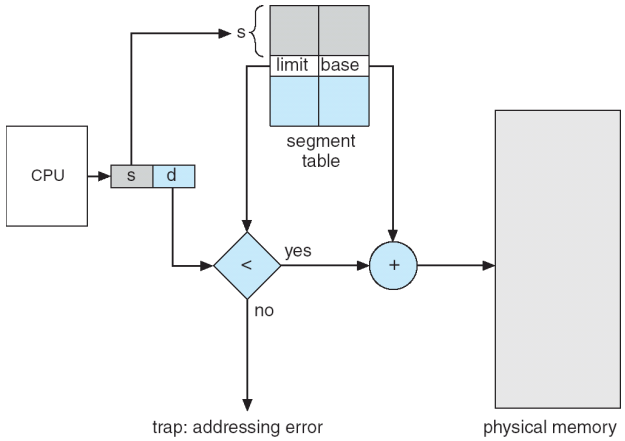


Segmentation Architecture

- Logical address is a pair *<segment-number, offset>*
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - *base* – physical starting address where the segment resides in memory
  - *limit* – specifies the length of the segment
- 2 registers (part of PCB):
  - *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 Architecture: Address Translation



Segmentation Architecture (cont.)

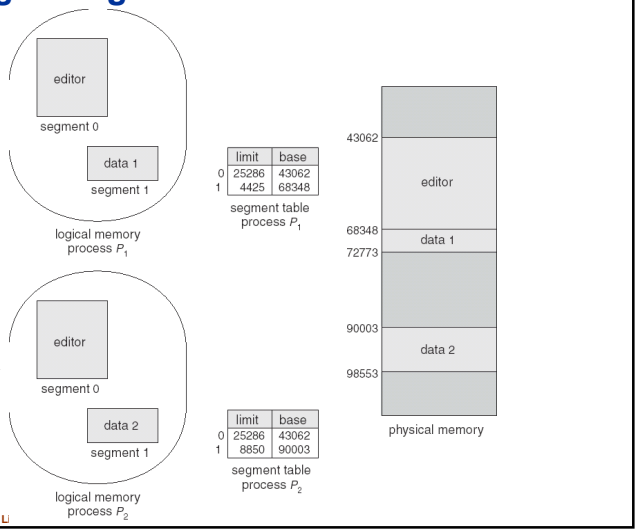
- **Relocation**.
  - dynamic
  - by segment table
- **Sharing**.
  - shared segments
  - same segment number
- **Allocation**.
  - first fit/best fit
  - external fragmentation

- **Protection**. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - **r**ead / **w**rite / e**x**ecute privileges

  Protection bits in table entries

Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Sharing of Segments



Combining Segmentation and Paging

- Each segment is organized as a set of pages.
- Segment table entries refer to a page table for each segment.
- TLB used to speed up effective access time.
- Common in today's operating systems (e.g. Solaris, Linux).