Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be run.

- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.

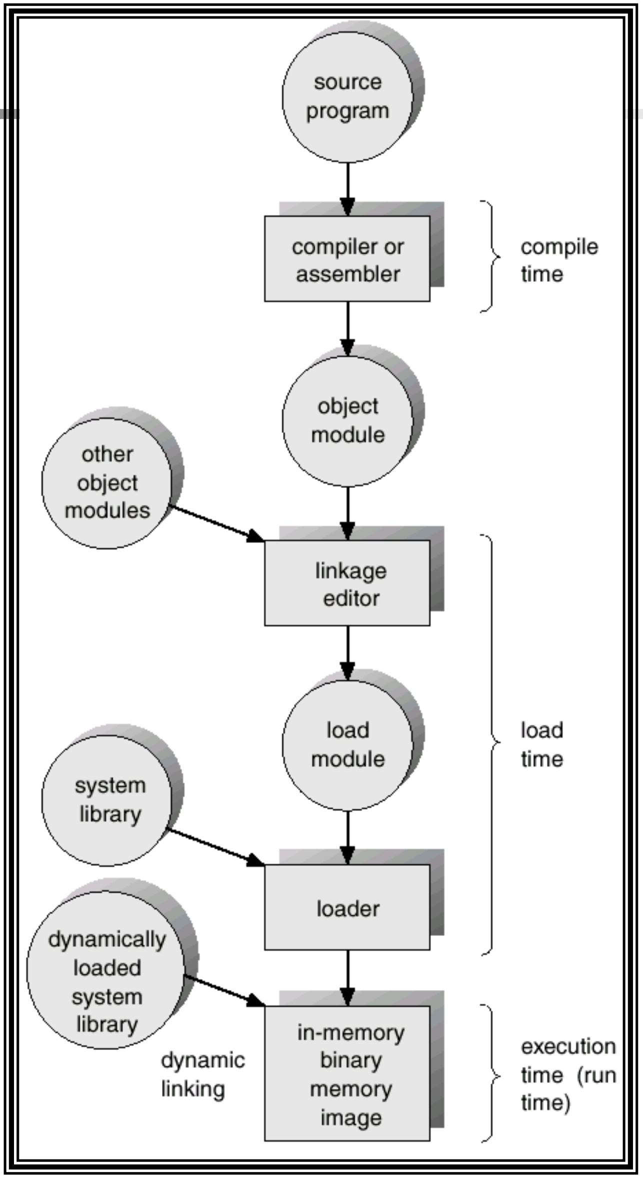
- User programs go through several steps before being run.

Binding of Instructions and Data to Memory

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

- **Compile 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*).

Multistep Processing of a User Program



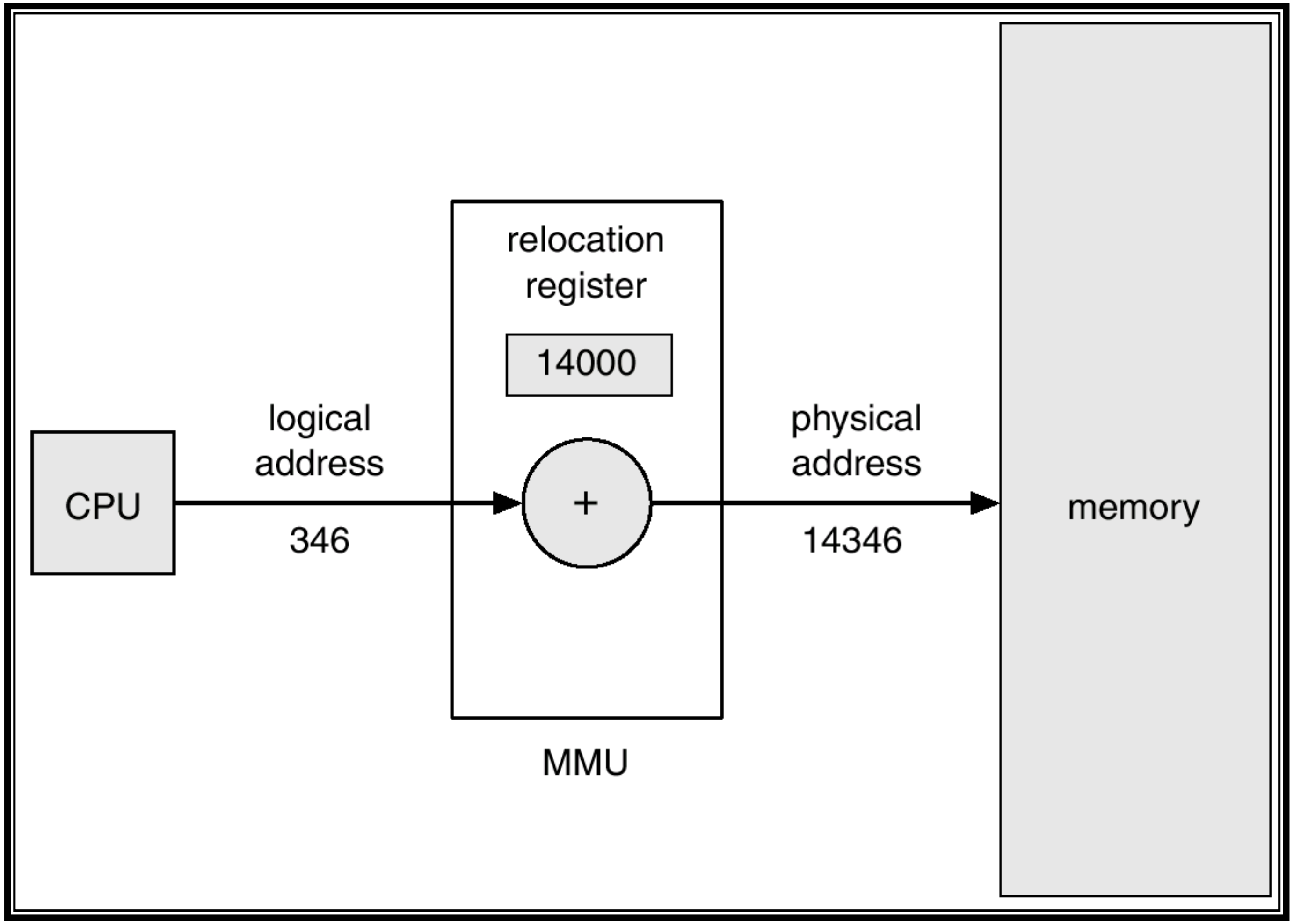
Logical vs. Physical Address Space

- 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.

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, 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



Dynamic Loading

- Routine is 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.

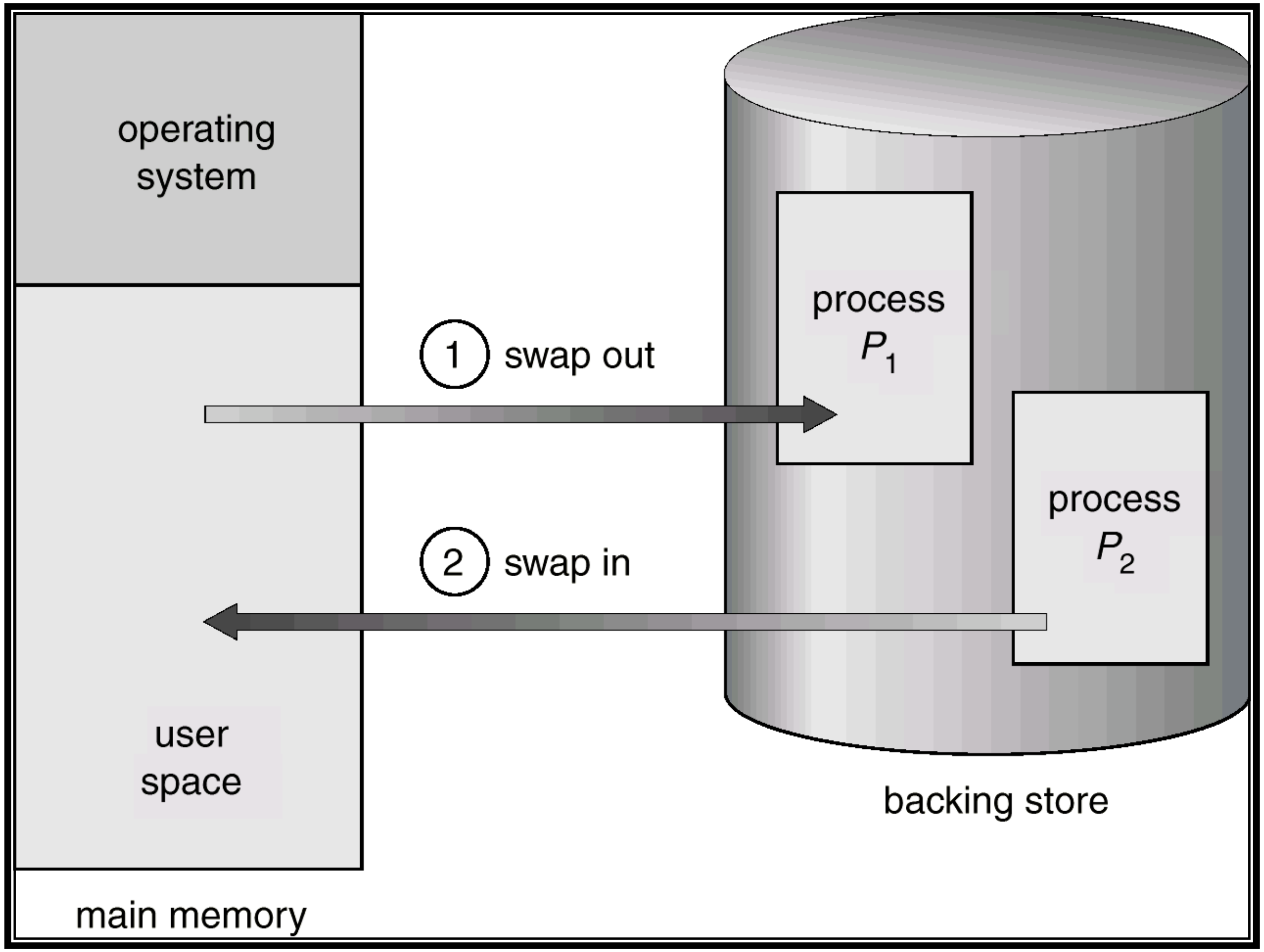
Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

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

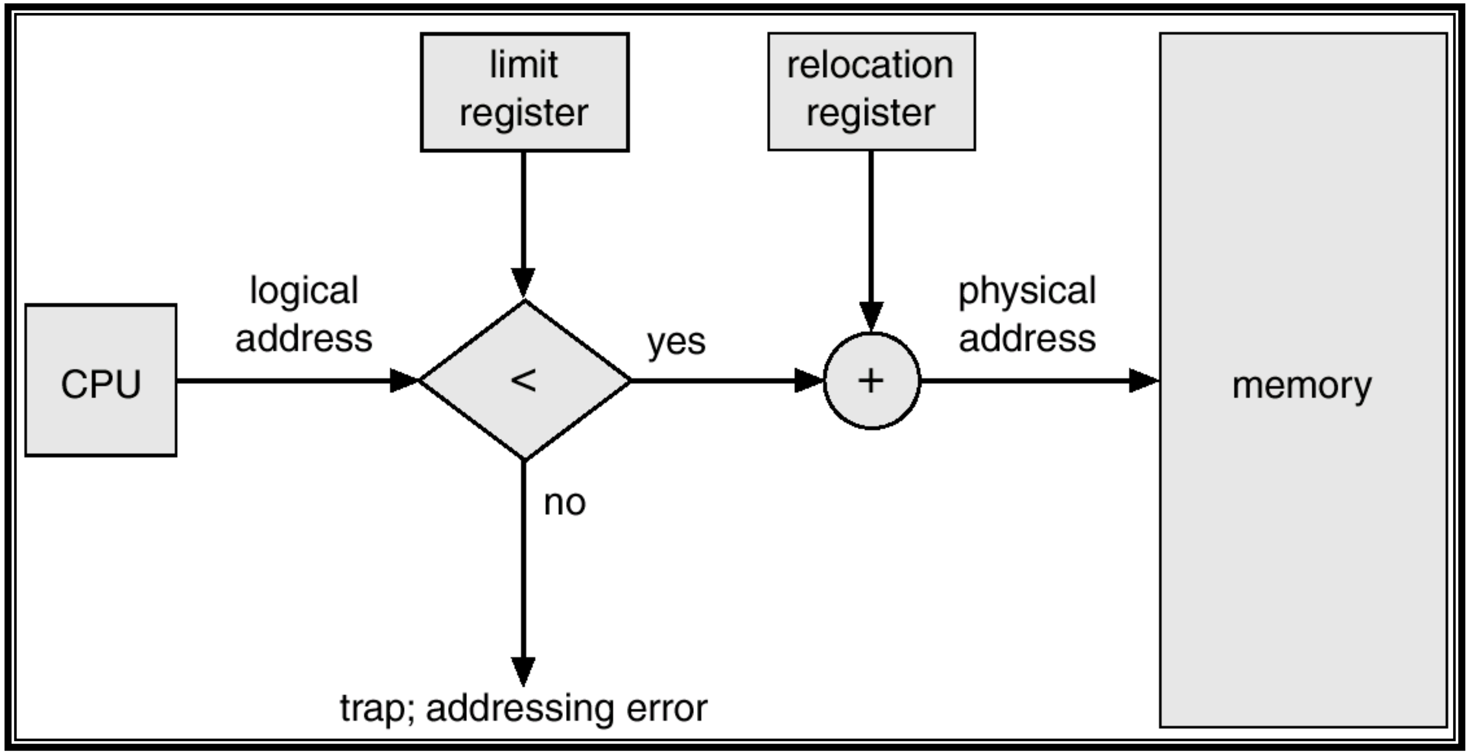
Schematic View of Swapping



Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers



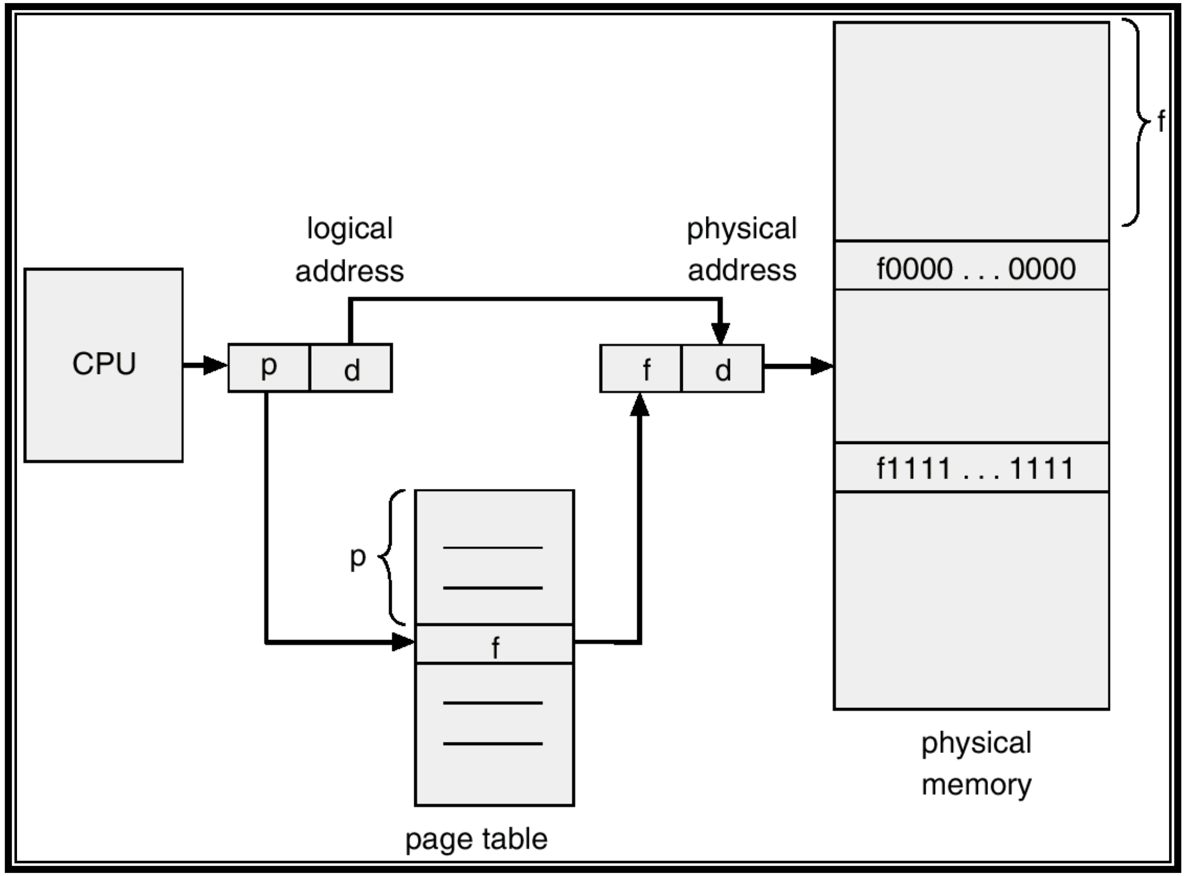
Paging

- The physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (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 .
- Actually not all the pages need to allocated frames at the same time therefore logical memory can be much bigger than physical memory.
- Set up a page table to translate logical to physical addresses.

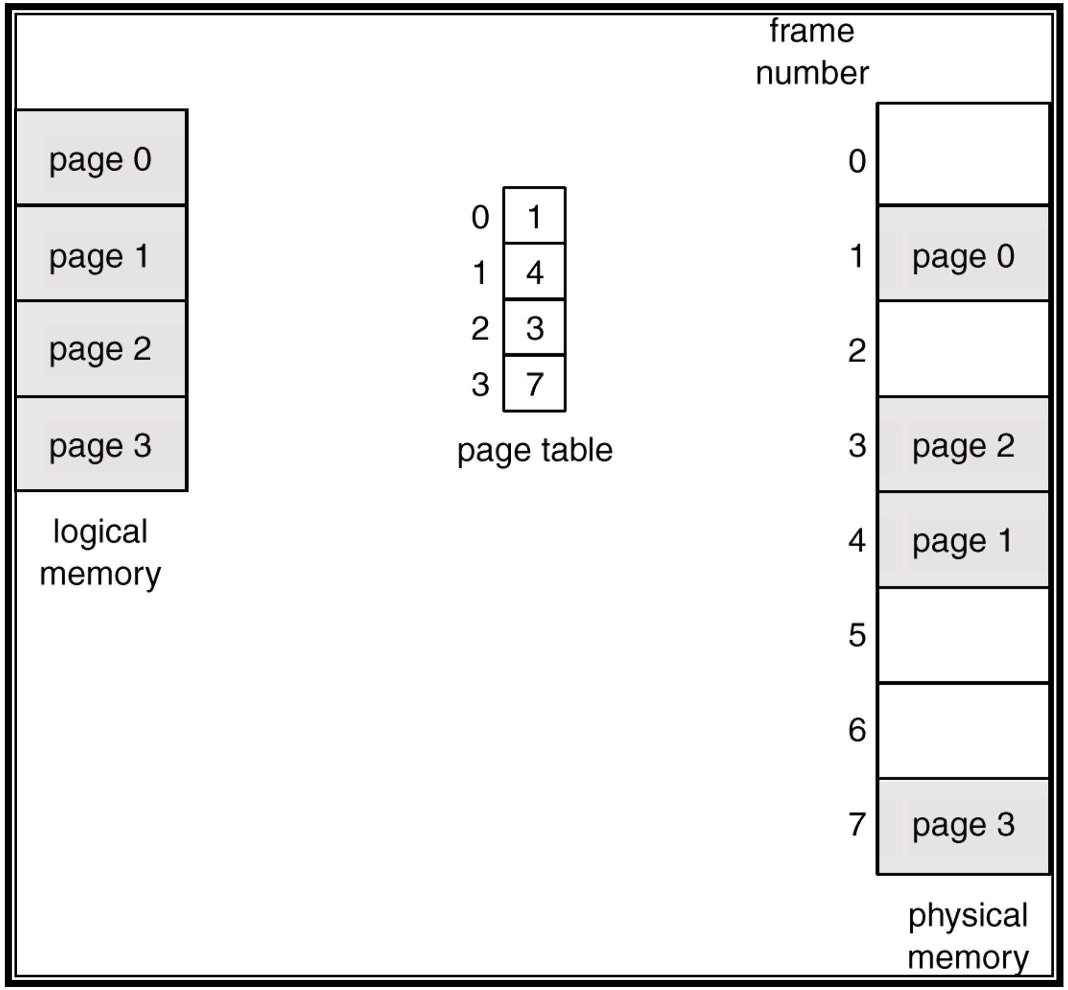
Address Translation Scheme

- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains 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.

Address Translation Architecture

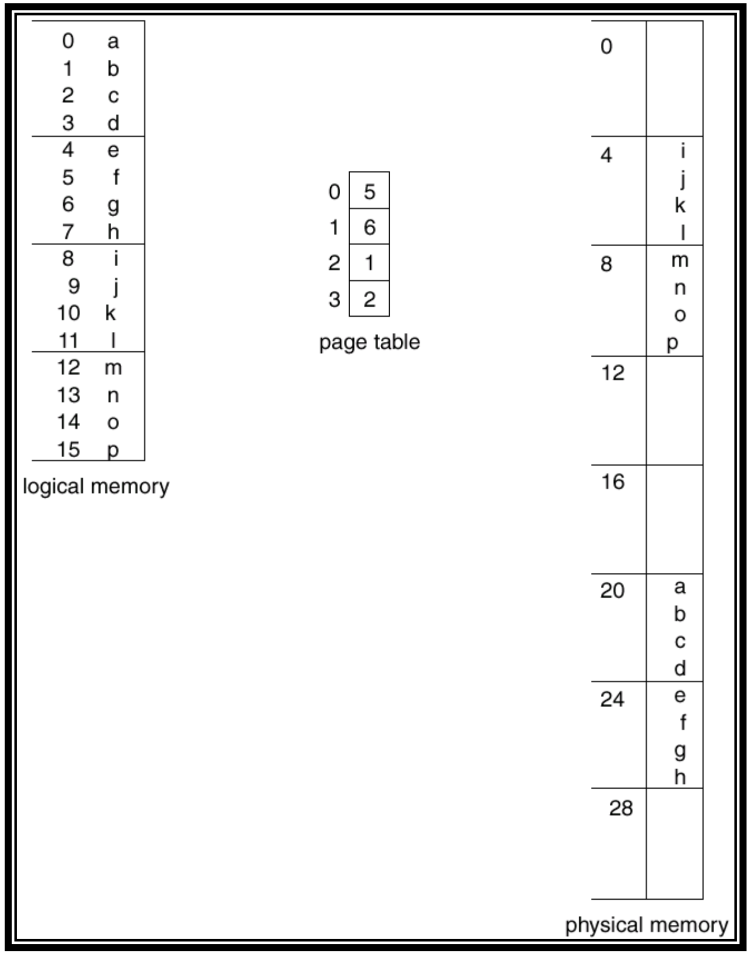


Paging Example

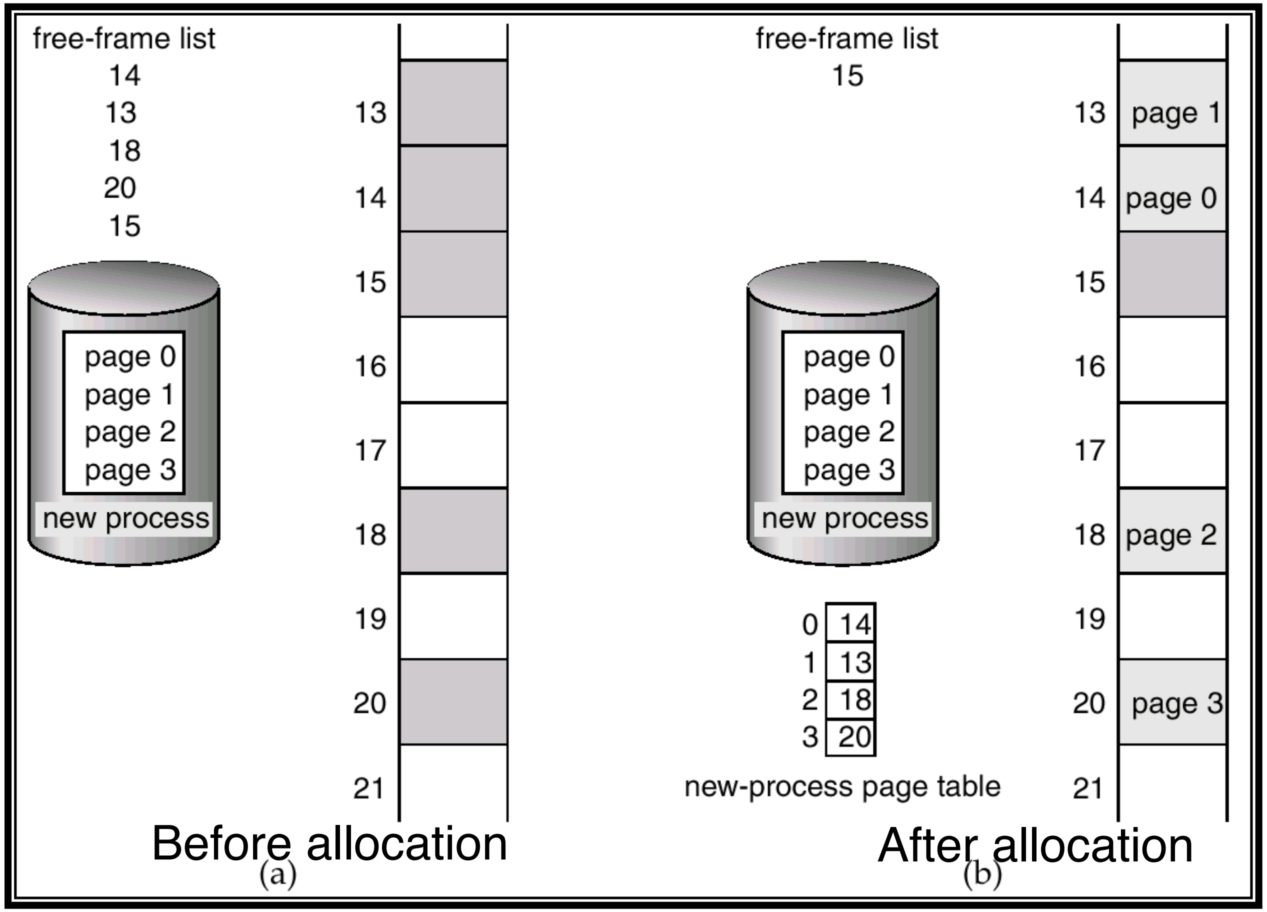


0	1
1	4
2	3
3	7

Paging Example



Free Frames



0	14
1	13
2	18
3	20

new-process page table

Before allocation

After allocation

Implementation of 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.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*

Associative Memory

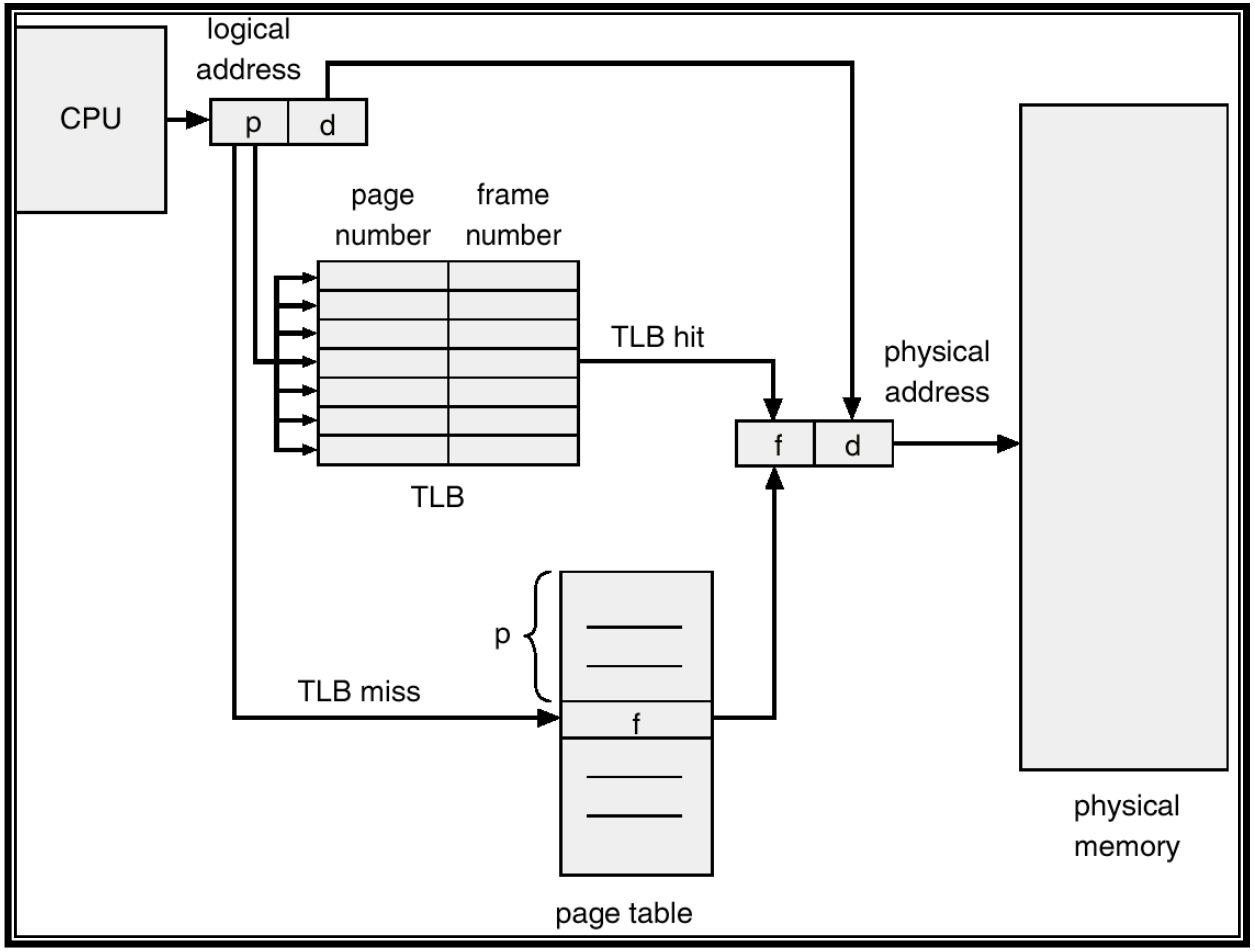
- Associative memory – parallel search

Page #	Frame #

Address translation (A´, A´´)

- If A´ is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

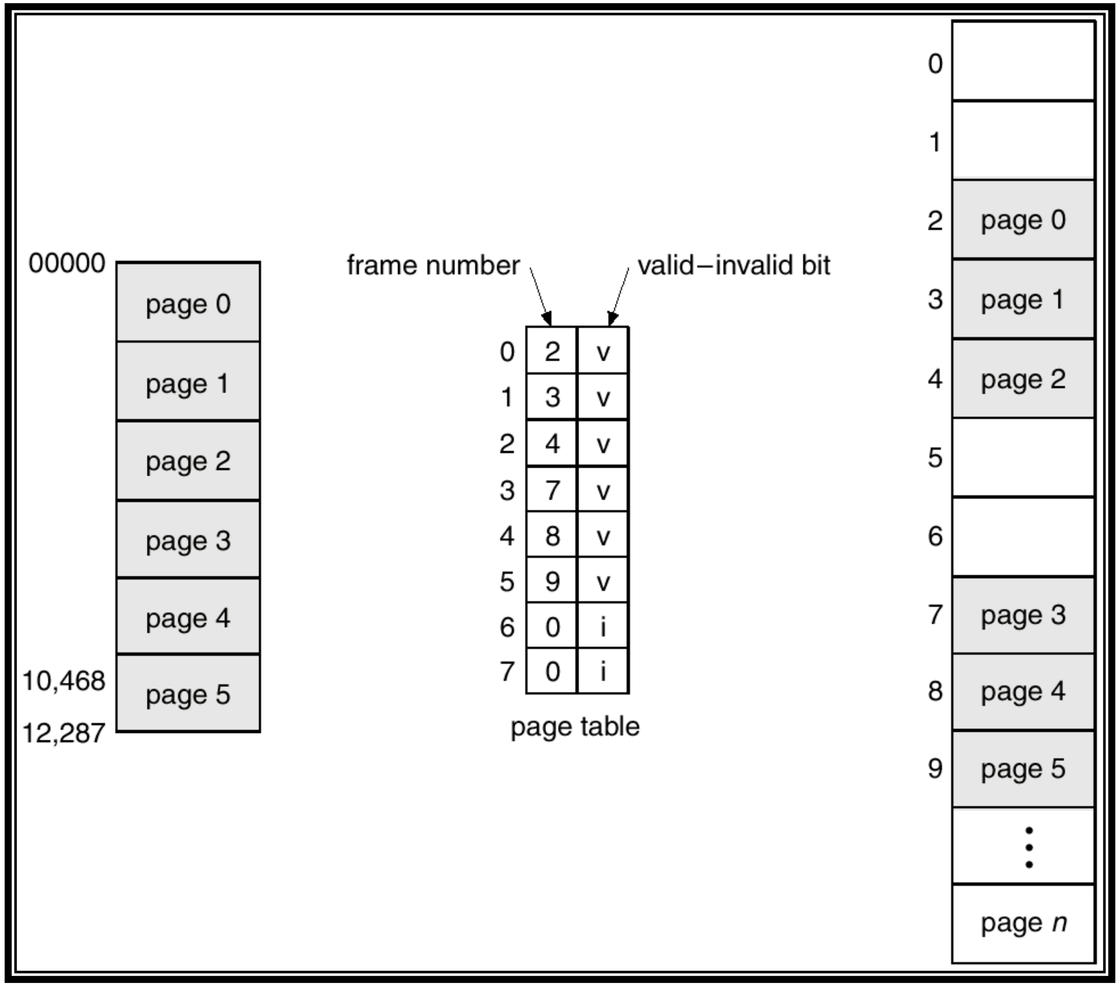
Paging Hardware With TLB



Memory Protection

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

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



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Page Table Structure

- Hierarchical Paging

- Hashed Page Tables

- Inverted Page Tables

Hierarchical Page Tables

- With 32-bit addresses and a 4KB pages we need 2^{20} page entries. If each entry is 4 bytes we will need 4MB for the page table storage per process!
- To deal with this problem we break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.
- The top level page table contains 1024 entries. If each entry is 4 bytes we need 4KB per process which is quite reasonable.

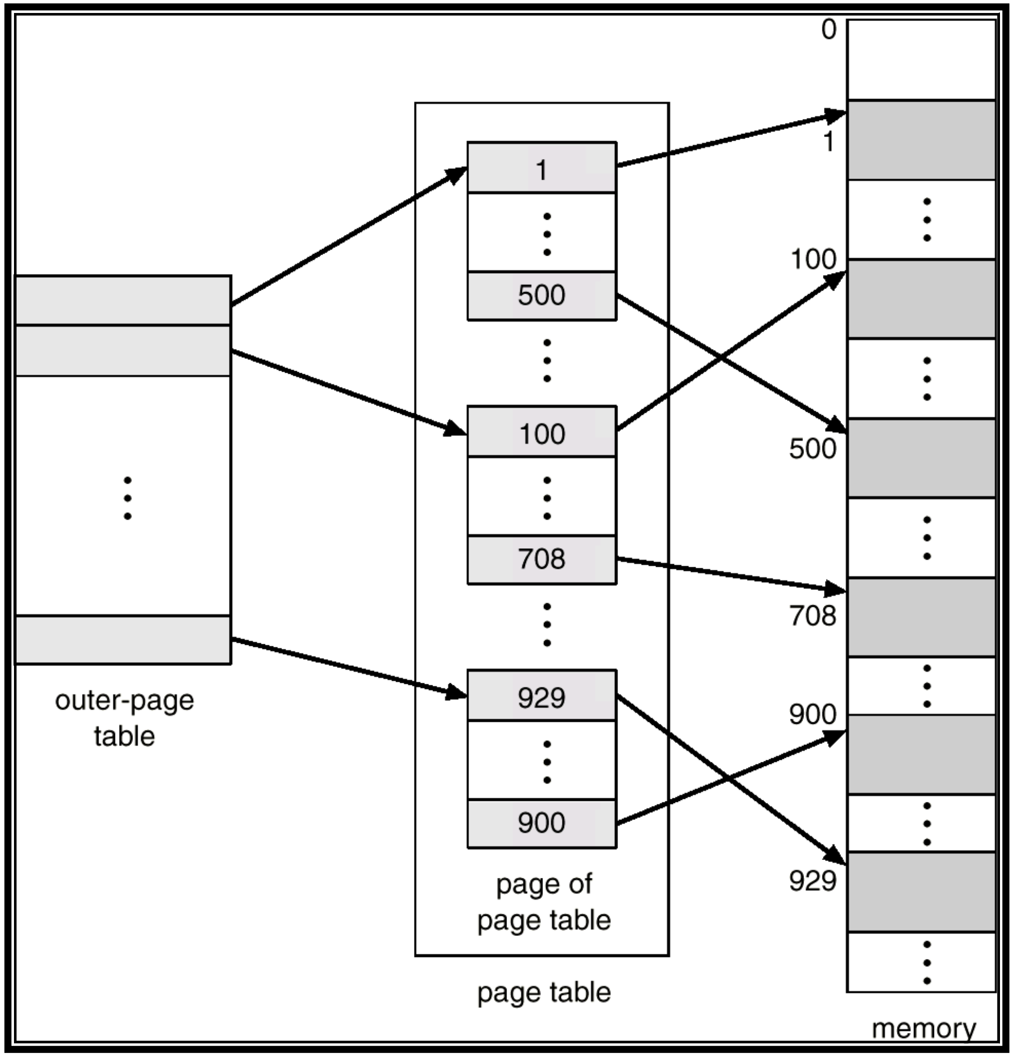
Two-Level Paging Example

- 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

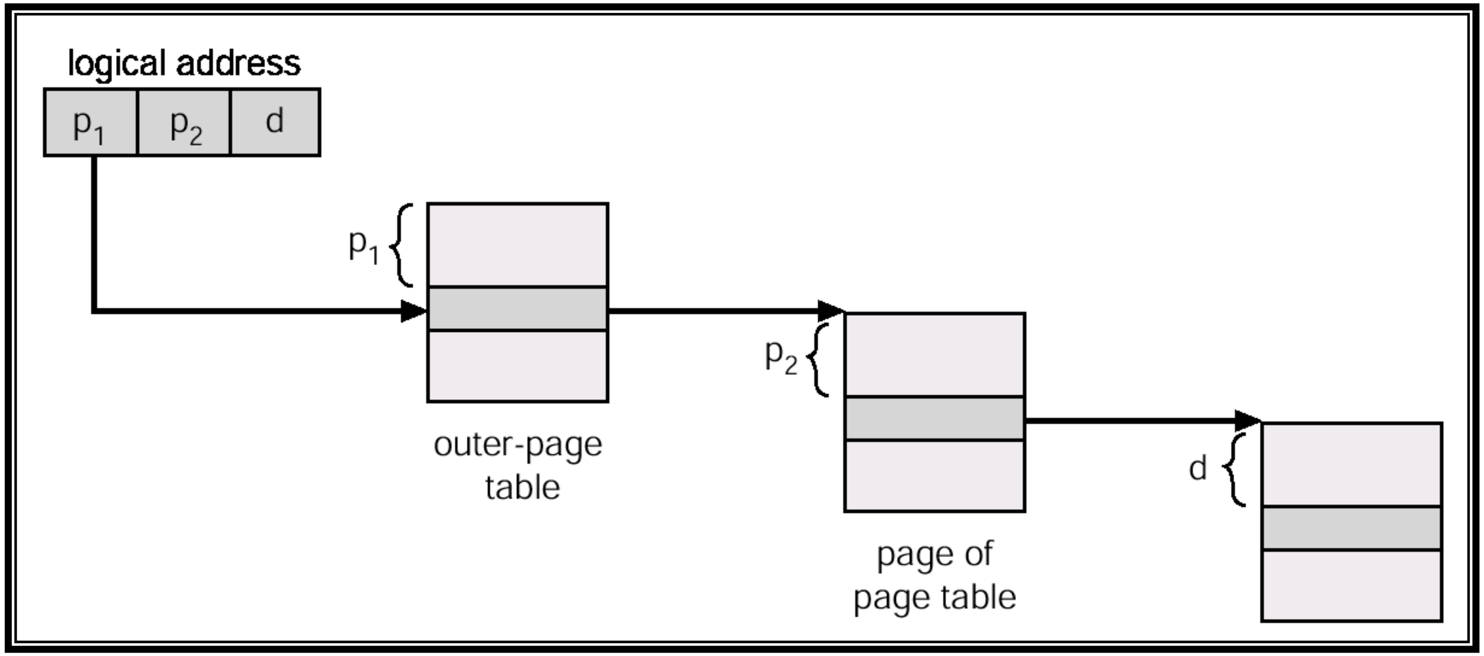
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme



Address-Translation Scheme

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



Example

- Consider a two-level page structure. A given process needs 4MB for text, 4MB for data and 4MB for stack.
- The system allocates 1024 entries in the top-level table with only three entries have the valid bit set.
- For each of the valid entries we need to allocate a second-level table with 1024 entries.
- The total storage needed would be

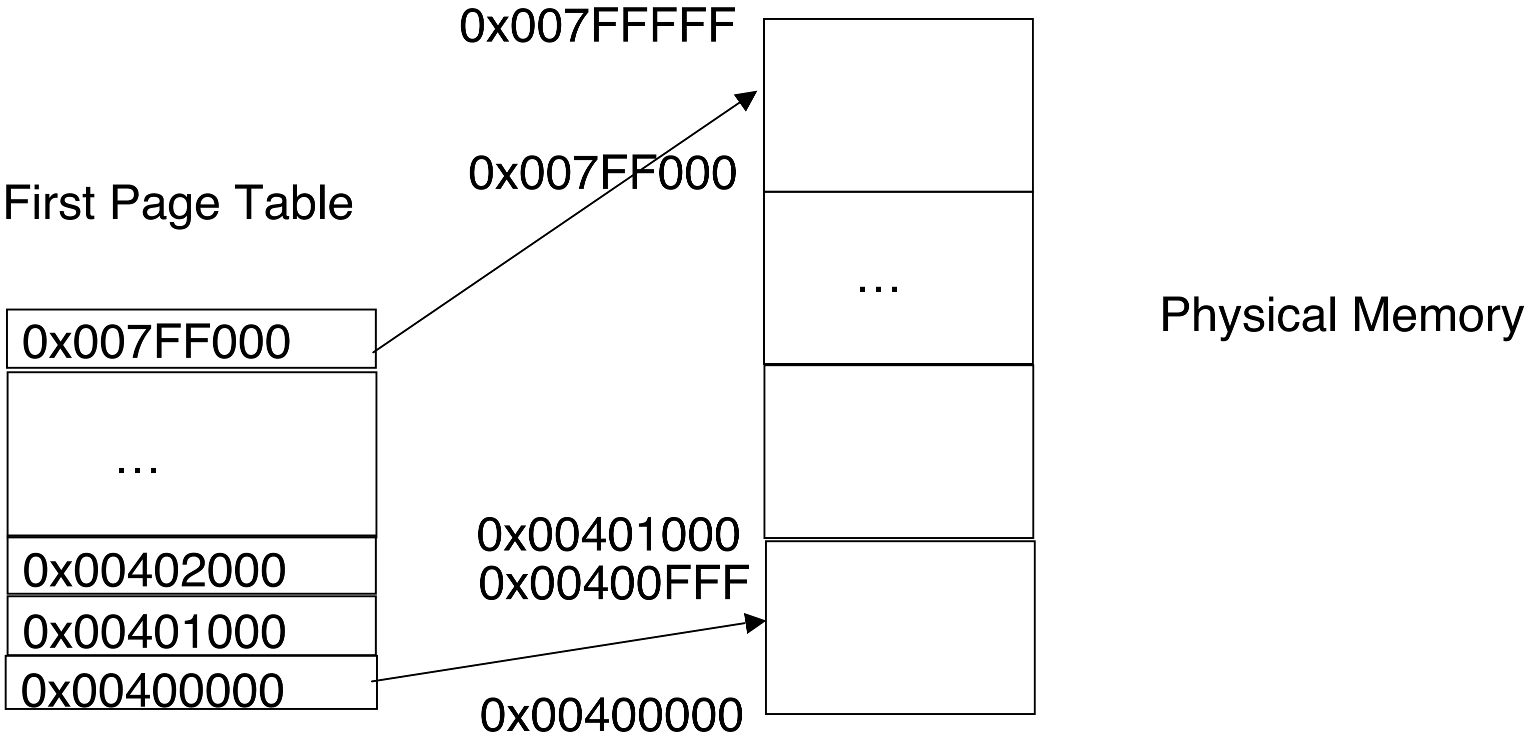
4KB for top level and 12KB for second-level=16KB

- Suppose that the OS allocates physical memory as follows
  - 4MB of text allocated between 0x00400000 and 0x00800000. (4MB and 8 MB)
  - 4MB of data allocated between 0x01400000 and 0x01800000 (20MB and 24MB)
  - 4MB of stack allocated between 0x02800000 and 0x02c00000 (40MB and 44MB)
- We need three page tables and thus three entries in the page directory.

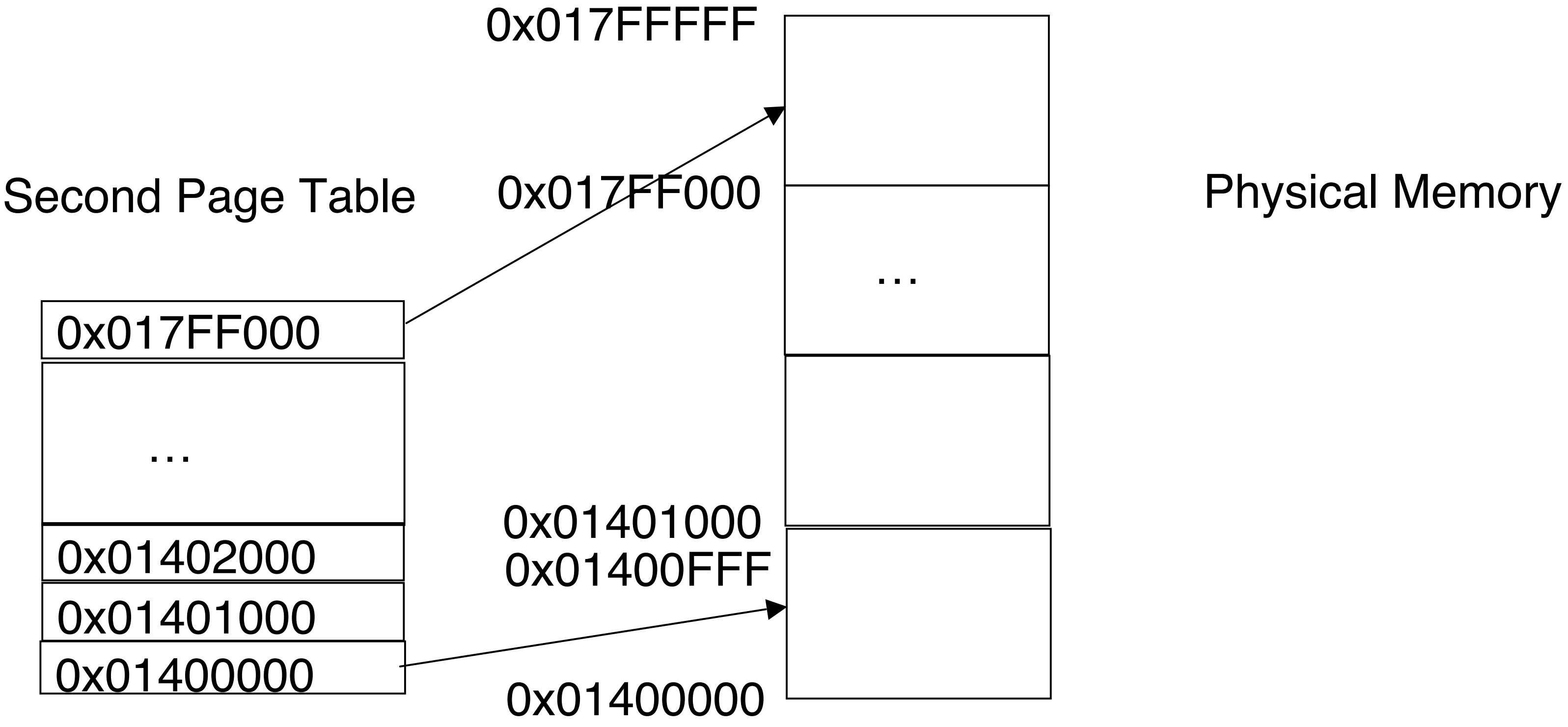
- Assume that the virtual address space for the process is 12MB starting from 0.
- Therefore the first 4MB should map to 0x00400000-0x00800000
- The second 4MB should map to 0x01400000-0x01800000
- The third 4MB should map to 0x02800000-0x02c00000

Constructing the page tables

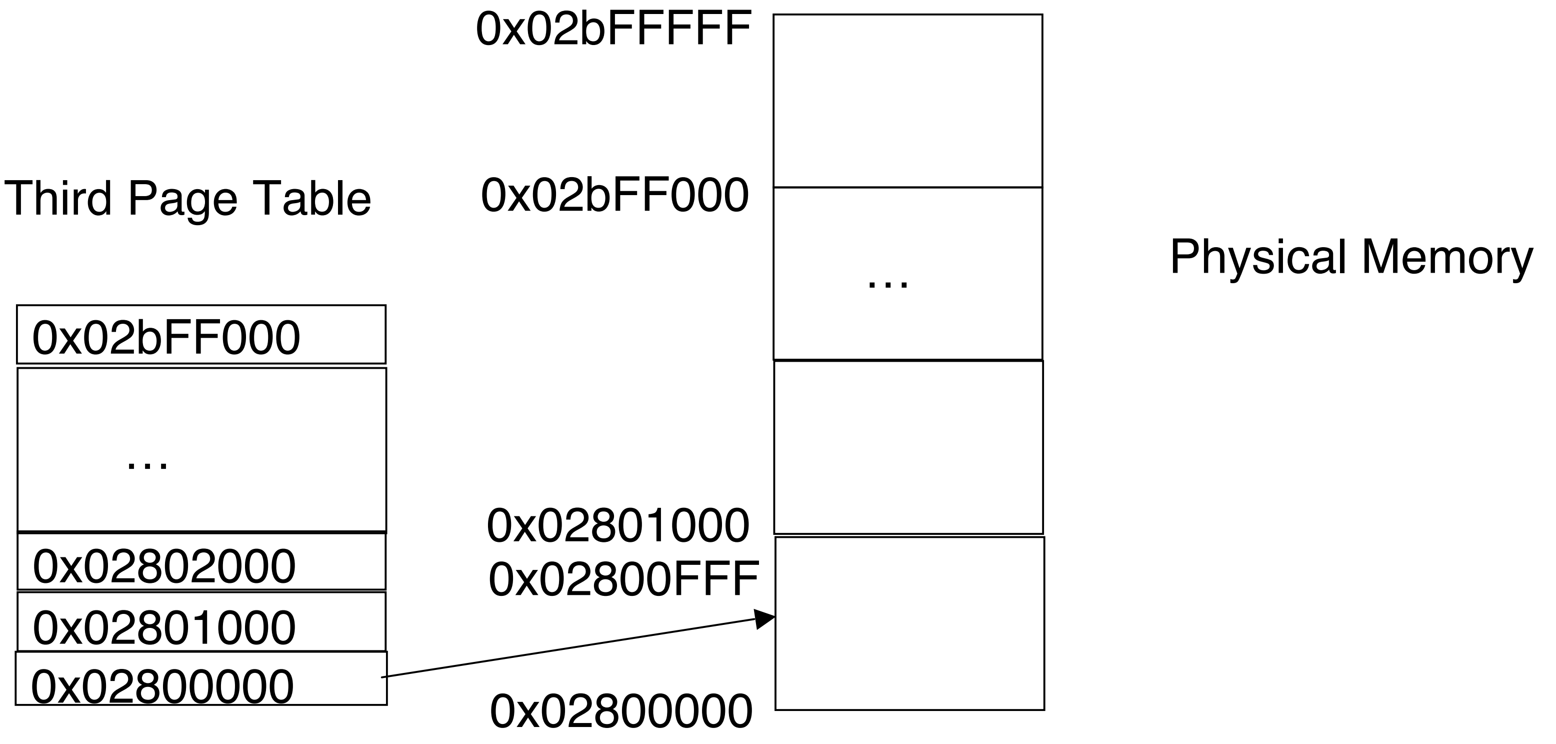
- The first page table correspond to the first mapping.



- The second page table correspond to the second mapping.



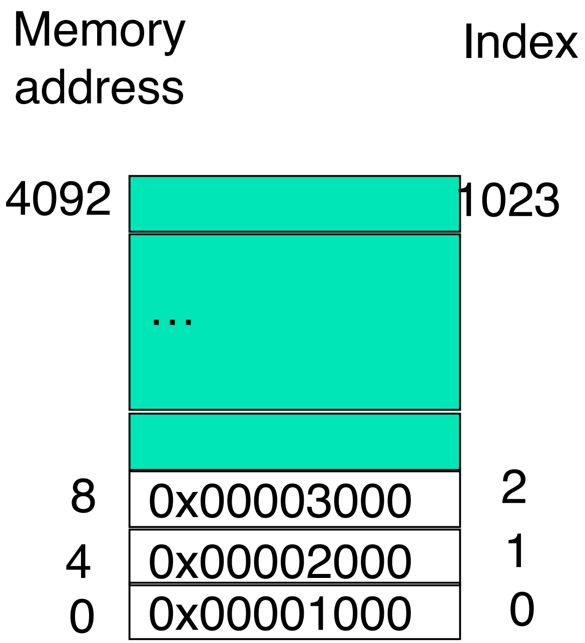
- The third page table correspond to the third mapping



Storing the tables

- Assume that the OS stores the tables starting at physical address 0.
- The first entry of the directory is at 0, the second is at 4…
- Since the directory takes 4KB, the first table starts at 4KB, the second at 8KB and the third at 12KB.
- Therefore the three entries in the directory are as follows

Page Directory



Memory address		Index
4092		1023
	…	
8	0x00003000	2
4	0x00002000	1
0	0x00001000	0

- Consider an access to virtual address a=0x00003004. (code area)
- The corresponding directory entry is a>>22=0 (the first page table)
- So the MMU retrieves the entry at 0 offset from page directory which contains 0x00001000
- The page table offset is (a>>12)&3FF=3 which is the entry offset at page table located at 0x00001000. That entry yields 0x00403000. Therefore the address is located in the page frame starting at address 0x00403000.
- The address offset=a&FFF=4 and finally the physical address is 0x00403004

- Consider the virtual address a=0x00b34567 (stack area).
- The page directory entry is a>>22=0x2. The entry whose index is 2 contains 0x00003000 which points to the third page table.
- The page table offset is (a>>12)&0x3ff=0x334=820(decimal)
- The page entry is the 820th entry in the third page table.
- In this example the pages are contiguous thus the address stored at the 820th entry is 0x02800000+0x334*0x1000=0x02b34000. This is the address of the corresponding page frame.
- Finally the physical address is 0x02b34000+0x567=0x02b34e567

Example2

- Usually the stack area is allocated towards the end of the virtual address space.
- Assume that the 4MB of stack are allocated in the range 0xbfc00000-0xc0000000
- Therefore we need the following mapping

Virtual	Physical
[0-0x00400000)	[0x00400000-0x00800000)
[0x00400000-0x00800000)	[0x01400000-0x01800000)
[0xbfc00000-0xc0000000)	[0x02800000-0x02c00000)

- Setup the directory table and all necessary page tables.
- Assume that the directory table starts at 0 physical address and the page tables are stored after the directory table.

Example 3

- Assume that the page directory is stored at 0x00100000 and all subsequent page tables are stored after it.
- Build the necessary page tables and directory entries.

Virtual	Physical
[0-0x00400000)	[0x00400000-0x00600000) And [0x01000000-0x01200000)
[0x00400000-0x00800000)	[0x01400000-0x01800000)
[0xbfc00000-0xc0000000)	[0x02800000-0x02c00000)

Shared Pages

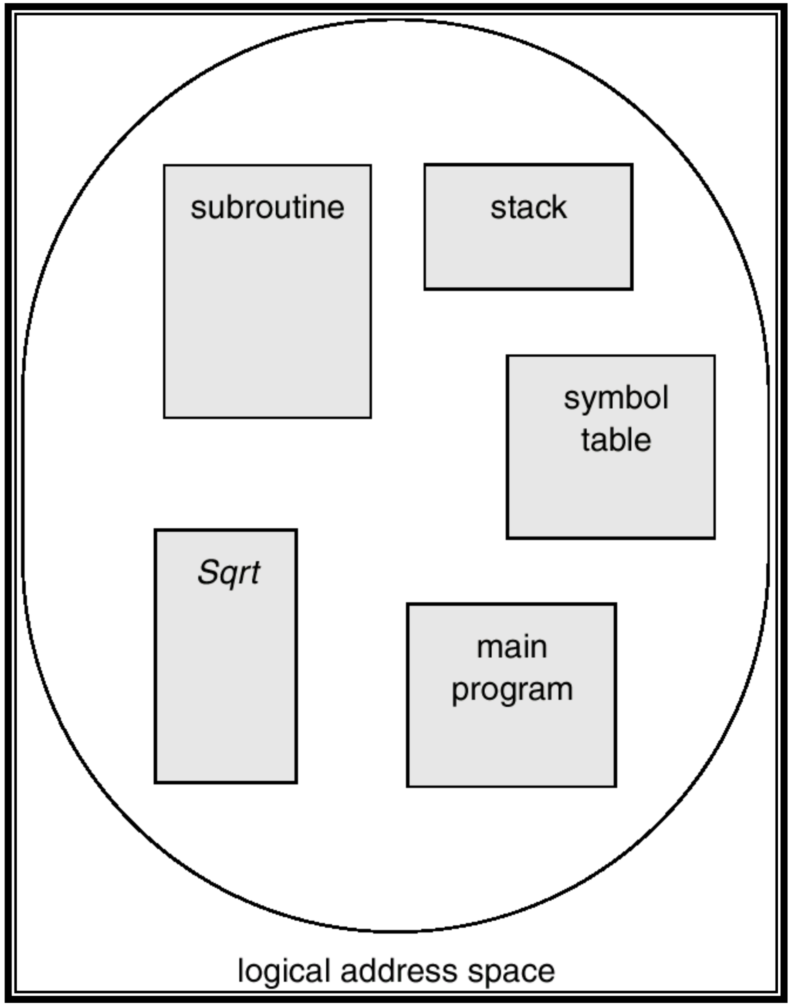
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Segmentation

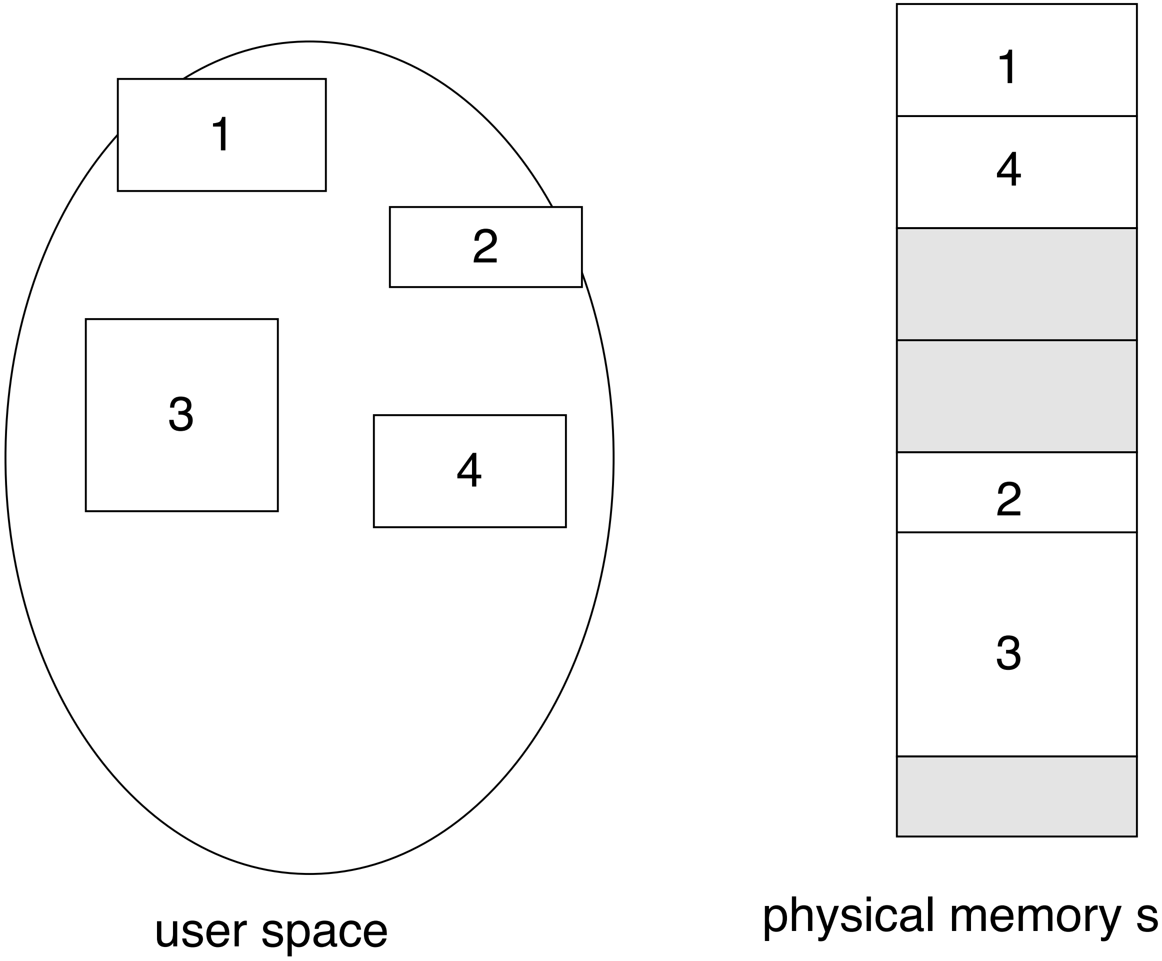
- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:

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

User's View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

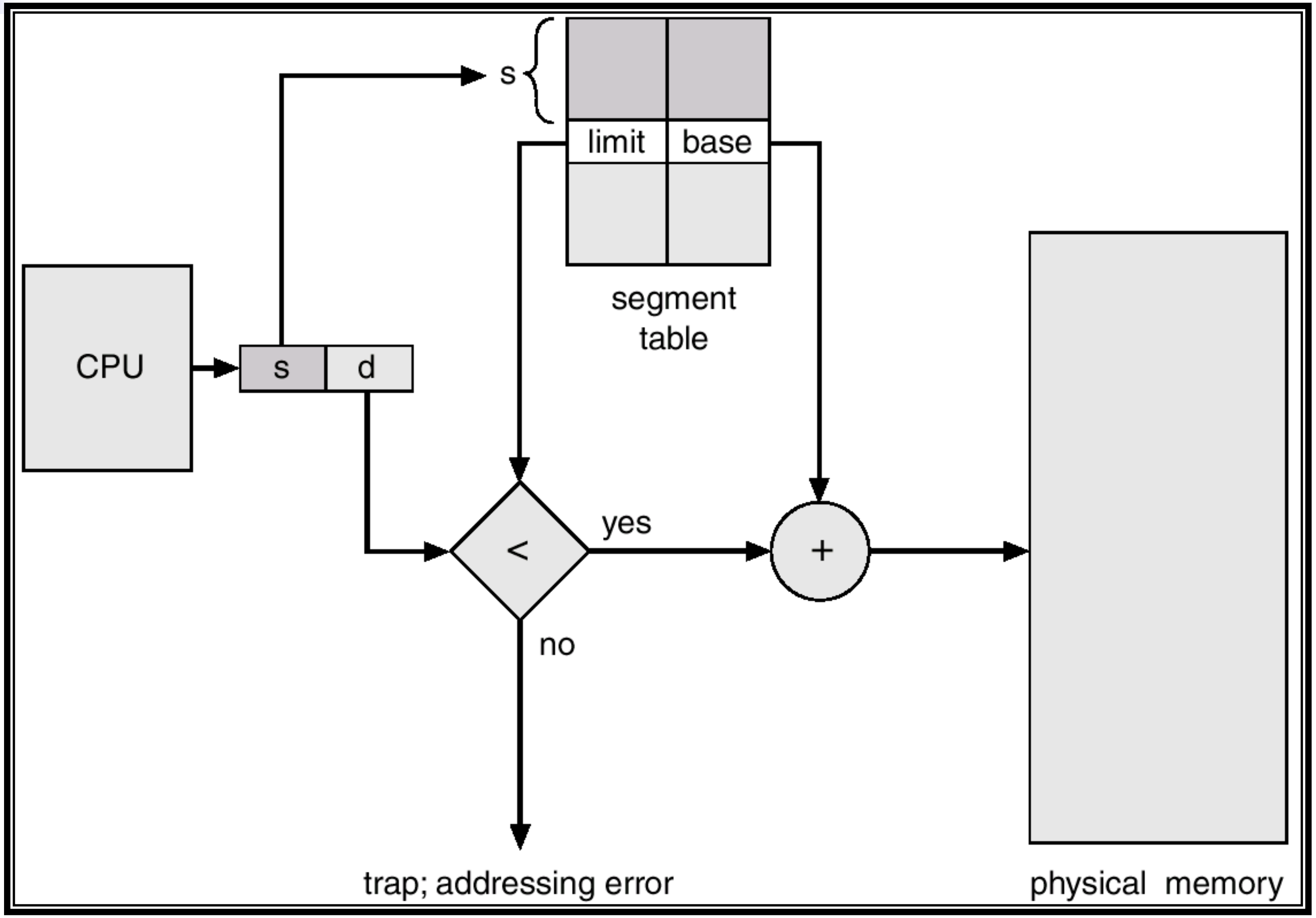
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

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

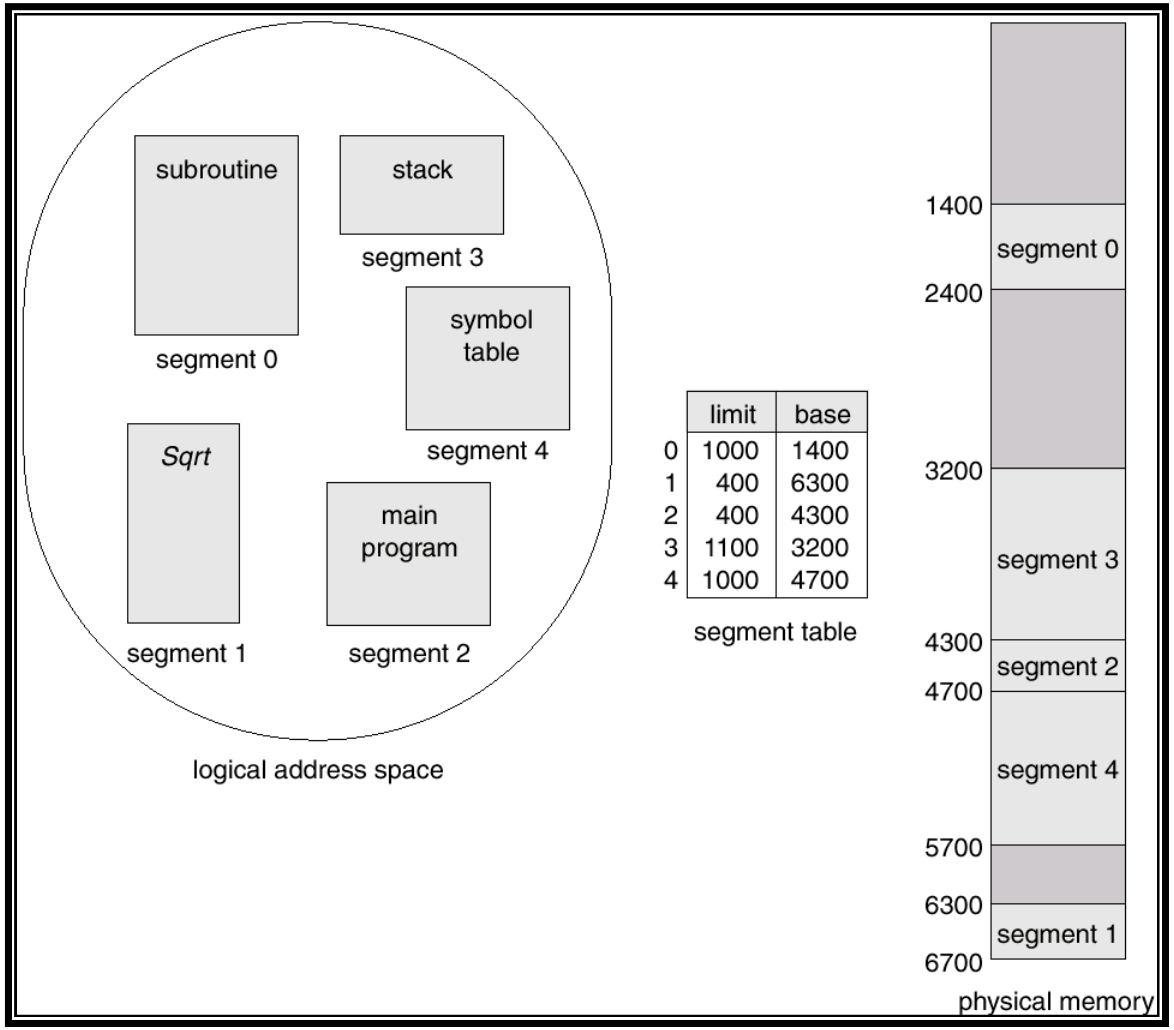
Segmentation

- 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 storage-allocation problem.
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



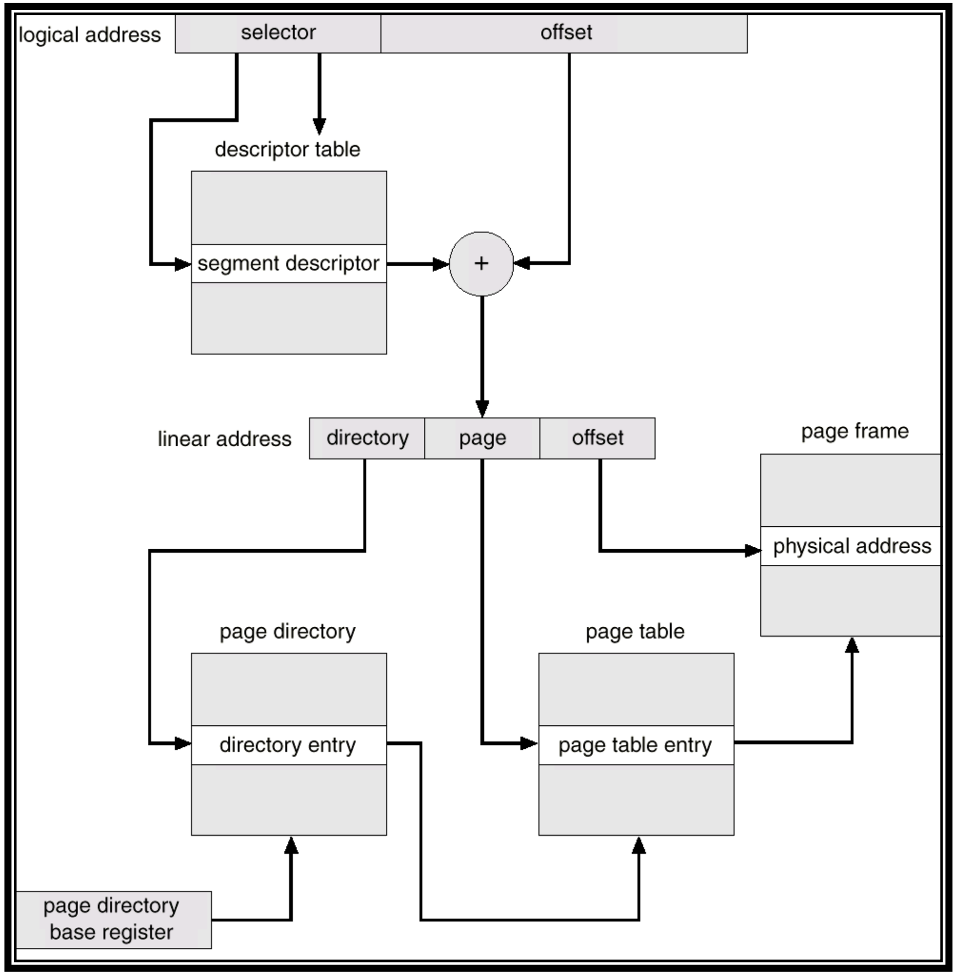
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Intel x86

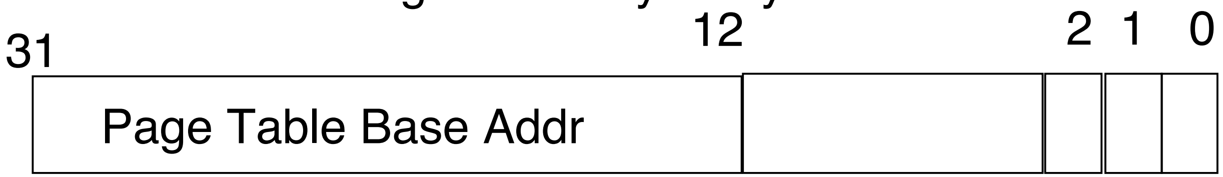
- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

Intel x86 Address Translation



X86 Page Tables

Page Directory Entry



31 12 2 1 0

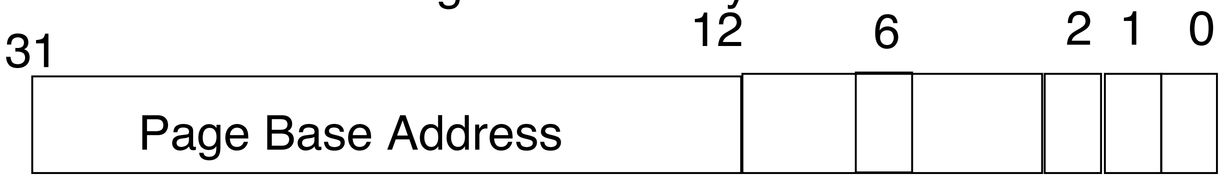
Page Table Base Addr

0-Present

1-Read/Write

2-User/Supervisor

Page Table Entry



31 12 6 2 1 0

Page Base Address

0-Present

1-Read/Write

2-User/Supervisor

6-Dirty

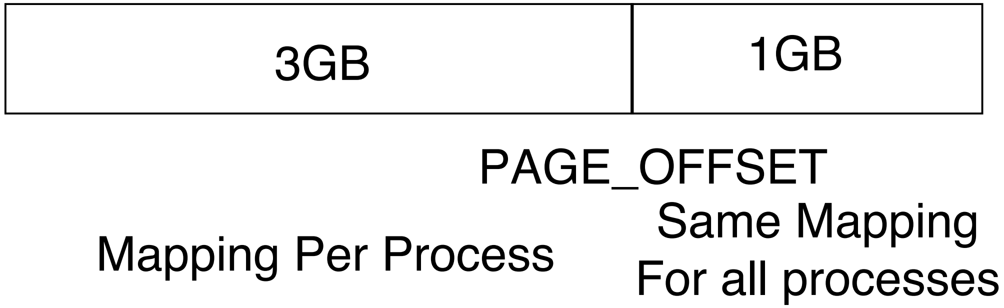
How to Fetch Table Entries

- Given a page directory entry that contains a page table base address x how do we fetch the page table entry?
- For example if x is the base address of the table and we need to fetch entry at offset 2, then the entry’s address is at x+8
- int *entry_p=x+8;

The problem is that x is a physical address and the above code does not work

Linux Memory Layout

- Linux allocates 4GB of virtual address space for each process.
- The first 3GB are for user and the fourth is for the kernel.



3GB

1GB

PAGE_OFFSET

Mapping Per Process

Same Mapping
For all processes

PAGE_OFFSET=0xc0000000

Retrieving Page Entries

We know that

1. All paging data structures are stored in kernel space.
2. Kernel space mapping is fixed

   physical address=virtual address-PAGE_OFFSET