Lecture 5: Deadlock – Banker's Algorithm Memory management

Contents

- Deadlock - Banker's Algorithm
- Memory management - history
- Segmentation
- Paging and implementation
- Page table
- Segmentation with paging
- Virtual memory concept
- Paging on demand
- Page replacement
- Algorithm LRU and it's approximation
- Process memory allocation, problem of thrashing

Banker's Algorithm

- Banker's behavior (example of one resource type with many instances):
  - Clients are asking for loans up-to an agreed limit
  - The banker knows that not all clients need their limit simultaneously
  - All clients must achieve their limits at some point of time but not necessarily simultaneously
  - After fulfilling their needs, the clients will pay-back their loans

  - Example:
    - The banker knows that all 4 clients need 22 units together, but he has only total 10 units

Client	Used	Max.
Adam	0	6
Eve	0	5
Joe	0	4
Mary	0	7

Available: 10

State (a)

Client	Used	Max.
Adam	1	6
Eve	1	5
Joe	2	4
Mary	4	7

Available: 2

State (b)

Client	Used	Max.
Adam	1	6
Eve	2	5
Joe	2	4
Mary	4	7

Available: 1

State (c)

Banker's Algorithm (cont.)

- Always keep so many resources that satisfy the needs of at least one client
- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and
m = number of resources types.

- *Available:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- *Max:* n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- *Need:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i,j] = *Max*[i,j] – *Allocation* [i,j].

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 1,2, …, *n.*

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$

   Finish[*i*] = *true*

   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i.
$Request_i[j] == k$ means that process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Test to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5instances, and C (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Need*	*Total*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	10 5 7
P_1	2 0 0	3 2 2	1 2 2	*Allocated*
P_2	3 0 2	9 0 2	6 0 0	7 2 5
P_3	2 1 1	2 2 2	0 1 1	*Available*
P_4	0 0 2	4 3 3	4 3 1	3 3 2

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria

Example (Cont.): P_1 requests (1,0,2)

- Check that Request ≤ Available
  that is, (1, 0, 2) ≤ (3, 3, 2) ⇒ true.

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	**2 3 0**
P_1	2 0 0	3 2 2	**0 2 0**	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- Executing safety algorithm shows that sequence ⟨P_1, P_3, P_4, P_0, P_2⟩ satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

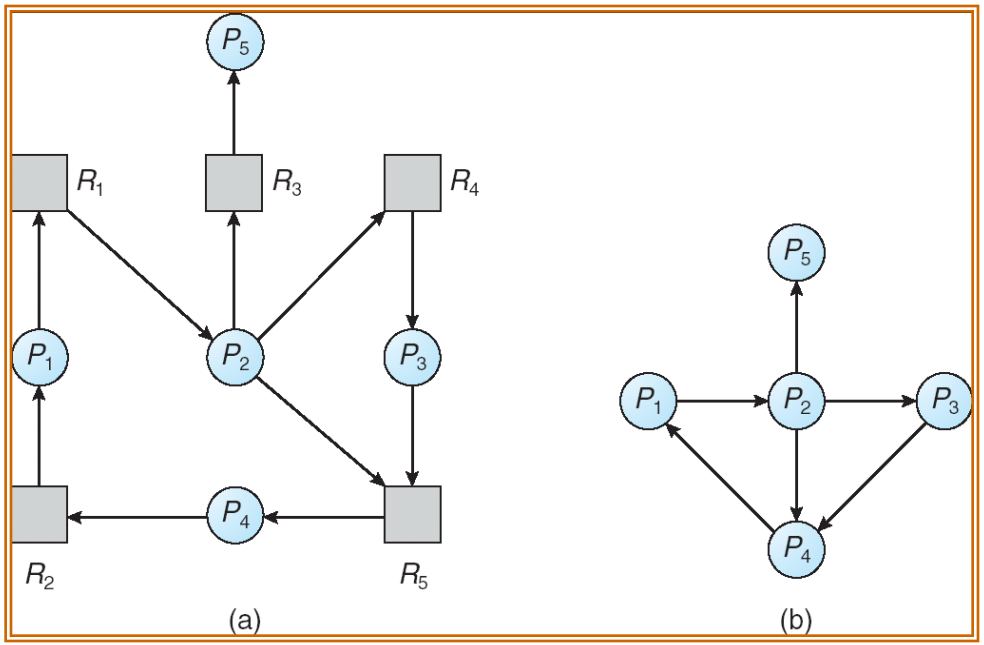
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.

- *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

- *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] == *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

The algorithm requires an order of O($m \times n^{2)}$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Total*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	7 2 6
P_1	2 0 0	2 0 2	*Allocated*
P_2	3 0 3	0 0 0	7 2 6
P_3	2 1 1	1 0 0	*Available*
P_4	0 0 2	0 0 2	**0 0 0**

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*. The *Request* matrix changes

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - System would now get deadlocked
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - Very expensive
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Memory management

Why memory?

- CPU can perform only instruction that is stored in internal memory and all it's data are stored in internal memory too
- Physical address space – physical address is address in internal computer memory
  - Size of physical address depends on CPU, on size of address bus
  - Real physical memory is often smaller then the size of the address space
    - Depends on how much money you can spend for memory.
- logical address space – generated by CPU, also referred as virtual address space. It is stored in memory, on hard disk or doesn't exist if it was not used.
  - Size of the logical address space depends on CPU but not on address bus

How to use memory

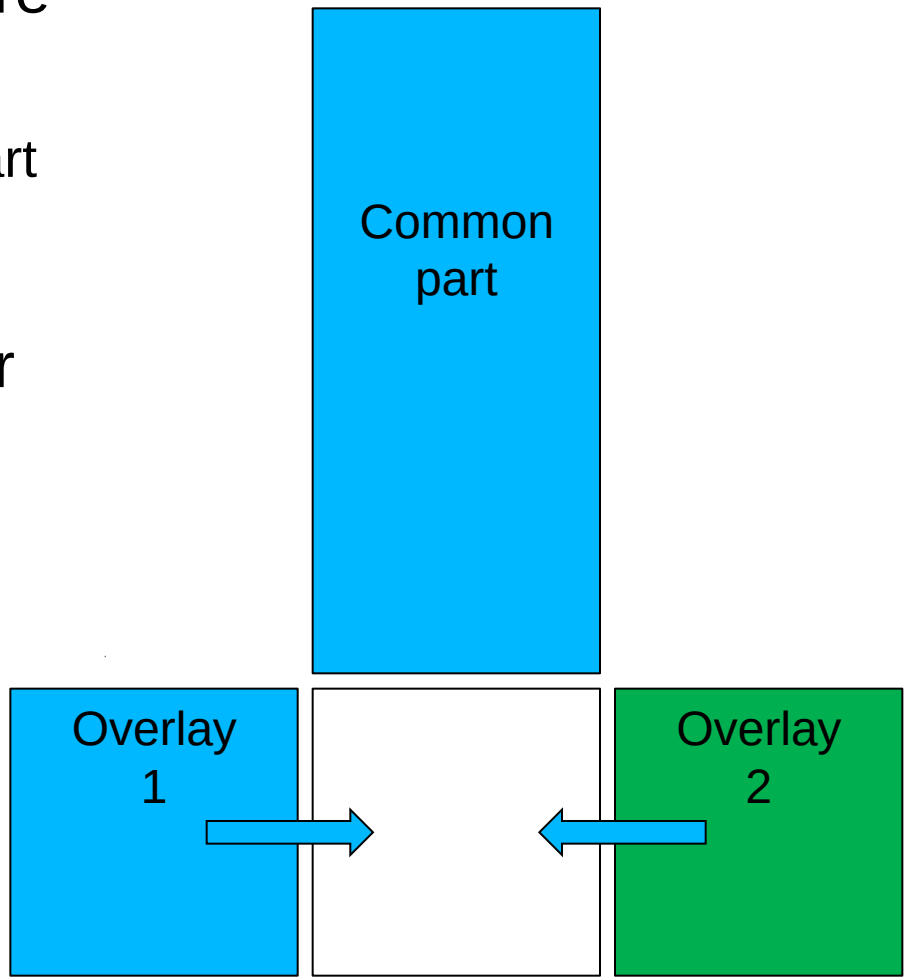
- Running program has to be places into memory
- Program is transformed to structure that can be implemented by CPU by differen steps
  - OS decides where the program will be and where the data for the program will be placed
  - Goal: **Bind address** of intructions and data to real address in address space
- Internal memory stores data and programs that are running or waiting
  - Long term memory is implemetd by secondary memory (hard drive)
- Memory management is part of OS
  - Application has no access to control memory management
    - Privilege action
  - It is not safe to enable aplication to change memory management
    - It is not effective nor safe

History of memory management

- First computer has no memory management – direct access to memory
- Advantage of system without memory management
  - Fast access to memory
  - Simple implementation
  - Can run without operating system
- Disadvantage
  - Cannot control access to memory
  - Strong connection to CPU architecture
  - Limited by CPU architecture
- Usage
  - First computer
  - 8 bits computers (CPU Intel 8080, Z80, …) - 8 bits data bus, 16 bits address bus, maximum 64 kB of memory
  - Control computers – embedded (only simple control computers)

First memory management - *Overlays*

- First solution, how to use more memory than the physical address space allows
  - Special instruction to switch part of the memory to access by address bus
- Overlays are defined by user and implemented by compiler
  - Minimal support from OS
  - It is not simple to divid data or program to overlays



Virtual memory

- Demand for bigger protected memory that is managed by somebody else (OS)
- Solution is virtual memory that is somehow mapped into real physical memory
- 1959-1962 first computer Atlas Computer from Manchesteru with virtual memory (size of the memory was 576 kB) implemented by paging
- 1961 - Burroughs creates computer B5000 that uses segment for virtual memory
- Intel
  - 1978 processor 8086 – first PC – simple segments
  - 1982 processor 80286 – protected mode – real segmentation
  - 1985 processor 80386 – full virtual memory with segmentation and paging

Simple segemnts – Intel 8086

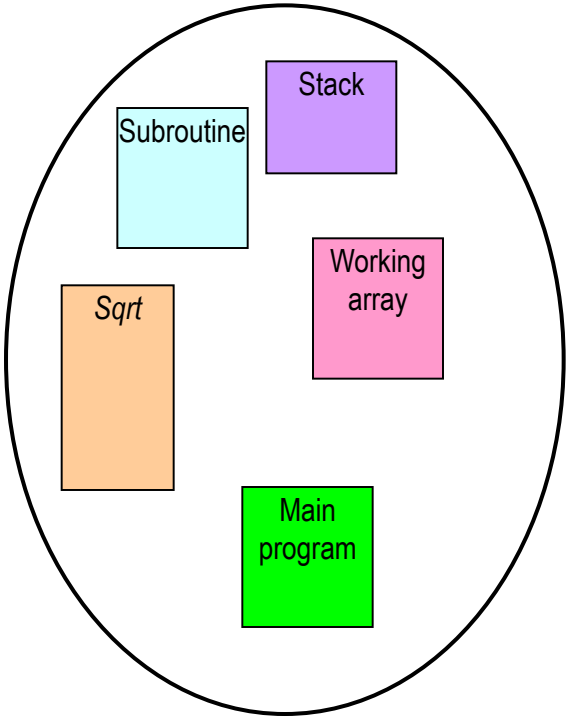
- Processor 8086 has 16 bits of data bus and 20 bits of address bus. 20 bits is problem. How to get 20 bits numbers?
- Solution is “simple” segments
- Address is composed with 16 bits address of segment and 16-bits address of offset inside of the segment.
- Physical address is computed as:

  (segment<<4)+offset

- It is not real virtual memory, only system how to use bigger memory
- Two types of address
  - near pointer – contains only address inside of the segment, segment is defined by CPU register
  - far pointer – pointer between segments, contains segment description and offset

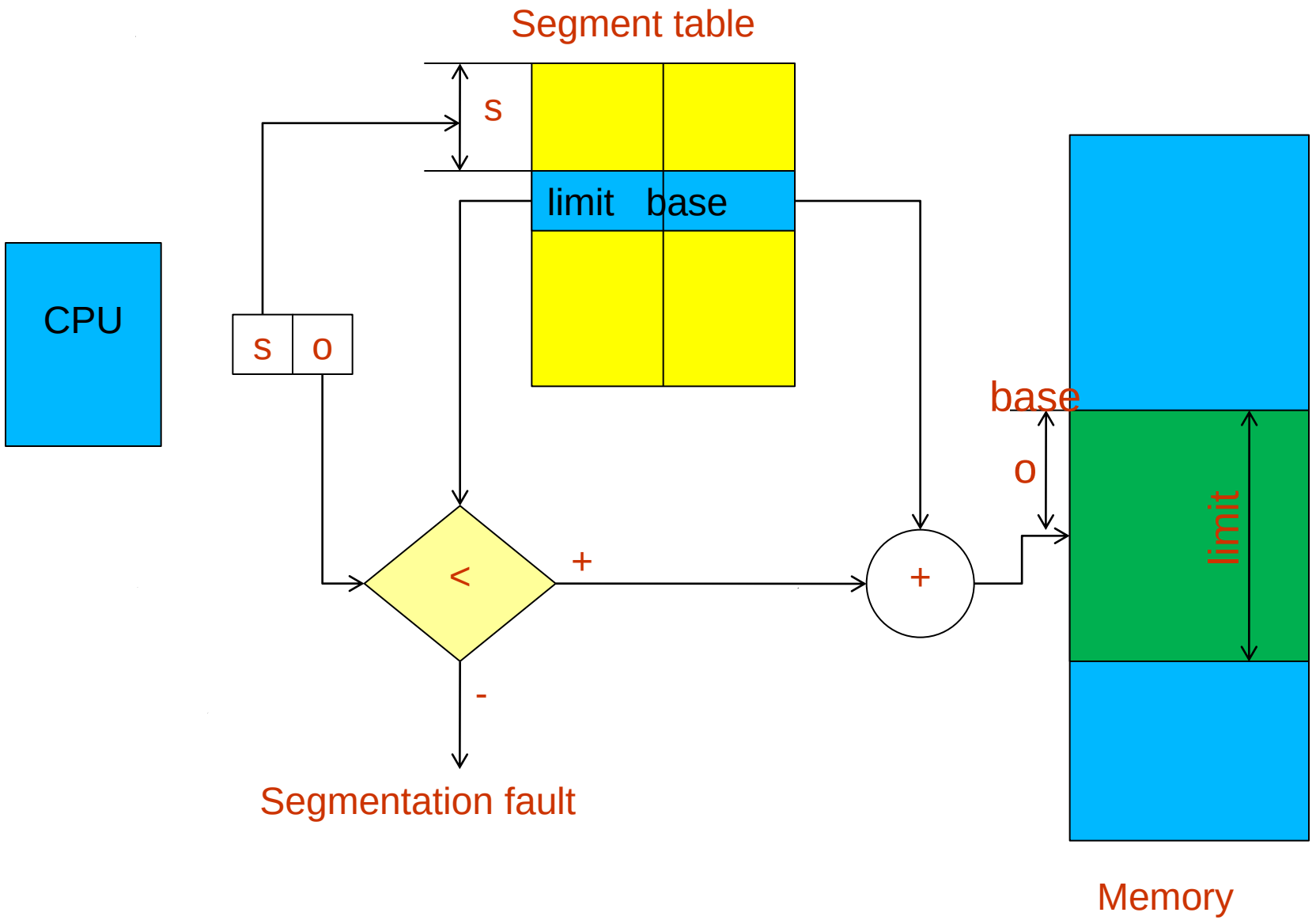
Segmentation – protected mode Intel 80286

- Support for user definition of logical address space
  - Program is set of segments
  - Each segment has it's own meaning: main program, function, data, library, variable, array, ...
- Basic goal – how to transform address (segment, offset) to physical address



- Segment table – ST
  - Function from 2-D (segment, offset) into 1-D (address)
  - One item in segment table:
    - base – location of segment in physical memory, limit – length of segment
  - Segment-table base register (STBR) – where is ST in memory
  - Segment-table length register (STLR) – ST size

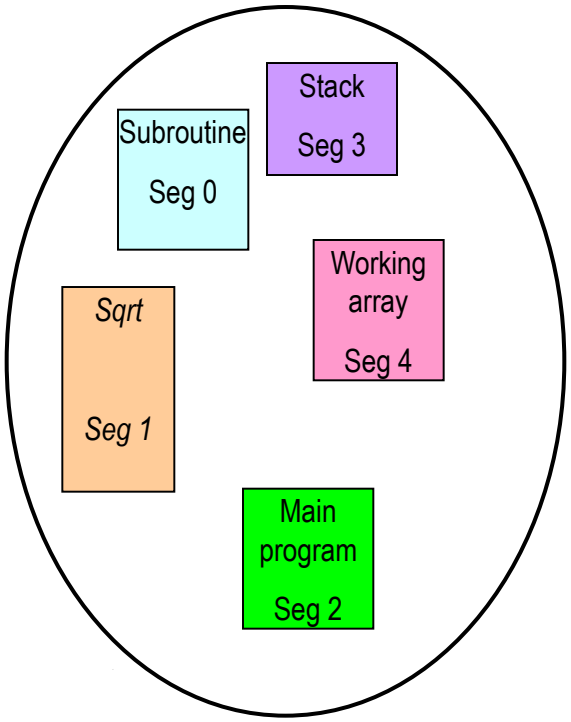
Hardware support for segmentation



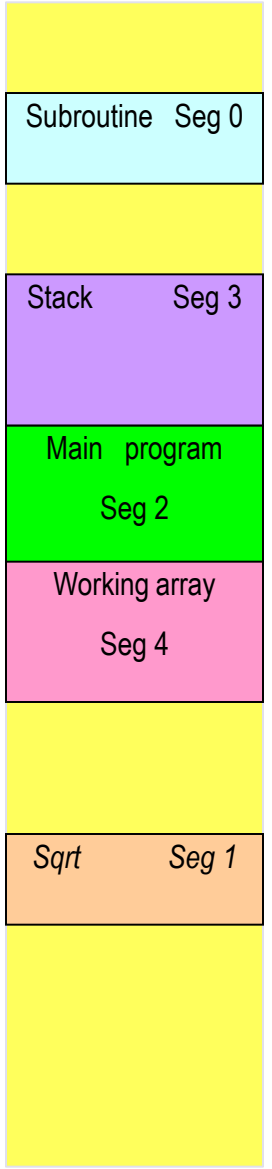
Segmentation

- Advantage of the segmentation
  - Segment has defined length
  - It is possible to detect access outside of the segment. It throws new type of error – segmentation fault
  - It is possible to set access for segment
    - OS has more privilege than user
    - User cannot affect OS
  - It is possible to move data in memory and user cannot detect this shift (change of the segment base is for user invisible)
- Disadvantage of segmentation
  - How to place segments into main memory. Segments have different length. Programs are move into memory and release memory.
  - Overhead to compute physical address from virtual address (one comparison, one addition)

Segmentation example



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



- It is not easy to place the segment into memory
  - Segments has different size
  - Memory fragmentation
  - Segemnt moving has big overhead (is not used)

Paging

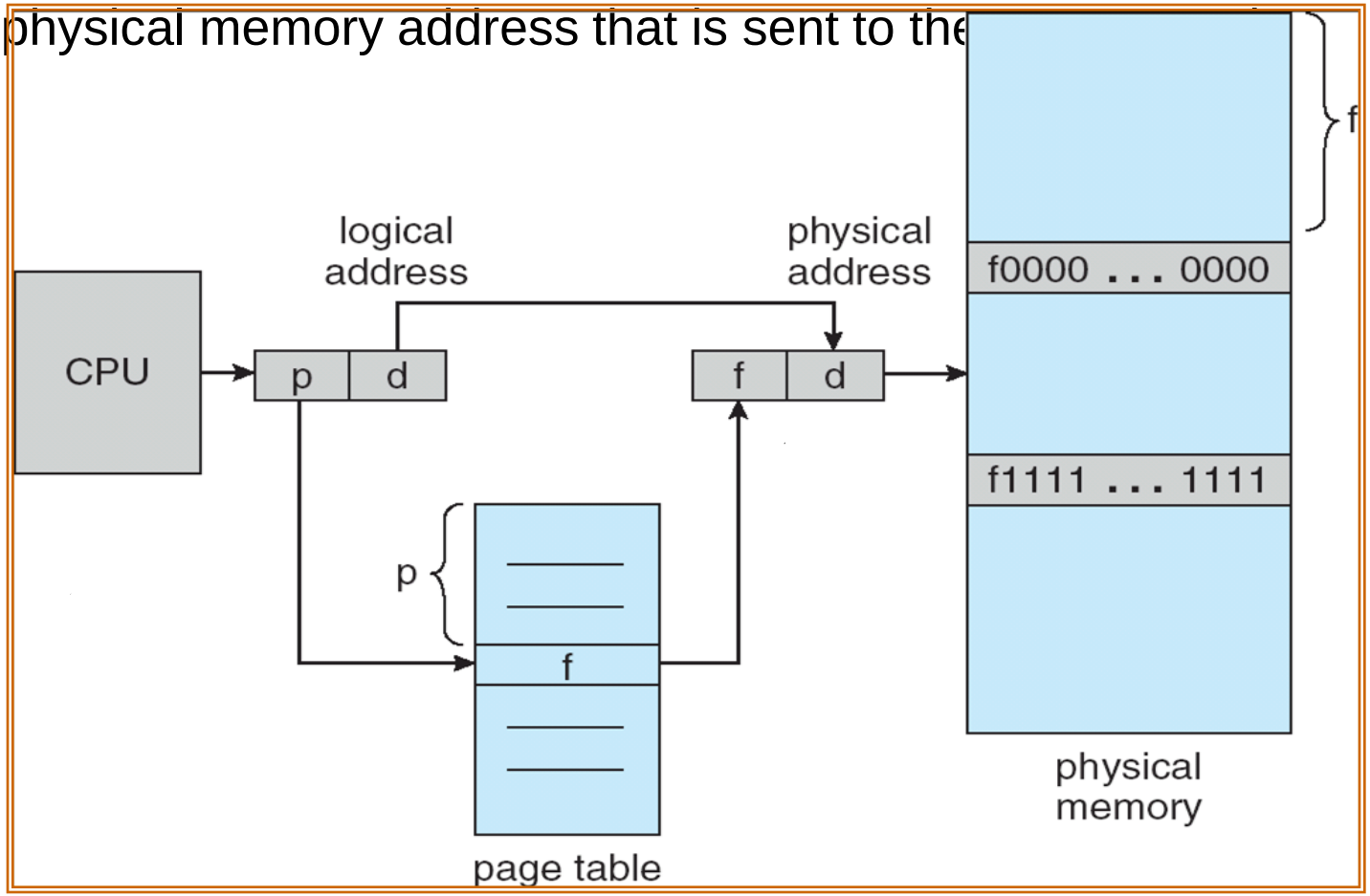
- Different solution for virtual memory implementation
- Paging remove the basic problem of segments – different size
- All pages has the same size that is defined by CPU architecture
- Fragmentation is only inside of the page (small overhead)

Paging

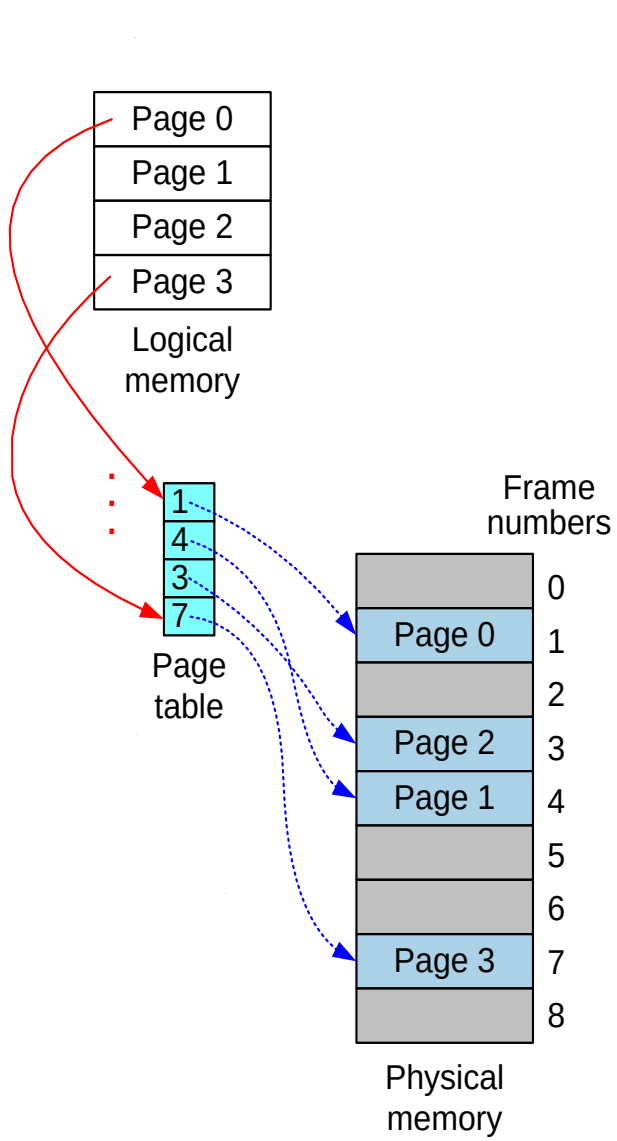
- Contiguous logical address space can be mapped to noncontiguous physical location
  - Each page has its own position in physical memory
- Divide physical memory into fixed-sized blocks called frames
  - The size is power of 2 between 512 and 8 192 B
- Dived logical memory into blocks with the same size as frames. These blocks are called pages
- OS keep track of all frames
- To run process of size *n* pages need to find *n* free frames, Tranformation from logical address → physical address by
  - PT = *Page Table*

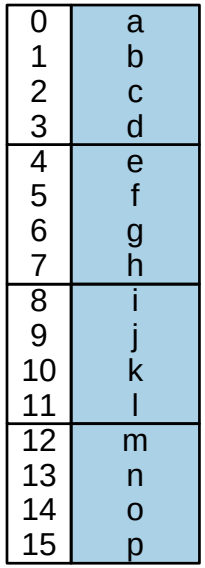
Address Translation Scheme

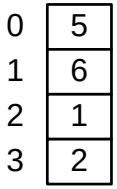
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the

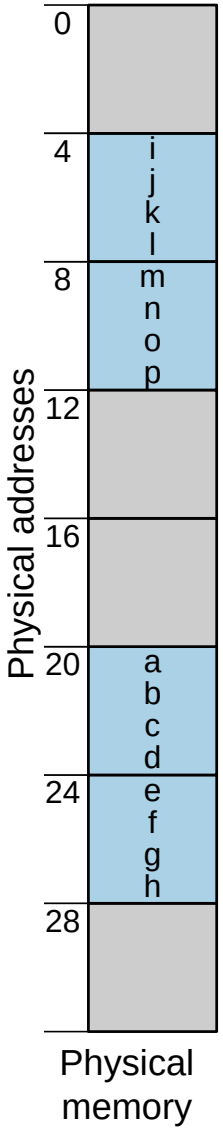


Paging Examples









Implementation of Page Table

- Paging is implemented in hardware
- **Page table is kept in main memory**
- ***Page-table base register*** *(*PTBR) points to the page table
- ***Page-table length register*** (PTLR) 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

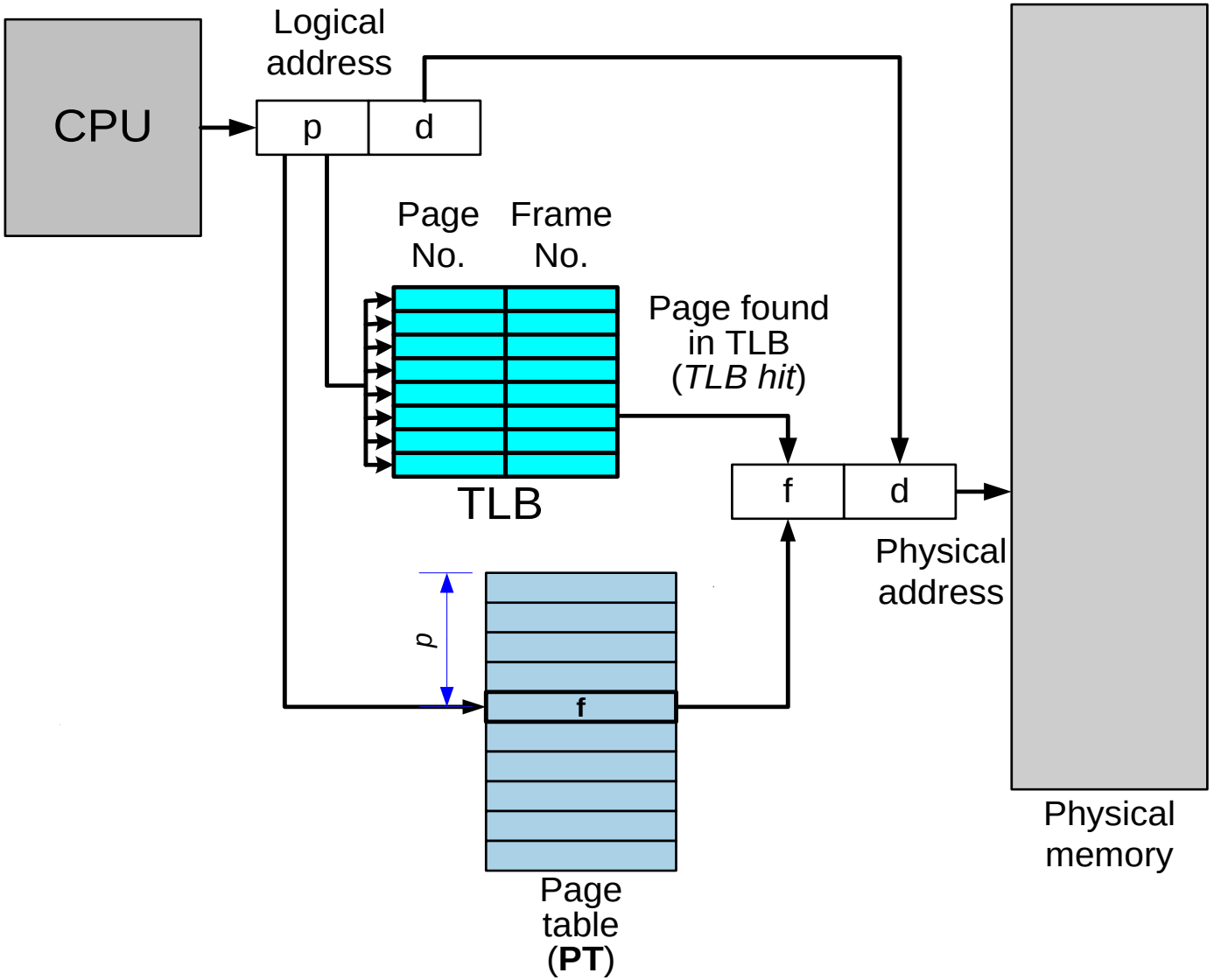
- Associative memory – parallel search – content-addressable memory
- Very fast search

TBL

Page # Input address	Frame # Output address
100000	ABC000
100001	201000
300123	ABC001
100002	300300

- Address translation (A´, A´´)
  - If A´ is in associative register, get Frame
  - Otherwise the TBL has no effect, CPU need to look into page table
- Small TBL can make big improvement
  - Usually program need only small number of pages in limited time

Paging Hardware With TLB



Paging Properties

- Effective Access Time with TLB
  - Associative Lookup = ε time unit
  - Assume memory cycle time is t = 100 nanosecond
  - Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers, Hit ratio = α
  - **Effective Access Time** (EAT)

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

Example for *t* = 100 *ns*

PT without *TLB*		*EAT* = 200 *ns*	Need two access to memory
ε = 20 *ns*	α = 60 %	*EAT* = 160 *ns*	*TLB* increase significantly access time
ε = 20 *ns*	α = 80 %	*EAT* = 140 *ns*	
ε = 20 *ns*	α = 98 %	*EAT* = 122 *ns*	

TLB

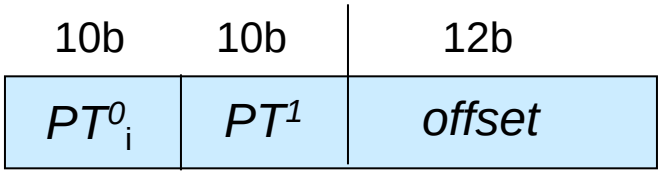
- Typical TLB
  - Size 8-4096 entries
  - Hit time 0.5-1 clock cycle
  - PT access time 10-100 clock cycles
  - Hit ration 99%-99.99%
- Problem with context switch
  - Another process needs another pages
  - With context switch invalidates TBL entries (free TLB)
- OS takes care about TLB
  - Remove old entries
  - Add new entries

Page table structure

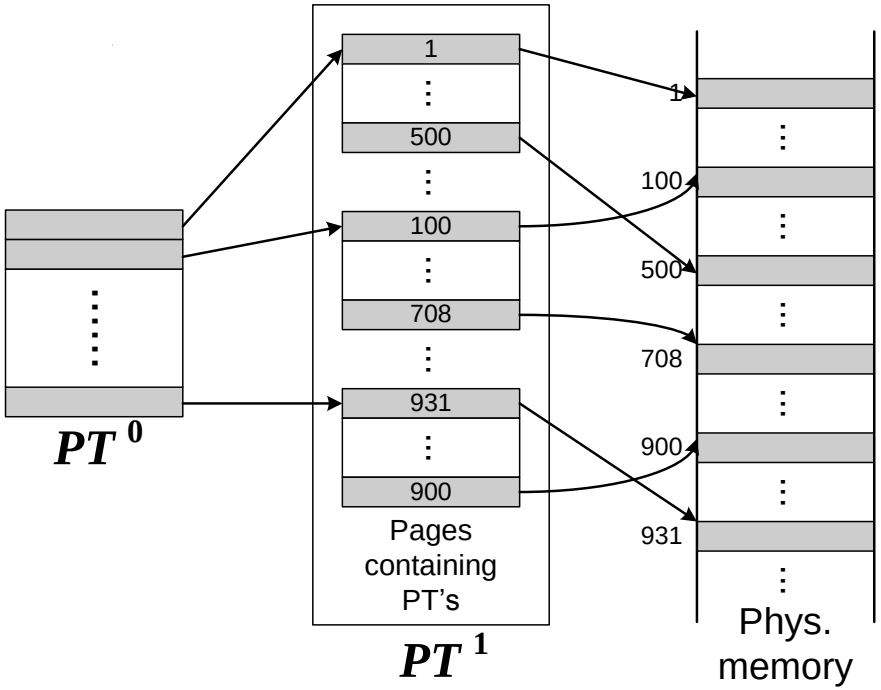
- Problem with PT size
  - Each process can have it's own PT
  - 32-bits logical address with page size 4 KB → PT has 4 MB
    - PT must be in memory
- Hierarchical PT
  - Translation is used by PT hierarchy
  - Usually 32-bits logical address has 2 level PT
  - ***PT*** ^{0} contains reference to ***PT*** ^{1}
  - ,Real page table ***PT*** ^{1} can be paged need not to be in memory
- Hash PT
  - Address ***p*** is used by hash function *hash*(*p*)
- Inverted PT
  - One PT for all process
  - Items depend on physical memory size
  - Hash function has address p and process pid *hash*(*pid, p*)

Hierarchical Page Tables

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

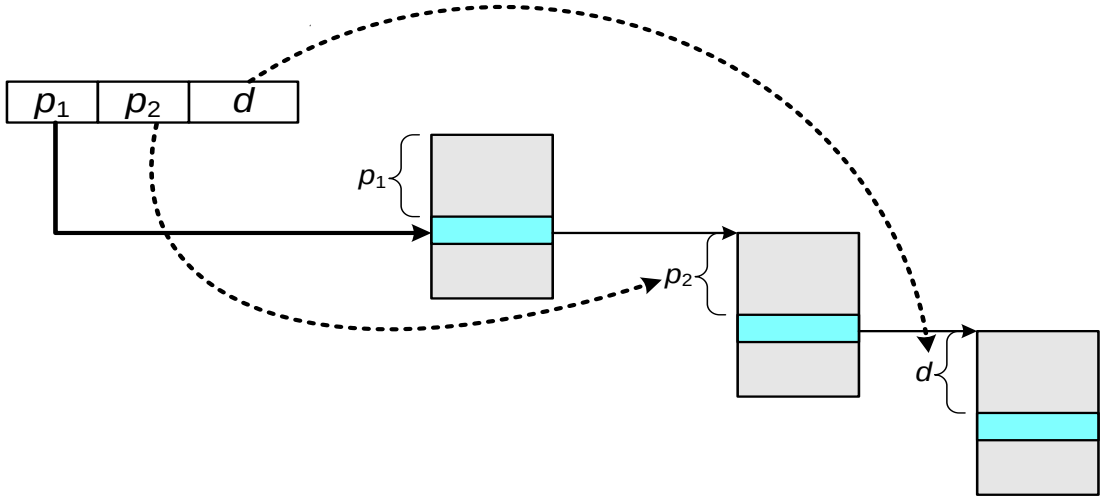


Two-Level Page-Table Scheme



Two-level paging structure

Logical → physical address translation scheme

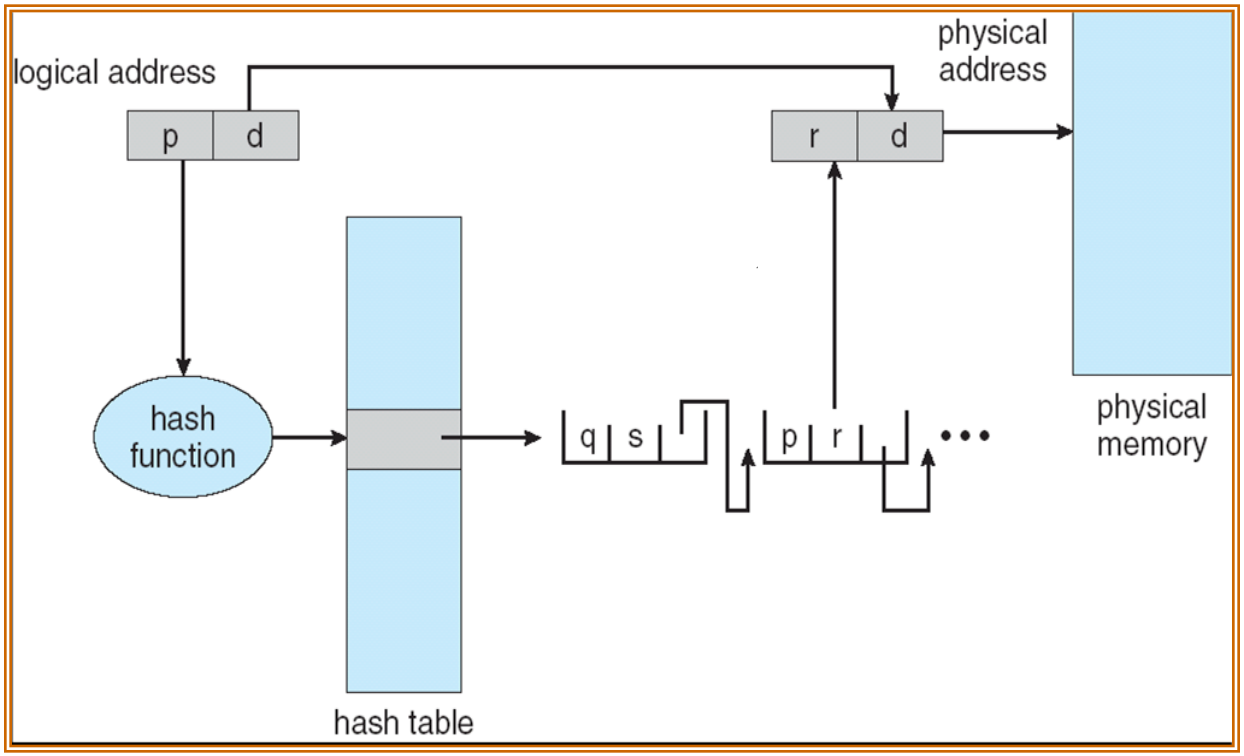


Hierarchical PT

- 64-bits address space with page size 8 KB
  - 51 bits page number → 2 Peta (2048 Tera) Byte PT
- It is problem for hierarchical PT too:
  - Each level brings new delay and overhead, 7 levels will be very slow
- UltraSparc – 64 bits ~ 7 level → wrong
- Linux – 64 bits (Windows similar)
  - Trick: logical address uses only 43 bits, other bits are ignored
  - Logical address space has only 8 TB
  - 3 level by 10 bits of address
  - 13 bits offset inside page
  - It is useful solution

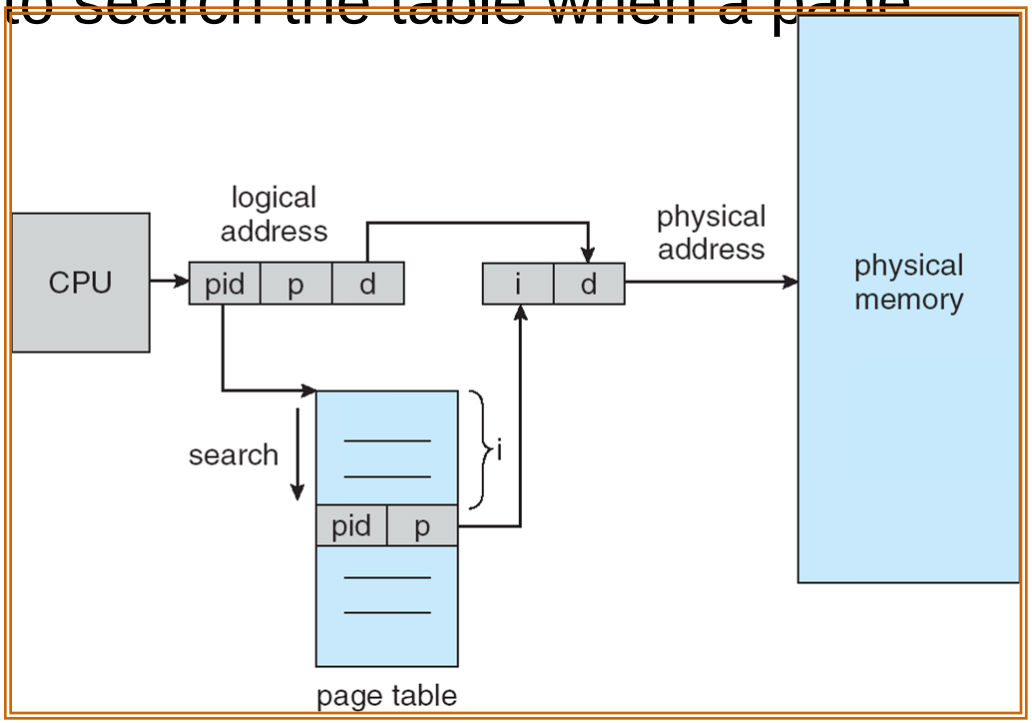
Hashed Page Tables

- 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 consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- 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



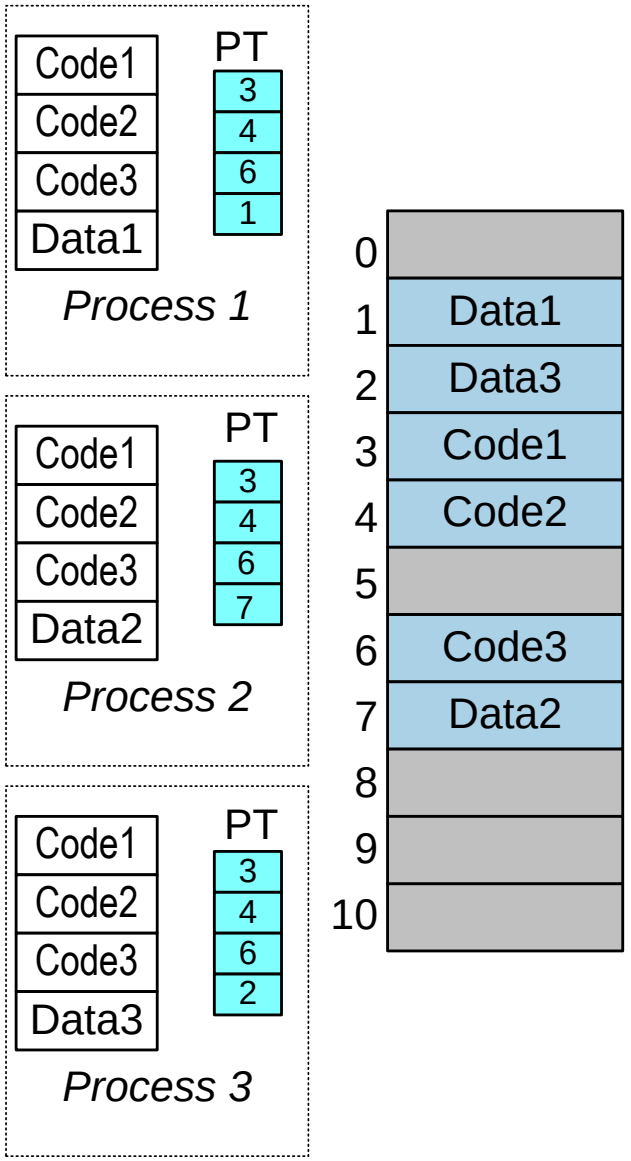
Shared Pages

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



Three instances of a program

Segmentation with paging

- Combination of both methods
- Keeps advantages of segmentation, mainly precise limitation of memory space
- Simplifies placing of segments into virtual memory. Memory fragmentation is limited to page size.
- Segmentation table ST can contain
  - address of page table for this segment PT
  - Or linear address this address is used as virtual address for paging

Segmentation with paging

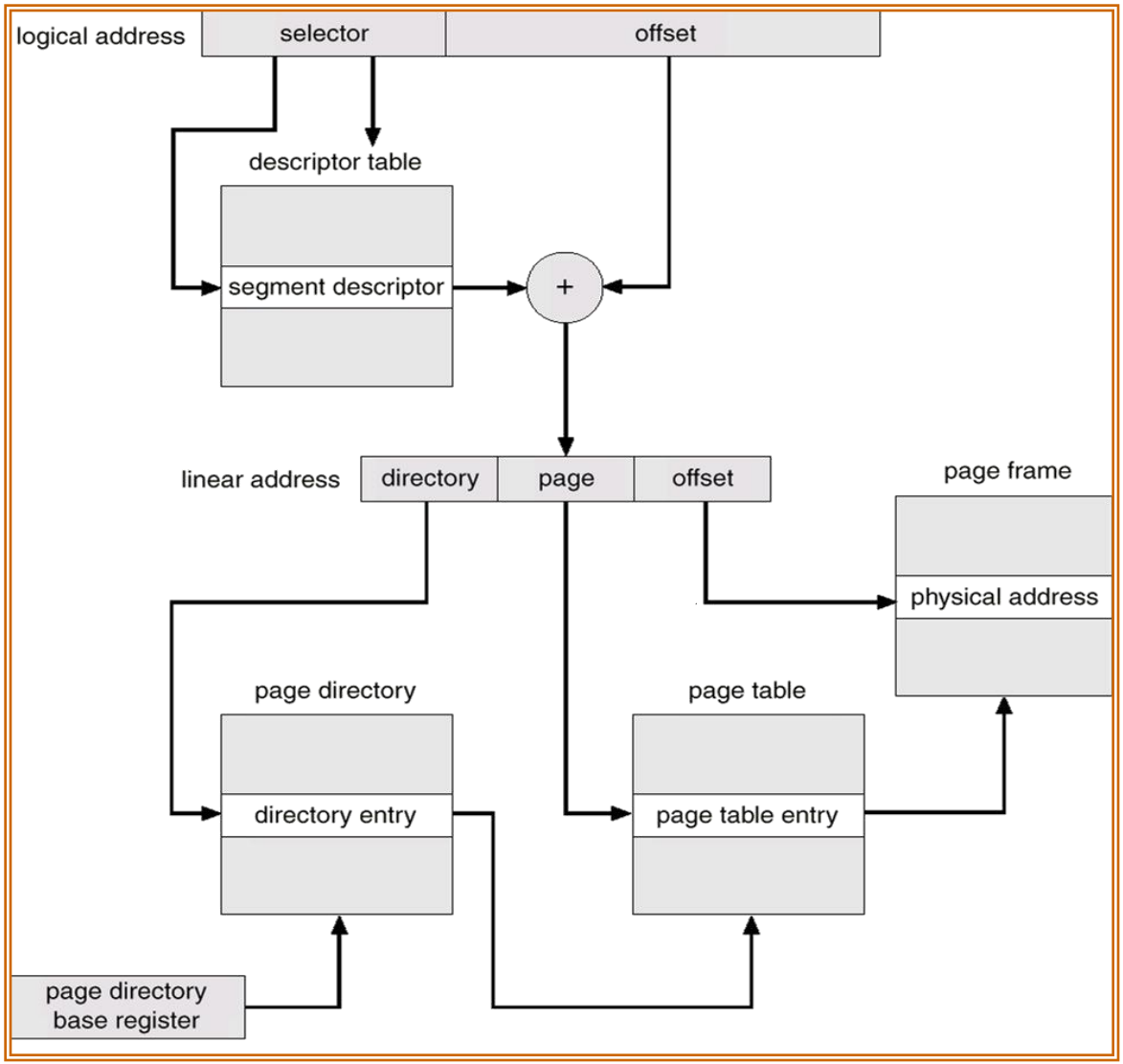
- Segmentation with paging is supported by architecture IA-32 (e.g. INTEL-Pentium)
- IA-32 transformation from logical address space to physical address space with different modes:
  - logical linear space (4 GB), transformation **identity**
    - Used only by drivers and OS
  - logical linear space (4 GB), **paging**,
    - 1024 oblastí à 4 MB, délka stránky 4 KB, 1024 tabulek stránek, každá tabulka stránek má 1024 řádků
    - Používají implementace UNIX na INTEL-Pentium
  - logical 2D address (segemnt, offset), **segmentation**
    - 2^{16} =16384 of segments each 4 GB ~ 64 TB
  - logical 2D address (segemnt, offset), **segmentatation with paging**
    - Segments select part of linear space, this linear space uses paging
    - Used by windows and linux

Segmentation with paging IA-32

- 16 K of segments with maximal size 4 GB for each segment
- 2 logic subspaces (descriptor TI = 0 / 1)
  - 8 K private segments – Local Description Table, LDT
  - 8 K shared segments – Global Description Table, GDT
- Logic addres = (segment descriptor, offset)
  - offset = 32-bits address with paging
  - Segment descriptor
    - 13 bits segemnt number,
    - 1 bit *descriptorTI*,
    - 2 bits Privilege levels : OS kernel, ... ,application
    - Rights for *r*/*w*/*e* at page level
- Linear address space inside segemnt with hierchical page table with 2 levels
  - Page size 4 KB, offset inside page 12 bits,
  - Page number 2x10 bits

Segmentation with Paging – Intel 386

- IA32 architecture uses segmentation with paging for memory management with a two-level paging scheme



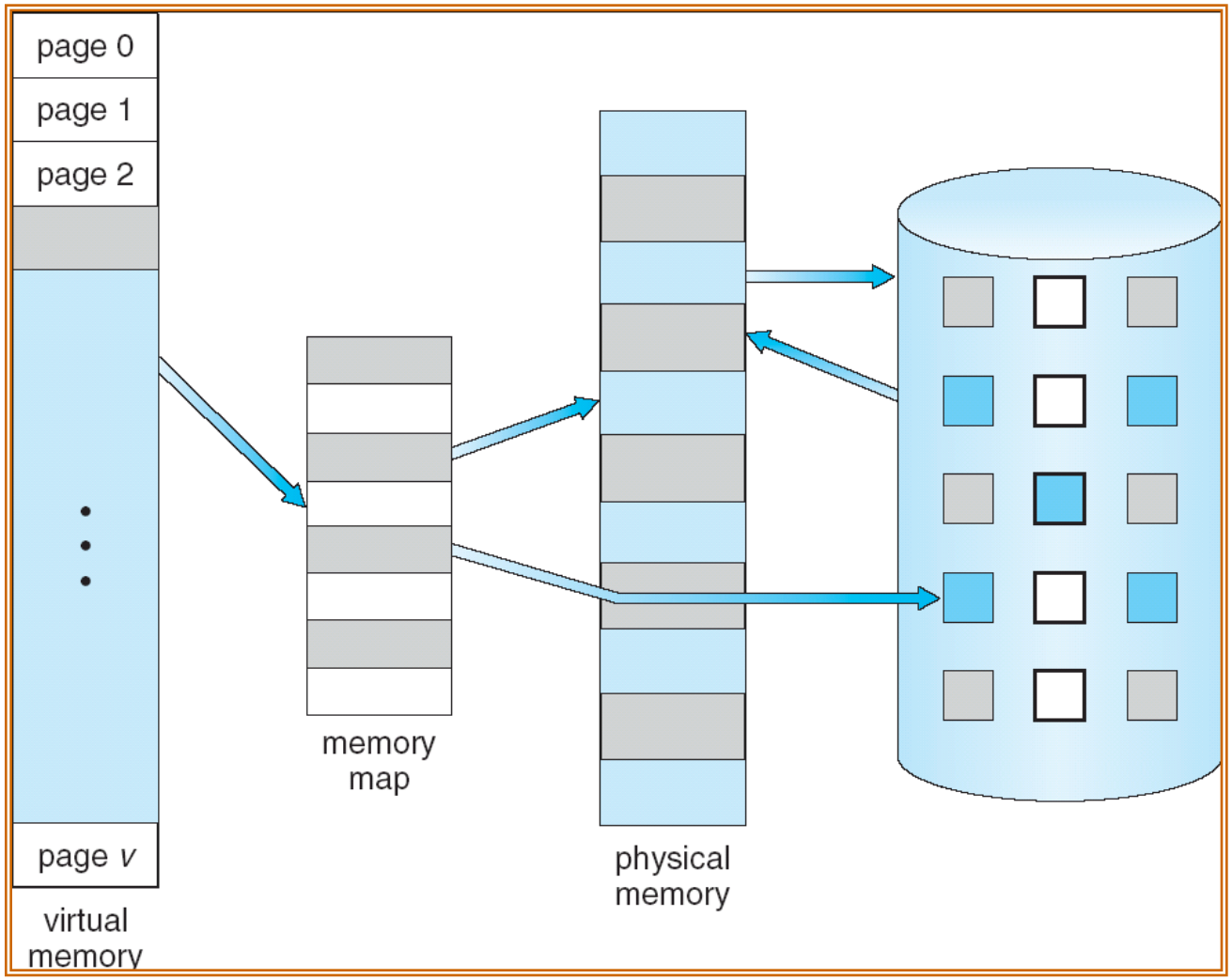
Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

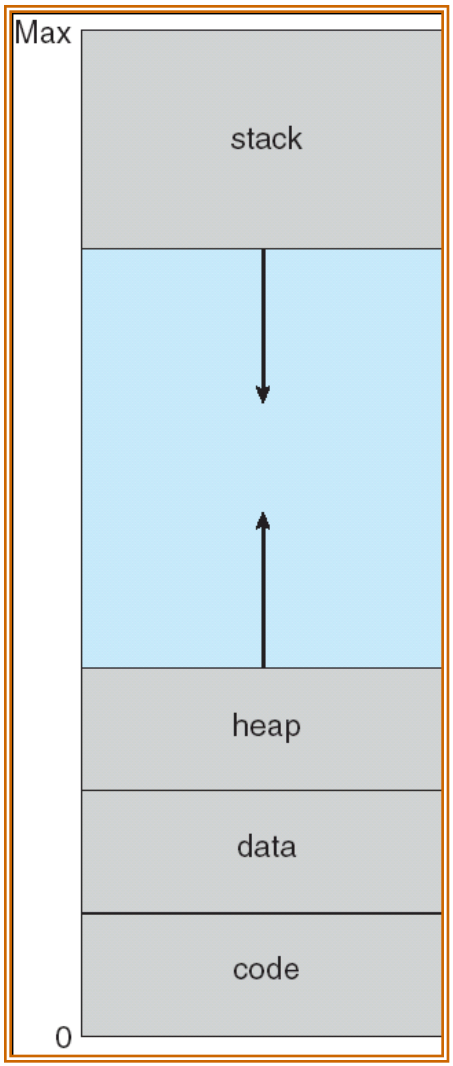
Virtual memory

- Virtual memory
  - Separation of physical memory from user logical memory space
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
- Synonyms
  - Virtual memory – logical memory
  - Real memory – physical memory

Virtual Memory That is Larger Than Physical Memory

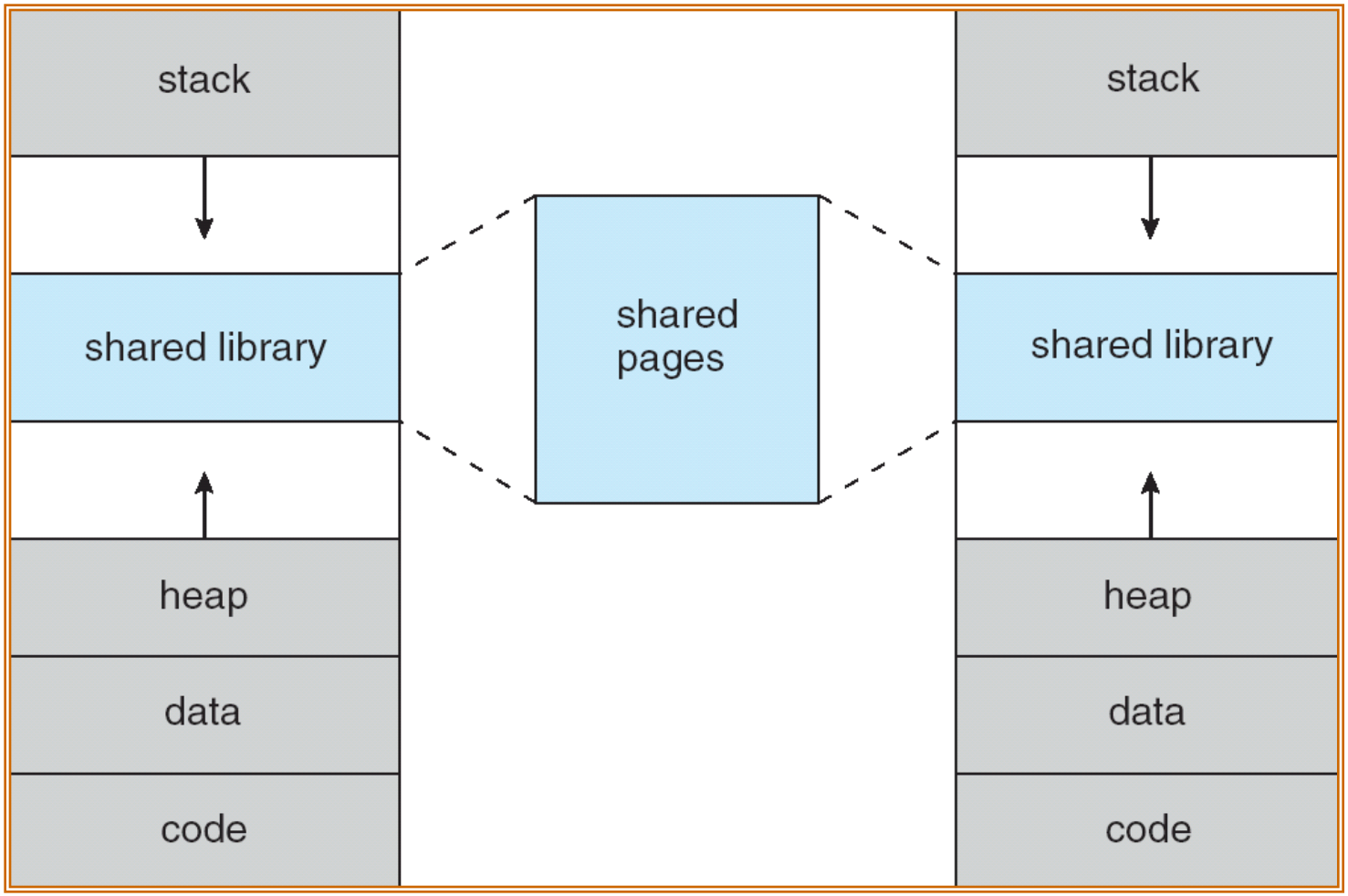


Virtual-address Space



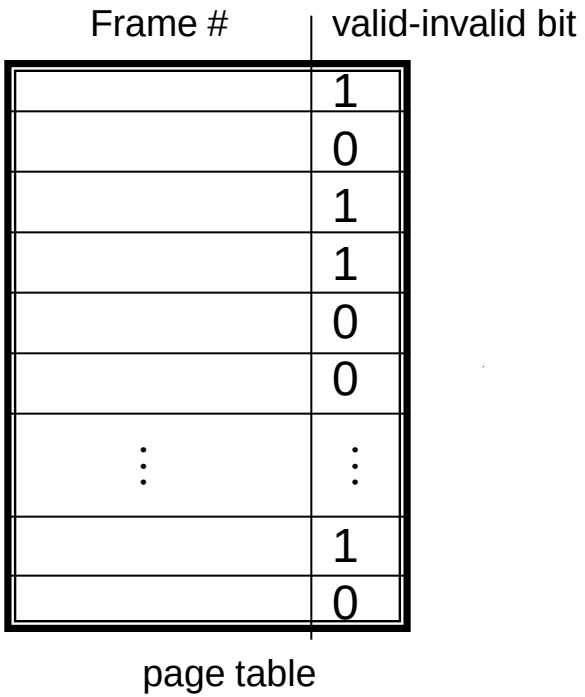
- Process start brings only initial part of the program into real memory. The virtual address space is whole initialized.
- Dynamic exchange of virtual space and physical space is according context reference.
- Translation from virtual to physical space is done by page or segment table
- Each item in this table contains:
  - *valid/invalid* attribute – whether the page if in memory or not
  - *resident set* is set of pages in memory
  - reference outside resident set create *page/segment fault*

Shared Library Using Virtual Memory



Page fault

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

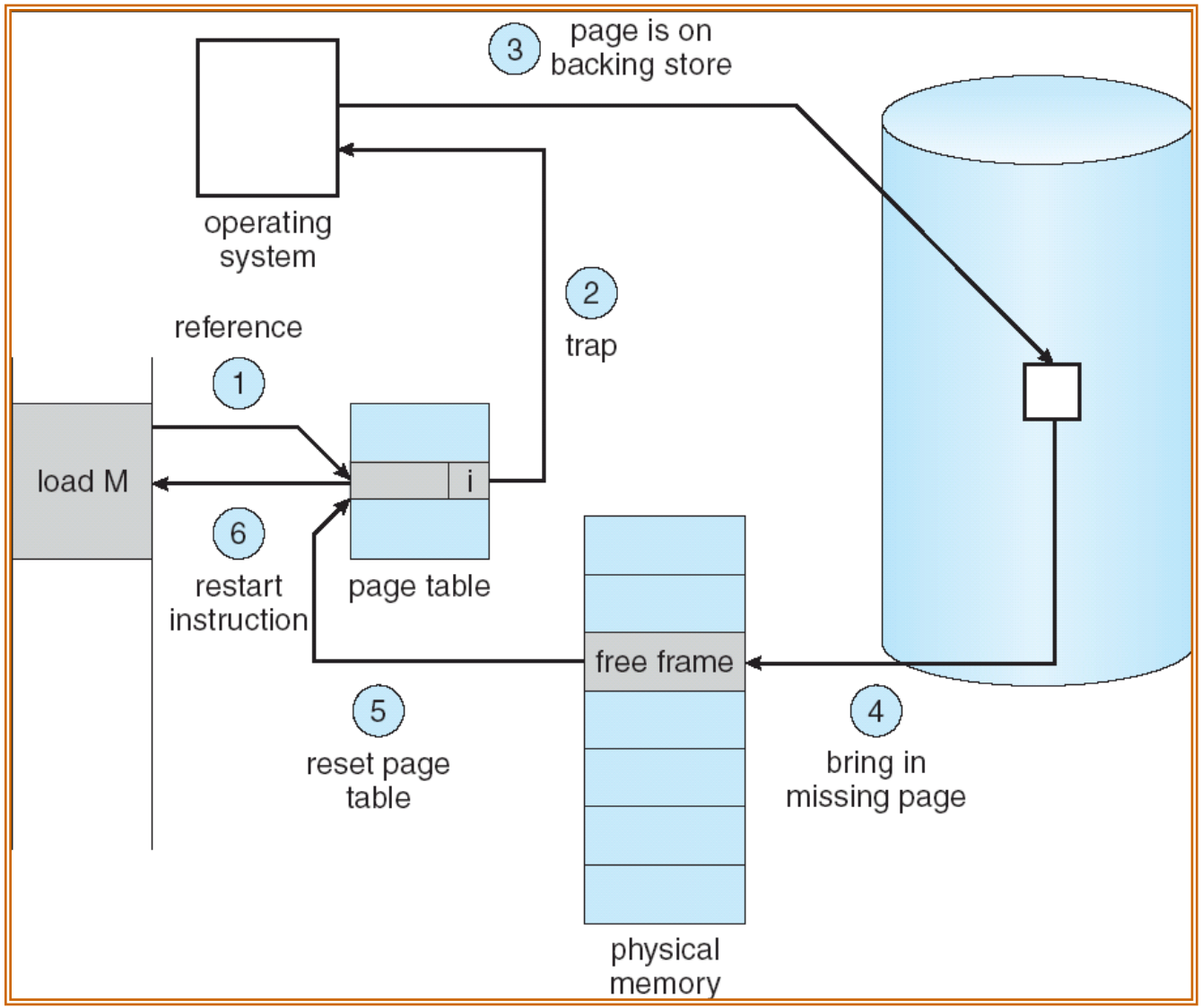
Paging techniques

- Paging implementations
  - *Demand Paging (Demand Segmentation)*
    - Lazy method, do nothing in advance
  - *Paging at process creation*
    - Program is inserted into memory during process start-up
  - *Pre-paging*
    - Load page into memory that will be probably used
  - *Swap pre-fetch*
    - With page fault load neighborhood pages
  - *Pre-cleaning*
    - Dirty pages are stored into disk

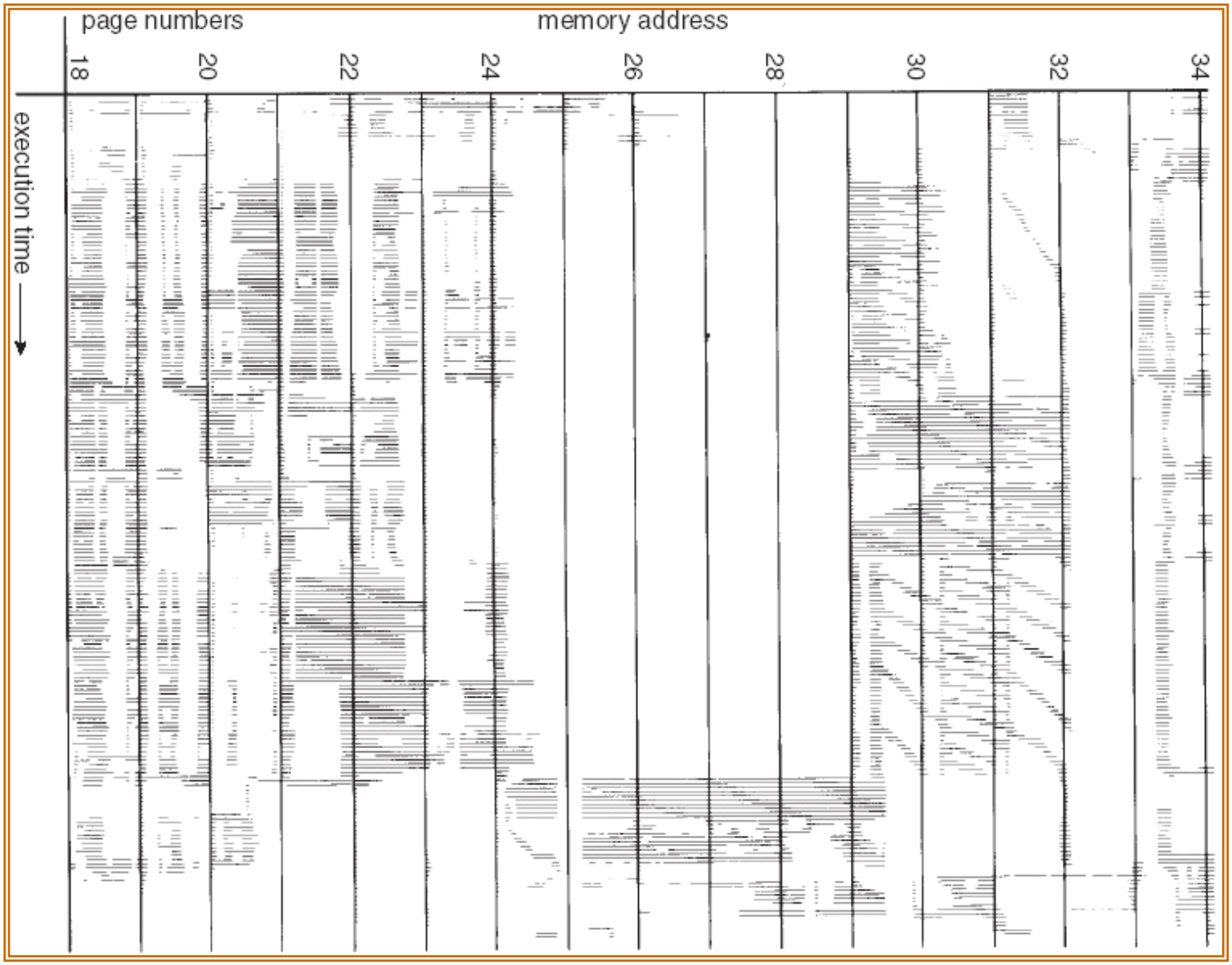
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
  - Slow start of application
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ page fault $\Rightarrow$ bring to memory
- Page fault solution
  - Process with page fault is put to waiting queue
  - OS starts I/O operation to put page into memory
  - Other processes can run
  - After finishing I/O operation the process is marked as ready

Steps in Handling a Page Fault



Locality In A Memory-Reference Pattern



Locality principle

- Reference to instructions and data creates clusters
- Exists time locality and space locality
  - Program execution is (excluding jump and calls) sequential
  - Usually program uses only small number of functions in time interval
  - Iterative approach uses small number of repeating instructions
  - Common data structures are arrays or list of records in neighborhoods memory locations.
- It’s possible to create only approximation of future usage of pages
- Main memory can be full
  - First release memory to get free frames

Other paging techniques

- Improvements of demand paging
  - *Pre-paging*
    - Neighborhood pages in virtual space usually depend and can be loaded together – speedup loading
    - Locality principle – process will probably use the neighborhood page soon
    - Load more pages together
    - Very important for start of the process
    - Advantage: Decrease number of page faults
    - Disadvantage: unused page are loaded too
  - Pre-cleaning
    - If the computer has free capacity for I/O operations, it is possible to run copying of changed (dirty) pages to disk in advance
    - Advantage: to free page very fast, only to change validity bit
    - Disadvantage: The page can be modified in future - boondoggle

End of Lecture 5

Questions?