4

Memory

Uwe R. Zimmer – International University Bremen

References for this chapter

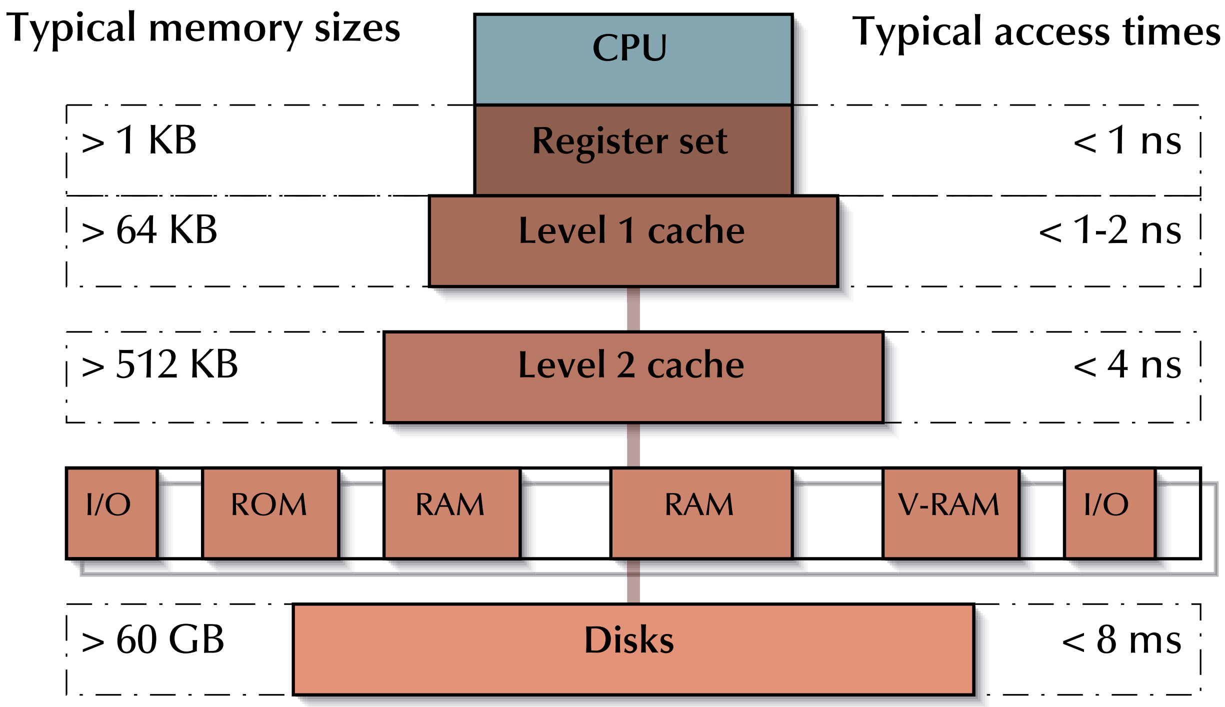
[Silberschatz01] – Chapter 9,10
Abraham Silberschatz, Peter Bear Galvin, Greg Gagne
Operating System Concepts
John Wiley & Sons, Inc., 2001

[Stallings2001] – Chapter 7,8
William Stallings
Operating Systems
Prentice Hall, 2001

all references and some links are available on the course page

Memory levels and fragments

Basic memory hierarchy



What is the challenge?

- Main memory is too small (regardless how large it is)

☞ The operating system needs to place (parts of) processes in and out of main memory during the life-time of the system.

- Swapping memory blocks between primary and secondary memory is an extremely slow operation.

☞ The operating system needs to supply highly efficient strategies to avoid system stalls or unacceptable delays.

Goals / optimization criteria

- Supply address spaces, which are independent from the physically available address space.
- Supply multiple memory modes, e.g. allow processes to reside permanently in main memory
- Support for multiple address spaces

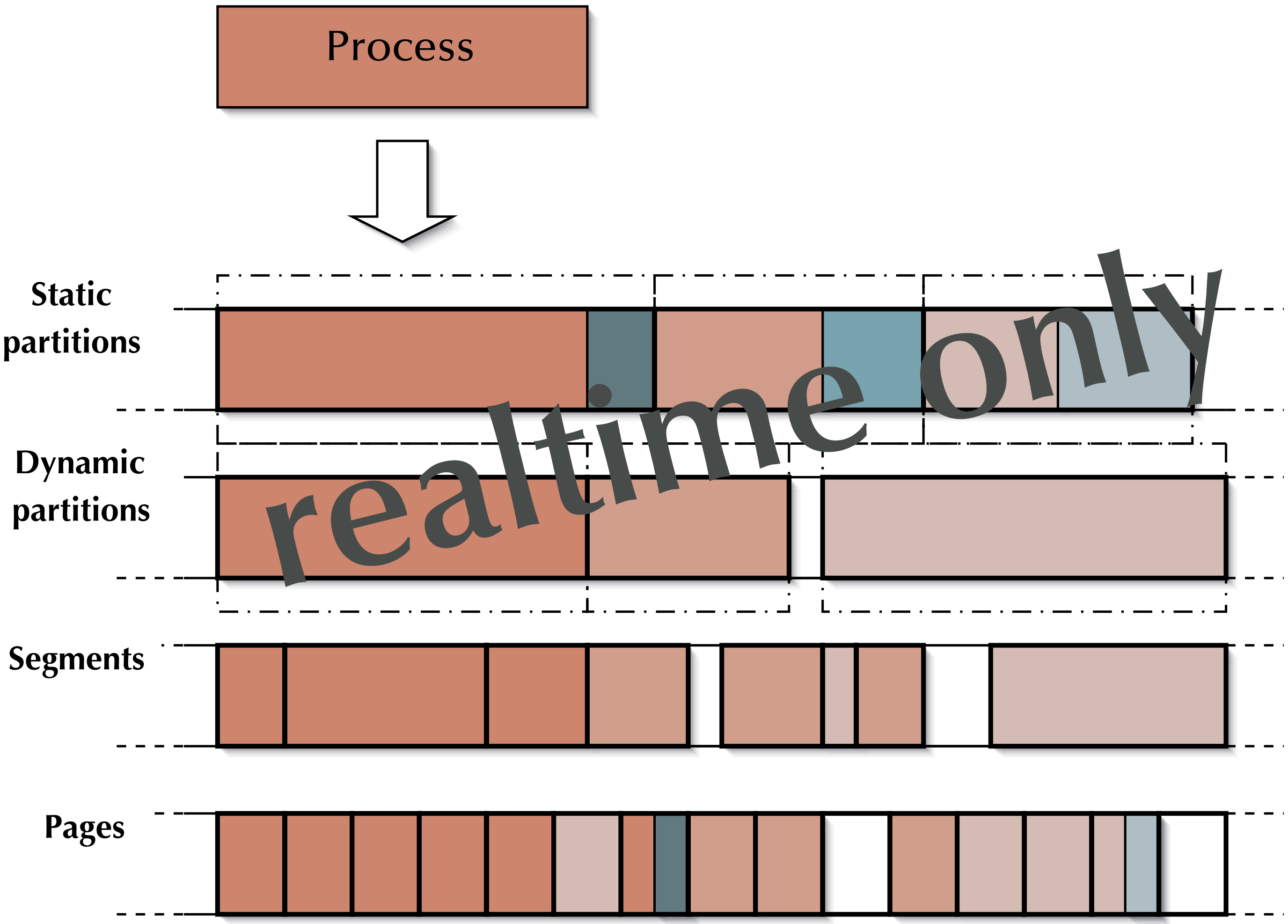
- Protection between address spaces
- Supply methods to share address spaces

- Support memory based I/O methods
- Allow for predictable behaviours of memory accesses
- Minimize any overhead for memory accesses and program executions

Required support

- **Relocation**
  Assembler level addressing modes as well as compilers and linkers need to support relocatable programs and data structures.
- **Protection**
  Memory protection needs hardware support, since the operating system itself has no knowledge which memory cells will be addressed by a specific process next.
- **Sharing**
  The protection scheme needs to be flexible enough to allow for shared memory areas.
- **Control of secondary memory**
  Since swapping speeds between primary and secondary memory is a critical factor, the operating system needs to have close access to the secondary memory interface.
- **Project logical structures to memory modules** (optional)
  It might be useful to supply addressing modes, which allow the use of logical structures in the programs itself as the basis for memory structuring.

Process Mapping



Pros	Cons
simple	internal & strong external fragmentation
no internal fragmentation	strong external fragmentation
no internal fragmentation	external fragmentation
no external fragmentation	a small amount of internal fragmentation

The step from pagination/segmentation to

Virtual addressing

Segmentation / Paging:

- all memory references are logical addresses
- there is support to translate logical to physical addresses at run-time
- processes may be moved in memory and suspended to or loaded from secondary storage
- processes are divided in pages or segments (or both)
- pages or segments can be loaded in any order into primary memory
  (i.e. they need not to be dense or in sequence)

☞ Virtual addressing:

- not all pages or segments need to be loaded in order to run a process

Translating virtual to physical addresses

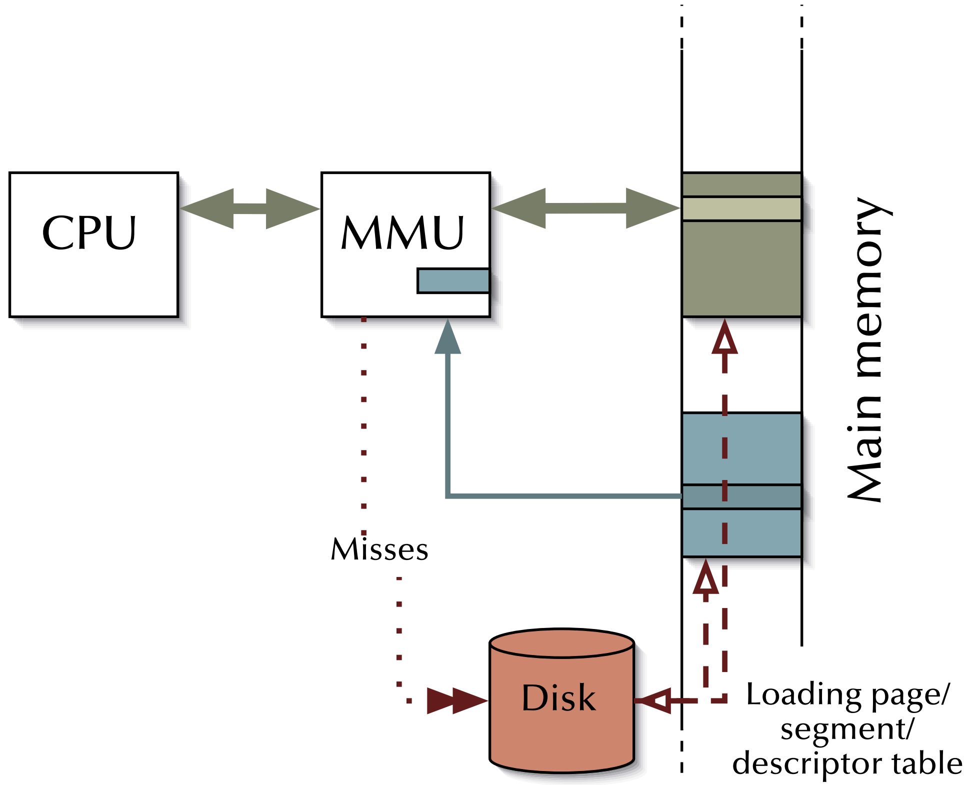
MMU

1. **Translate virtual to physical addresses**

   without any delay in most cases.

2. **Provide memory protection**

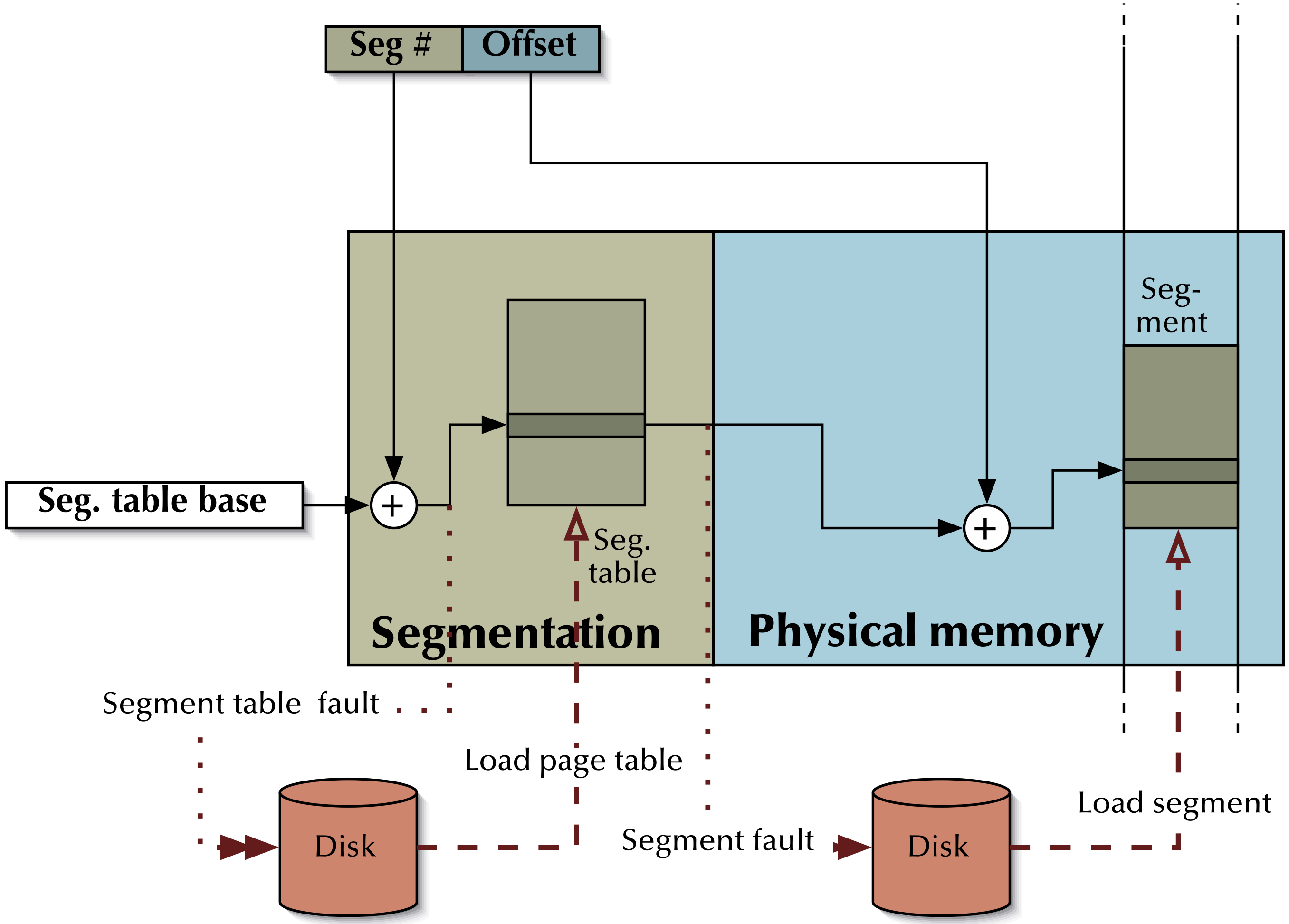
   according to the attributes, which are attached to individual memory areas in form of page or segment descriptors.



Memory – Segmentation

- Segment lengths is stored in segment table ☞ needs to be evaluated by the memory protection unit.
- Segment base address and offset need to be added.
- Parts of segment tables as well as segments themselves can be suspended to secondary memory.

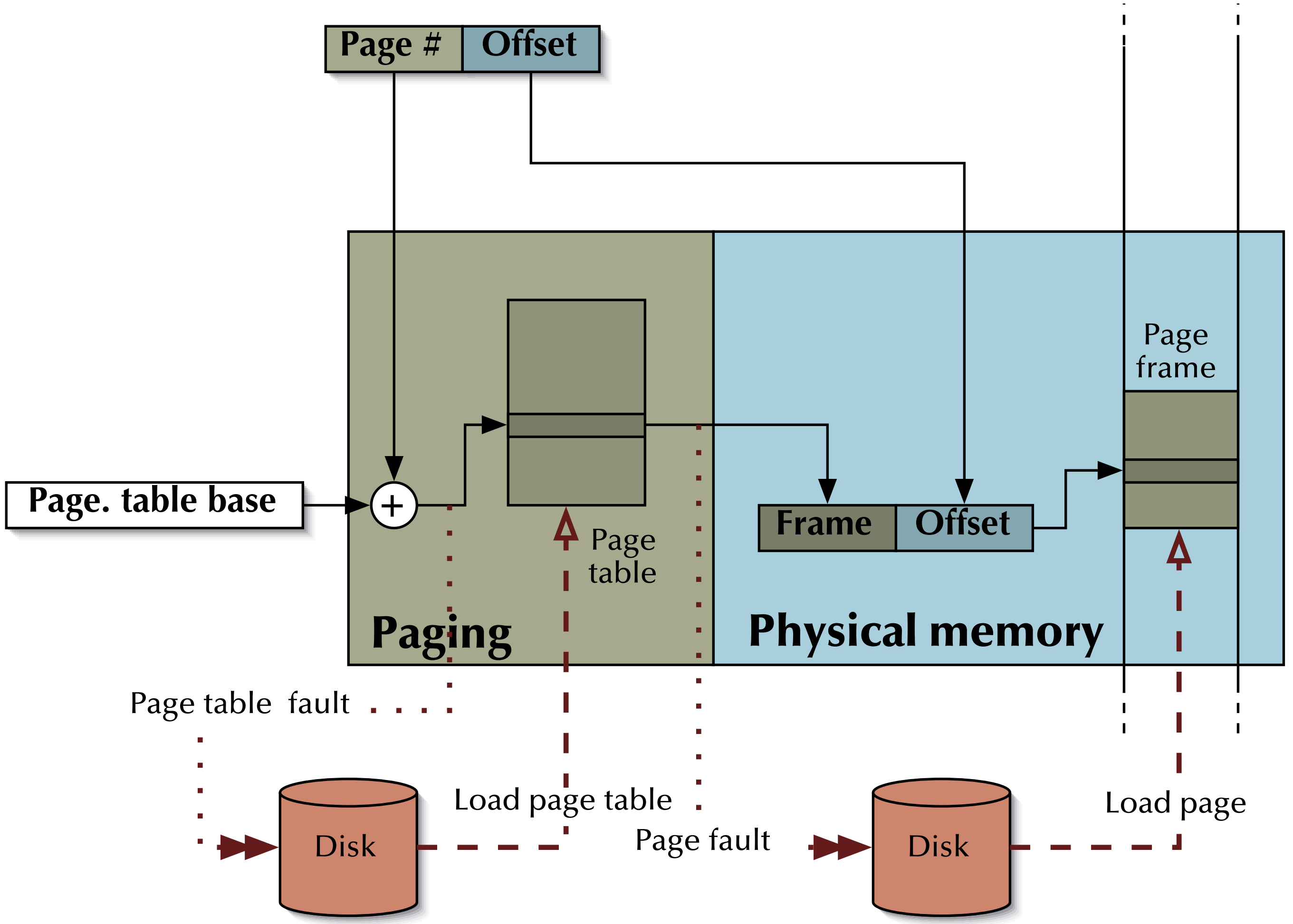
e.g. Intel x86



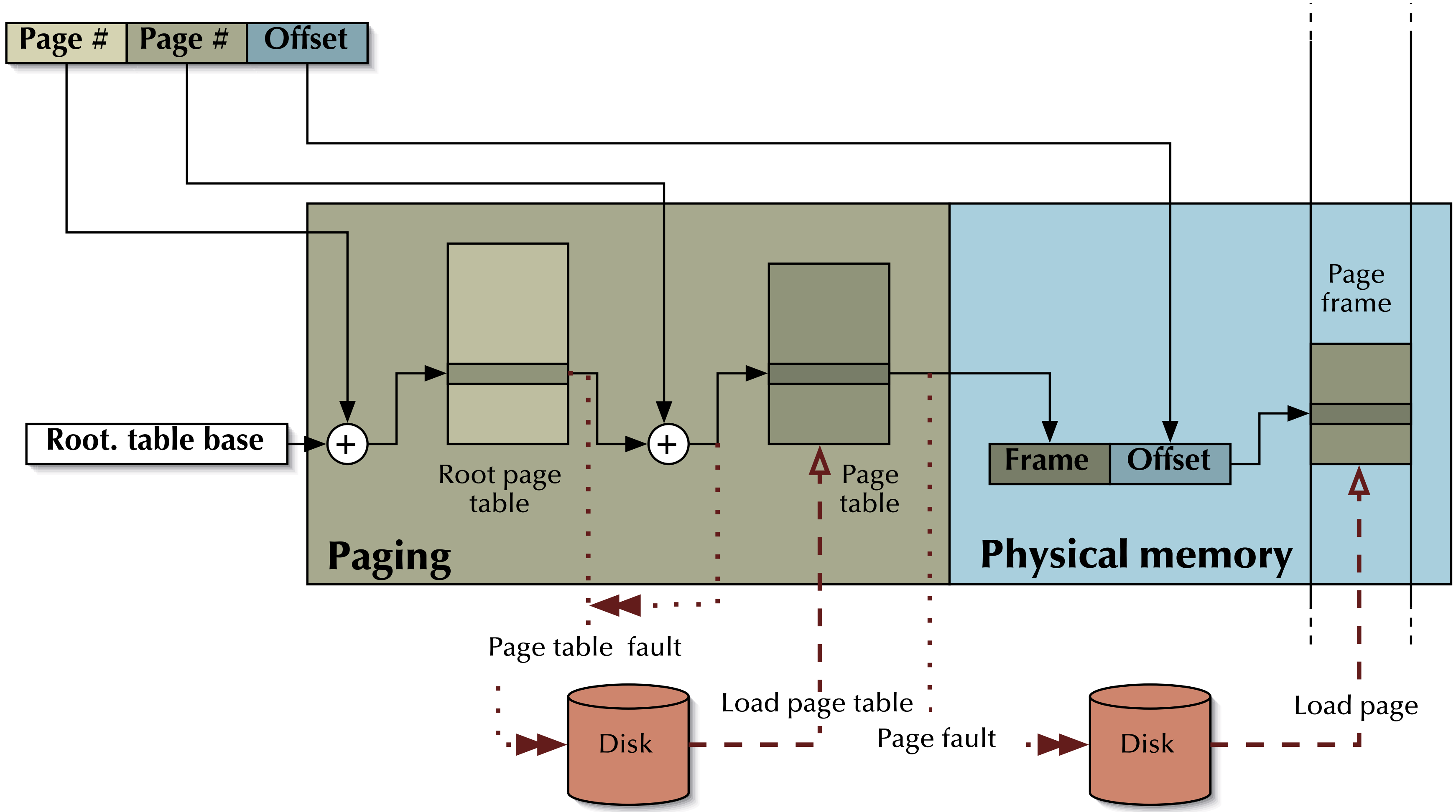
Memory – Paging

- Page frame address and address offset can be concatenated.
- Parts of page tables as well as pages themselves can be suspended to secondary memory (into 'frames').
- Page tables would be very large for modern processors (32-64 bit addressing)

not implemented in this pure form.



Memory – Multi stage page tables

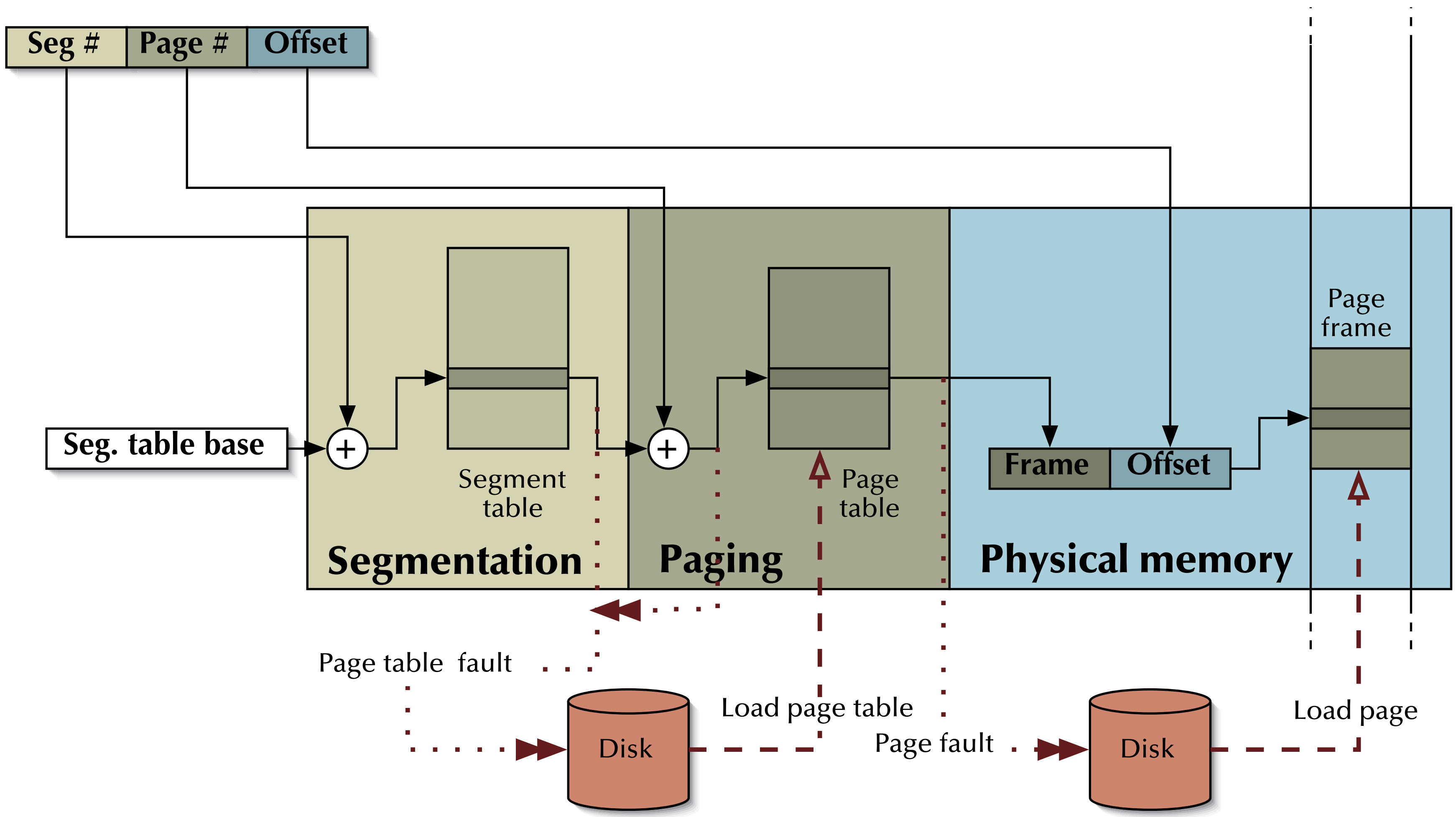


- Reducing page table sizes
- Up to four page levels (Sparc)
- More memory accesses required.

Sparc, PowerPC, Alpha, HP

Memory – Segmentation & Paging

- Allow segmentation for logical structure
- Allow paging for effective virtual memory management



x86, (PowerPC)

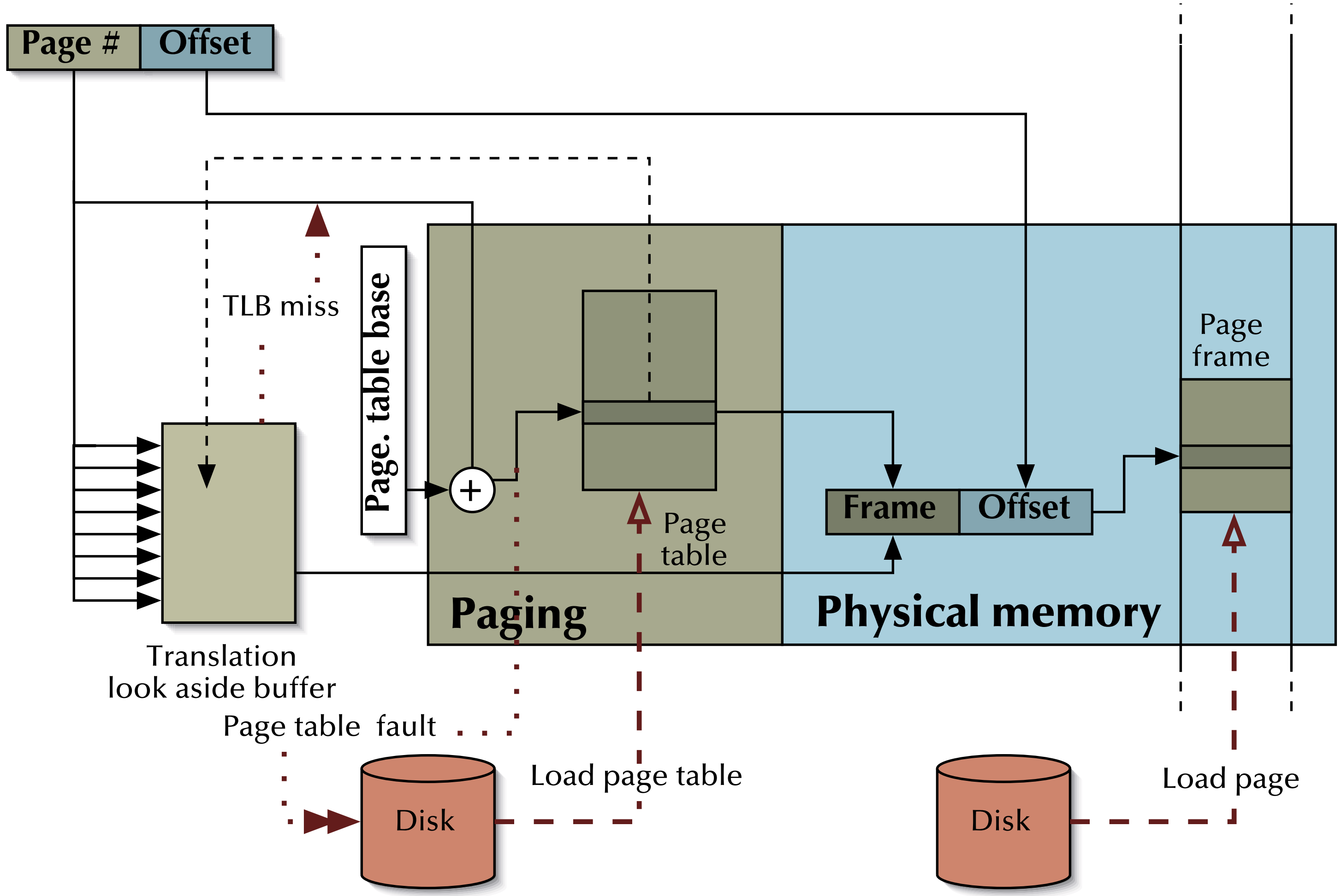
Memory – Translation look aside buffers

- Accessing page tables for each access is ineffective.

☞ Introducing address translation caches: **Translation look aside buffers (tlb).**

- Access cache (tlb) - memory - disk (in this order) for address translation

all modern MMUs

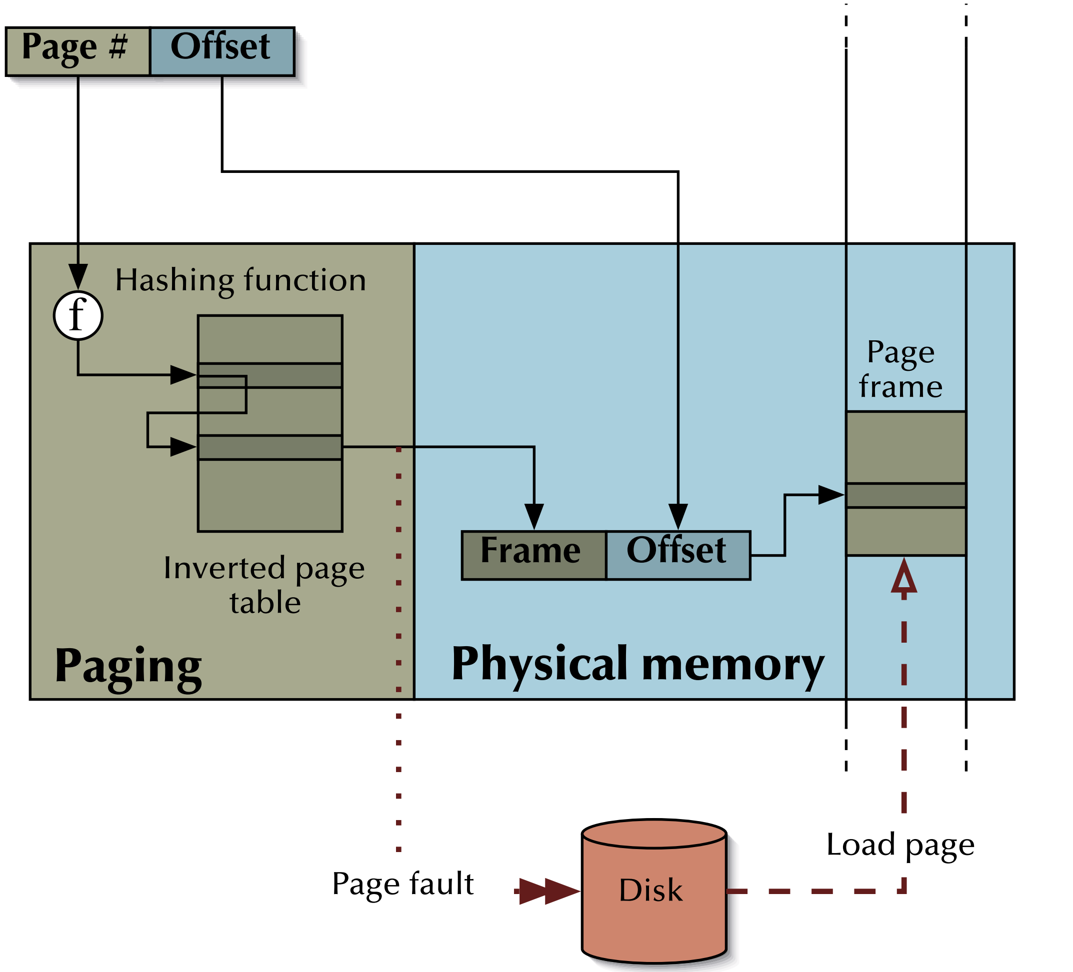


Memory – Inverted page tables

- Forward page tables grow with the size of the virtual address space.
- The number of loaded pages is bound by the physical memory.

☞ Keep only the loaded pages in the page table and resolve the virtual addresses via a hash table: ☞ **Inverted page tables (ipt)**

- IPTs are not suspended to secondary memory, but more than one access is required to translate the page number.

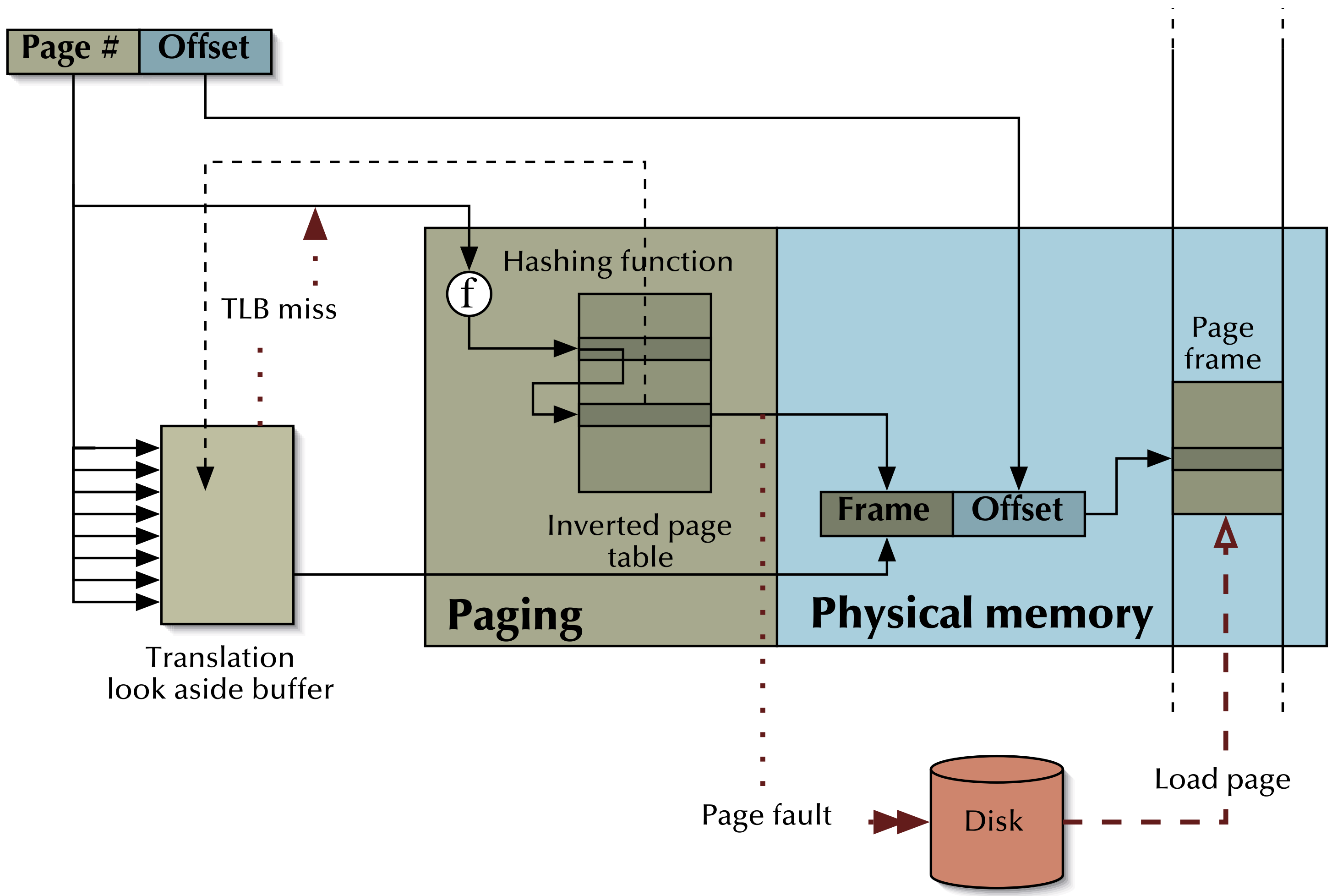


not implemented in this pure form.

Memory – Translation look aside & Inverted page tables

- Combining translation look aside buffers and inverted page tables.
- Mostly no delay (look aside buffer).
- Short delay if tlb misses (inverted page table).
- No page table loading.

PowerPC, UltraSparc



Some current MMU implementations

	Physical addresses	Virtual addresses	TLB size	Segments	Pages	Inverted/hashed tables
Pentium 4	36 bit	32 bit (per segment)	64	different types	4 k, 4 M (optional)	-
Itanium 2	50 bit	64 bit	4*32	-	4k … 4 G	-
Power PC 604	32 bit	52 bit	256	< 256 MB, (optional)	4 k	yes
Power PC 970	42 bit	64 bit	1024	< 256 MB, (optional)	4 k	yes
UltraSparc	36 bit	64 bit	64	-	8 k … 4 M	yes
Alpha	41 bit	64 bit	256	-	8 k … 4 M	-

Design alternatives

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - *when* to *load* a page/segment? ☞ **fetching**
  - *where* to *place* a page/segment? ☞ **placement**
  - *which* page/segment to *suspend*? ☞ **replacement**
  - *how many* pages/segments to *load* for a specific process? ☞ **resident set management**
  - *when* to *suspend* a page/segment? ☞ **cleaning**
  - *which* processes to run/suspend? ☞ **load control**

Fetching

- **Demand paging**:

  Fetch pages only if and exactly when requested by a reference to an address inside this page.

☞ may lead to a burst of page faults in some situations (e.g. starting a process).

☞ reduces the transfer between primary and secondary storage to a minimum.

- **Prepaging**:

  Predict which pages will also be required in the near future and pre-load them (together with the currently requested page).

☞ pages may be loaded, which will be never referenced

☞ multiple page loads can be more efficient if organized as a few transfers of a larger blocks

Fetching

- **Demand paging**:

  Fetch pages only if and exactly when requested by a reference to an address inside this page.

☞ may lead to a burst of page faults in some situations (e.g. starting a process).

☞ reduces the transfer between primary and secondary storage to a minimum.

- **Prepaging**:

  Predict which pages will also be required in the near future and pre-load them (together with the currently requested page).

☞ pages may be loaded, which will be never referenced

☞ multiple page loads can be more efficient if organized as a few transfers of a larger blocks

Most systems will combine both pure forms

Placement

☞ Required for partition or pure segmentation systems

apply standard 'best-fit', 'first-fit', etc. strategies to minimize fragmentation
– there is a trade-off between minimal fragmentation and minimal placement overhead

☞ Irrelevant for all paging or mixed segmentation/paging systems

external fragmentation is not an issue here

Replacement

In order to load a new page, another page need to be suspended ☞ which one?

- **Optimal**:
  the page which will not be referenced for the longest period of *future* time

- **Least Recently Used** (**LRU**):
  the page which has not be referenced for the longest period of *past* time

- **First-In-First-Out** (**FIFO**):
  the page which resides in primary memory for the longest period of *past* time

Replacement

The practical implementation aspect of replacement algorithms:

- **Optimal**:
  ☞ can only be implemented, if all *future* memory references are known ☞ ✘

- **Least Recently Used** (**LRU**):
  ☞ can only be implemented, if all past access times/order are known ☞ check hardware support

- **First-In-First-Out** (**FIFO**):
  ☞ can be implemented without any hardware support ☞ ✔

Replacement

Full LRU implementations:

- ***Counter*** or ***time-of-access*** field in the page table:
  Update this entry with each reference to this page

  ☞ need to be supplied by hardware (not implemented in any practical system)

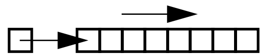
- ***Page stack***:
  bring a reference to the page on top of a stack with each access to this page
  (and replace the pages at the bottom of the stack)

  ☞ need to be supplied by hardware (not implemented in any practical system)

Replacement

LRU-approximations:

- **Reference-bit-shift-history algorithm***:*

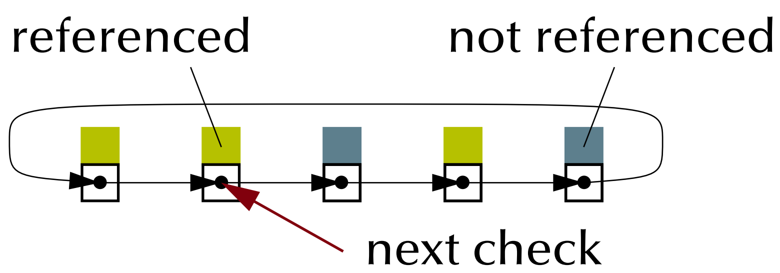
  Shift the reference bit of each page into a bit-field () in each page table entry at regular intervals (employing a timer-interrupt).
  Interpret the resulting bit-field as an integer and replace the page with the smallest value

  ☞ requires a reference-bit, which is updates by hardware, as well as a hardware timer (usually provided).

Replacement

LRU-approximations:

- **Second-chance (clock) algorithm**:



Implement a circular list of all pages. Search the list for a not referenced page:

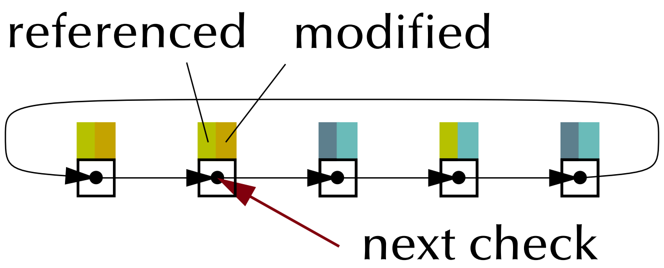
WHILE page was referenced DO
    reset reference bit and proceed to next page
END WHILE

☞ requires a reference-bit, which is updates by hardware (usually provided).

Replacement

LRU-approximations:

- **Enhanced second-chance (clock) algorithm**:



Replace pages applying the priorities:

- not referenced (first scan)
- referenced-but-not-modified (second scan)
- referenced-and-modified

☞ requires a reference and a modified-bit, which is updates by hardware (usually provided).

Replacement

Performances:

- **Optimal**:
  obviously the best algorithm — impossible to implement

- **Least Recently Used** (**LRU**):
  good approximation of the optimal algorithm — cannot be implemented in any current system

- **Approximated Least Recently Used** (**LRU**):
  approximates the performance of LRU — can be implemented in most systems

- **First-In-First-Out** (**FIFO**):
  performs worst — can be implemented in any system

Resident set management

How many pages are assigned to a specific process:

- too many:
  - the number of resident processes is reduced
  - due to localities, there is no noticeable speed-up for the specific process
- too few:
  - significant increase in the page-fault rate

☞ Challenge: find the essential working set of pages for each process at any given time

Resident set management

Strategies:

- Number of allocated pages per process can be
  - *fixed*
  - or *variable*
- Replacement can be either
  - *local* (inside each process' page set) – only possibility for fixed allocation scenes
  - *prioritized* (allow higher priority processes to expand their page sets)
  - or *global* (replace pages regardless of the processes which are using them)

Resident set management

☞ Challenge:
find the essential working page set for each process at any given time

- Calculating the optimal working set, required full knowledge of the future process behaviour

- Many approximations are suggested (and implemented), mostly employing:

  Page Fault Frequencies (PFF)
  or related statistical information on the past process behaviour

  Problems:
  - "the past does not always predict the future"
    i.e. multiple locality assumptions must hold

Cleaning

- **Demand cleaning**:

  Clean pages only if and exactly when a free pages is required.

☞ slows down process reaction times, since each page fault will result in a page cleaning.

☞ reduces the total transfer between primary and secondary storage to a minimum.

- **Precleaning**:

  Clean multiple pages according to replacement criteria introduced above
  before a page fault occurs.

☞ too many pages might be cleaned, resulting in an increase of page faults

☞ multiple page cleanings can be more efficient if organized as a few transfers of a larger blocks

Cleaning

- **Demand cleaning**:

  Clean pages only if and exactly when a free pages is required.

☞ slows down process reaction times, since each page fault will result in a page cleaning.

☞ reduces the total transfer between primary and secondary storage to a minimum.

- **Precleaning**:

  Clean multiple pages according to replacement criteria introduced above before a page fault occurs.

☞ too many pages might be cleaned, resulting in an increase of page faults

☞ multiple page cleanings can be more efficient if organized as a few transfers of a larger blocks

Most systems will combine both pure forms

Load Control

How many processes will be resident in primary memory?

- More processes in primary memory implies less pages per process
- Beyond a critical threshold of pages per process, the page fault rate rises significantly

☞ **Thrashing** occurs

- The overall performance of the system is approaching nil,
  since most of the time is spent for page loads

☞ Reduce the number of resident processes immediately

Load Control

Which process is to be suspended?

- Lowest priority process
- Process with the highest page fault frequency
- Process with the smallest current resident page set
- Process with the largest current resident page set
- Last activated process
- Process with the largest remaining execution time (see scheduling)

Design alternatives

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - *when* to *load* a page/segment? ☞ **fetching**
  - *where* to *place* a page/segment? ☞ **placement**
  - *which* page/segment to *suspend*? ☞ **replacement**
  - *how many* pages/segments to *load* for a specific process? ☞ **resident set management**
  - *when* to *suspend* a page/segment? ☞ **cleaning**
  - *which* processes to run/suspend? ☞ **load control**

Designing an OS memory module

Design alternatives

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - *when* to *load* a page/segment? ☞ **fetching**
  - *where* to *place* a page/segment? ☞ **placement**
  - *which* page/segment to *suspend*? ☞ **replacement**
  - *how many* pages/segments to *load* for a specific process? ☞ **resident set management**
  - *when* to *suspend* a page/segment? ☞ **cleaning**
  - *which* processes to run/suspend? ☞ **load control**

Real-time | predictable systems:

no virtual memory!

Summary

Memory

- **Requirements & hardware structures**
    - MMU features & requirements
- **Partitioning, segmentation, paging & virtual memory**
    - Simple segmentation
    - Simple paging, multi-level paging, combined segmentation & paging
    - Translation look aside buffers
    - Hashed tables, Inverted page tables
- **Virtual memory management algorithms**
    - Fetching & placement
    - Replacement
    - Resident set management
    - Cleaning
    - Load control