

*Operating  
Systems:  
Internals  
and Design  
Principles*

# Chapter 8 Virtual Memory

Ninth Edition  
William Stallings

<b>Virtual memory</b>	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
<b>Virtual address</b>	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
<b>Virtual address space</b>	The virtual storage assigned to a process.
<b>Address space</b>	The range of memory addresses available to a process.
<b>Real address</b>	The address of a storage location in main memory.

**Table 8.1 Virtual Memory Terminology**

# Hardware and Control Structures

- Two characteristics fundamental to memory management:
  - 1) All memory references are logical addresses that are dynamically translated into physical addresses at run time
  - 2) A process may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution
- If these two characteristics are present, it is not necessary that all of the pages or segments of a process be in main memory during execution

# Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set
  - Portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

# Execution of a Process

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state

# Implications

- More processes may be maintained in main memory
  - Because only some of the pieces of any particular process are loaded, there is **room for more processes**
  - This leads to **more efficient utilization** of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time
- A process may be larger than all of main memory
  - If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded separately in some sort of overlay strategy
  - With virtual memory based on paging or segmentation, that **job is left to the OS and the hardware**
  - The OS automatically loads pieces of a process into main memory as required

# Real and Virtual Memory

Real  
memory

Main  
memory, the  
actual RAM

Virtual  
memory

Memory on disk

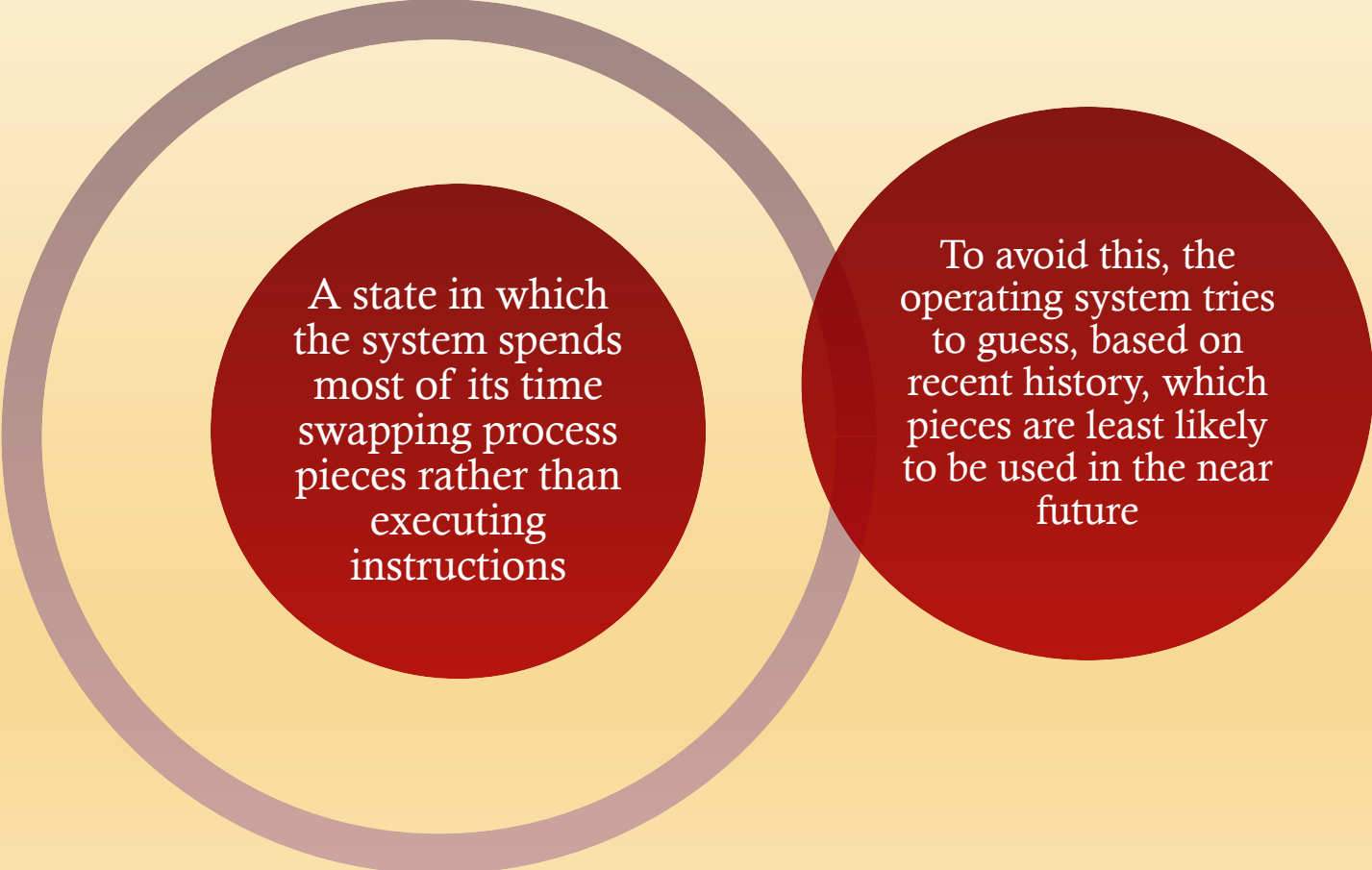
Allows for effective  
multiprogramming  
and relieves the user  
of tight constraints of  
main memory

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames		Main memory not partitioned	
Program broken into pages by the compiler or memory management system		Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	
Internal fragmentation within frames		No internal fragmentation	
No external fragmentation		External fragmentation	
Operating system must maintain a page table for each process showing which frame each page occupies		Operating system must maintain a segment table for each process showing the load address and length of each segment	
Operating system must maintain a free frame list		Operating system must maintain a list of free holes in main memory	
Processor uses page number, offset to calculate absolute address		Processor uses segment number, offset to calculate absolute address	
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

**Table 8.2**  
**Characteristics of Paging and Segmentation**



# Thrashing



A state in which the system spends most of its time swapping process pieces rather than executing instructions

The diagram features a large, light purple circle on the left. Inside it is a smaller, solid dark red circle containing text. To the right of this inner circle is another solid dark red circle, also containing text. The two dark red circles overlap slightly. The background is a light yellow gradient.

To avoid this, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future

# Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future
- Avoids thrashing

# Support Needed for Virtual Memory

For virtual memory to be practical and effective:

- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

# Paging

- The term *virtual memory* is usually associated with systems that employ paging
- Use of paging to achieve virtual memory was first reported for the Atlas computer
- Each process has its own page table
  - Each page table entry (PTE) contains the frame number of the corresponding page in main memory
  - A page table is also needed for a virtual memory scheme based on paging

Virtual Address



Page Table Entry

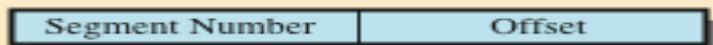


**(a) Paging only**

P: present (P) bit, indicating whether page is in main memory or not.

M: modify (M) bit, indicating whether the contents of the corresponding page have been altered

Virtual Address

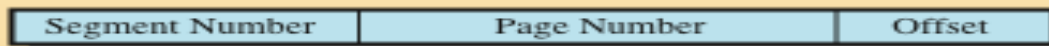


Segment Table Entry



**(b) Segmentation only**

Virtual Address



Segment Table Entry



Page Table Entry



**(c) Combined segmentation and paging**

P= present bit  
M = Modified bit

**Figure 8.1 Typical Memory Management Formats**

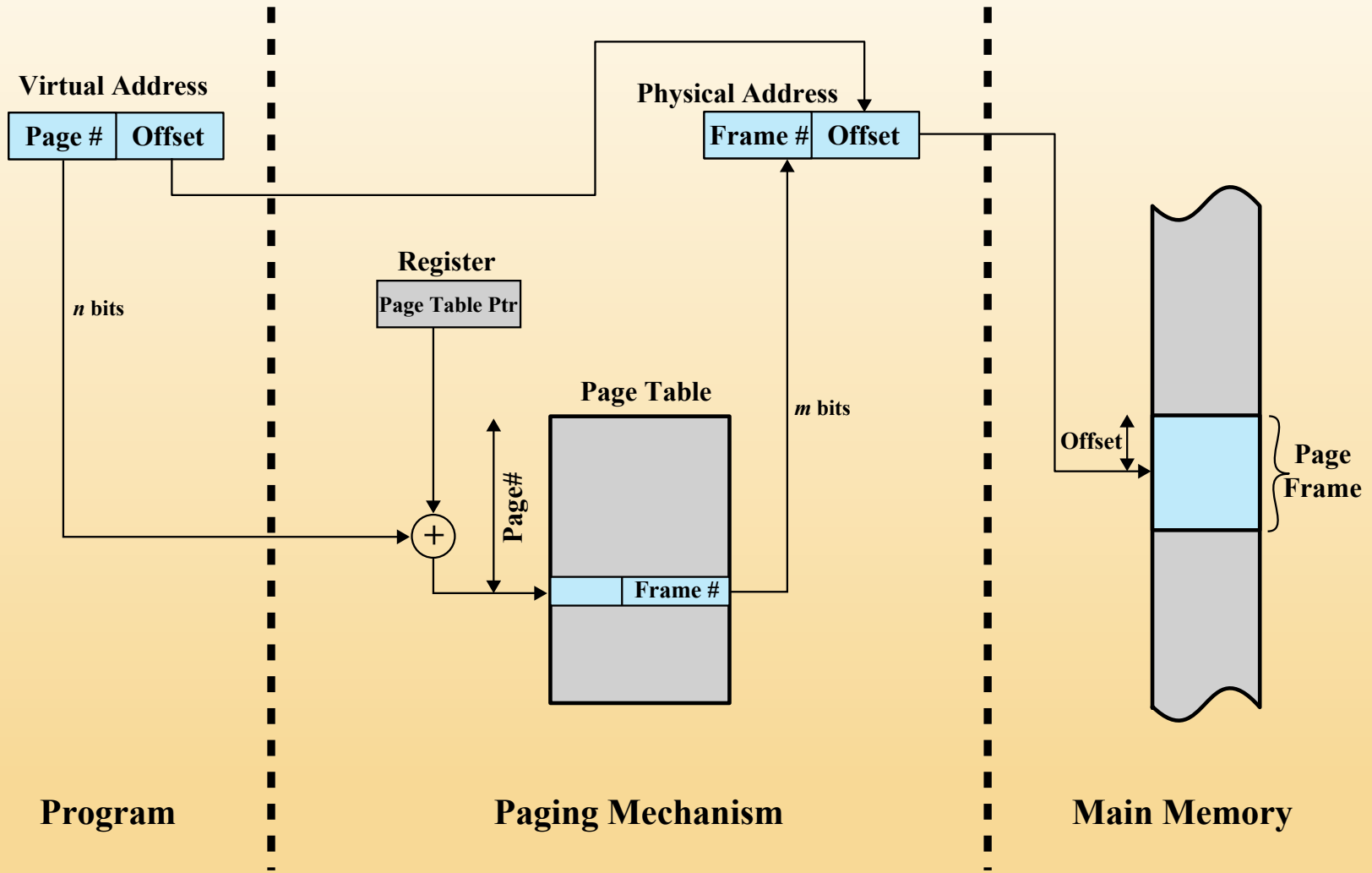
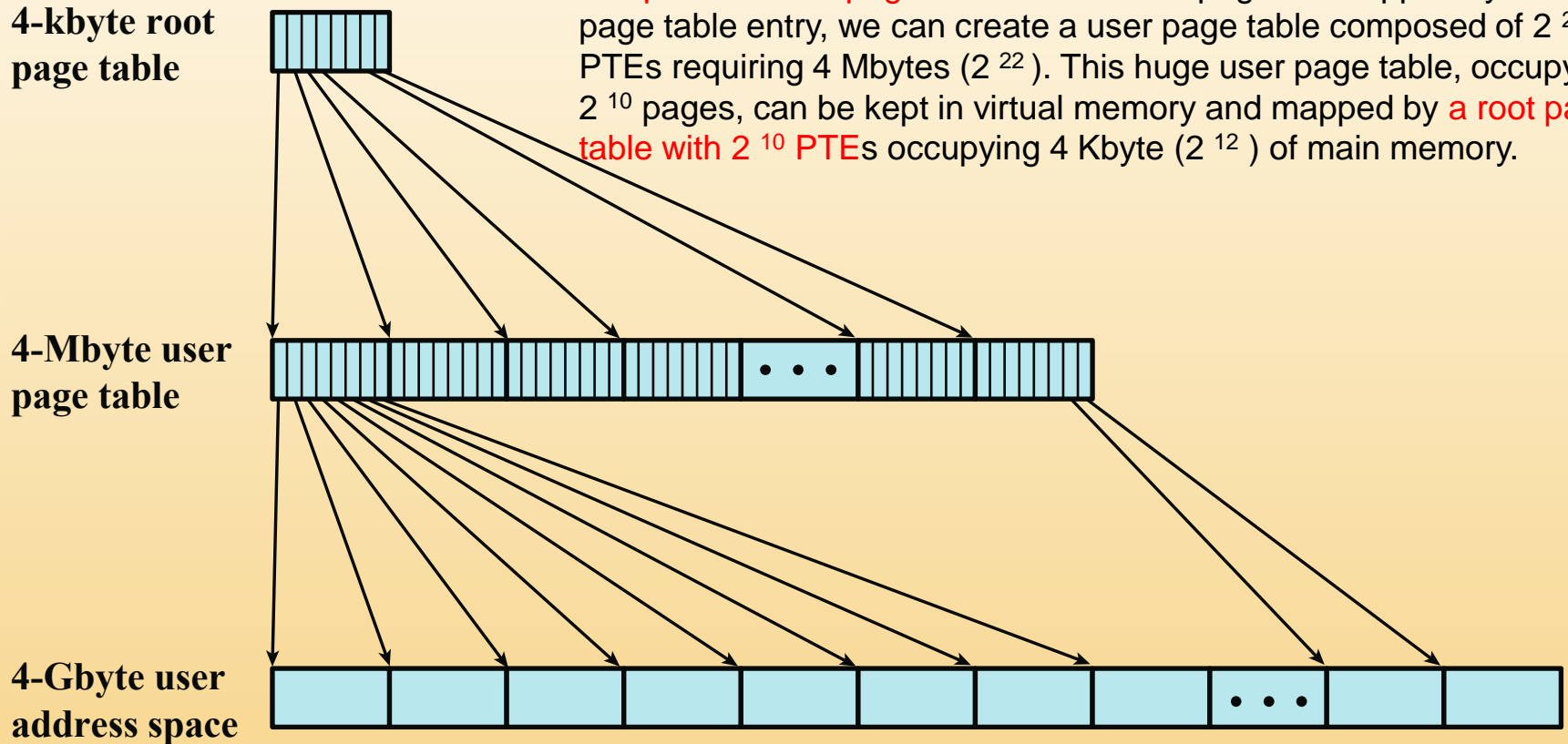
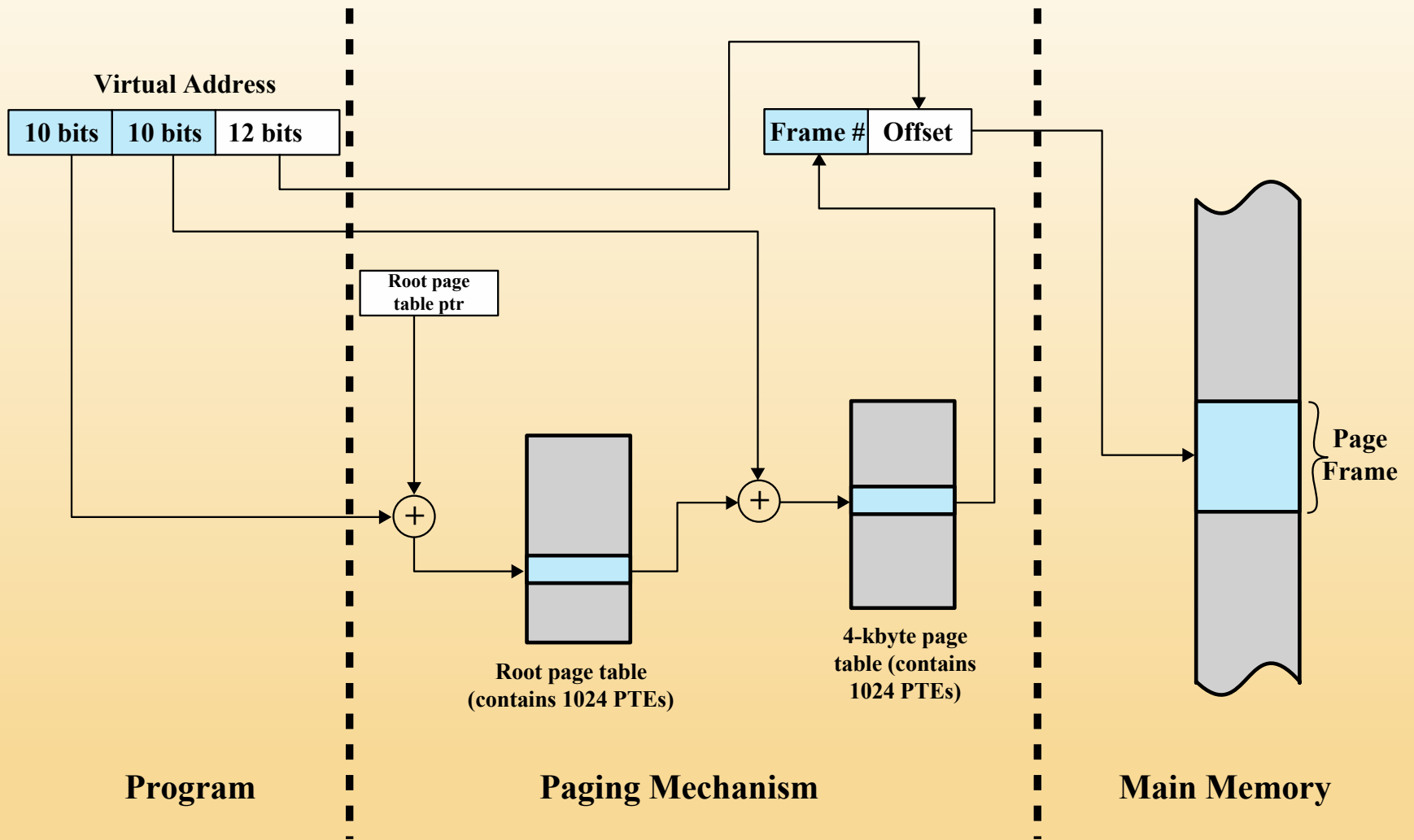


Figure 8.2 Address Translation in a Paging System

Figure 8.3 shows an example of a two-level scheme typical for use with a 32-bit address. If we assume byte-level addressing and 4-kbyte ( $2^{12}$ ) pages, then the 4-Gbyte ( $2^{32}$ ) virtual address space is composed of  $2^{20}$  pages. If each of these pages is mapped by a 4-byte page table entry, we can create a user page table composed of  $2^{20}$  PTEs requiring 4 Mbytes ( $2^{22}$ ). This huge user page table, occupying  $2^{10}$  pages, can be kept in virtual memory and mapped by a root page table with  $2^{10}$  PTEs occupying 4 Kbyte ( $2^{12}$ ) of main memory.



**Figure 8.3 A Two-Level Hierarchical Page Table**



**Figure 8.4 Address Translation in a Two-Level Paging System**



# Inverted Page Table

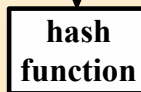
- Page number portion of a virtual address is mapped into a hash value
  - Hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported
- Structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number

Virtual Address

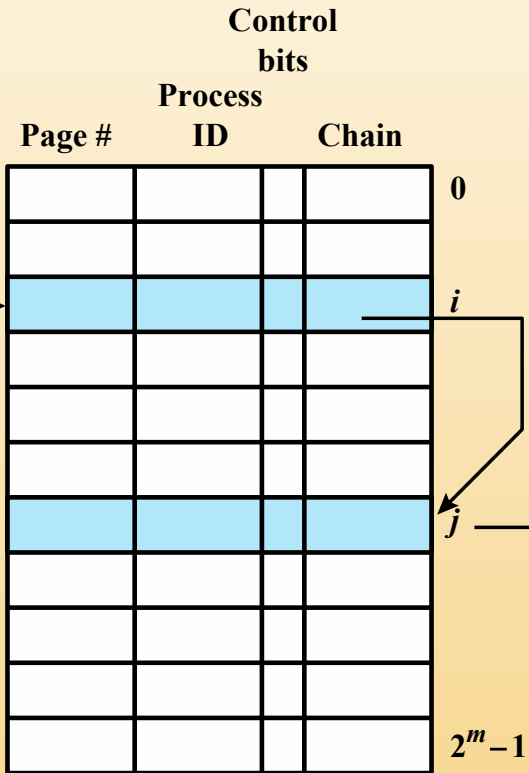
$n$  bits



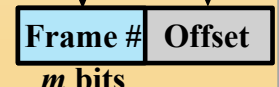
$n$  bits



$m$  bits



Inverted Page Table  
(one entry for each  
physical memory frame)



$m$  bits

Real Address

Figure 8.5 shows a typical implementation of the inverted page table approach. For a physical memory size of  $2^m$  frames, the inverted page table contains  $2^m$  entries, so that the  $i$ th entry refers to frame  $i$ .

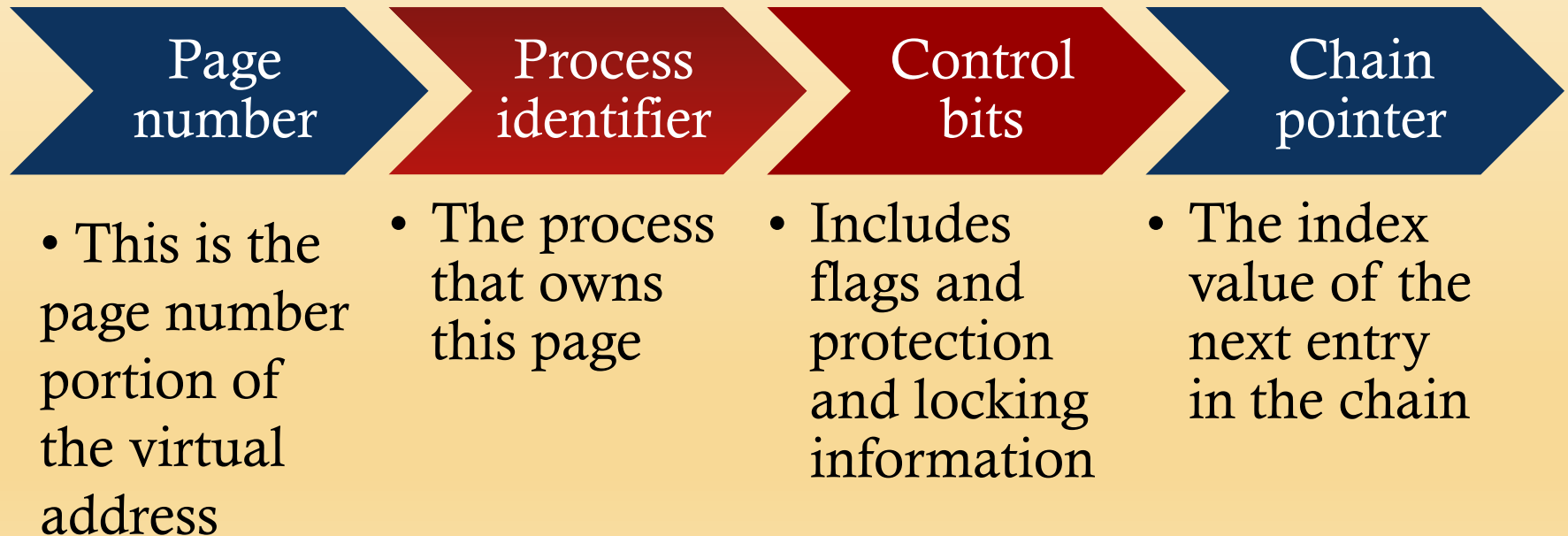
In this example, the virtual address includes an  $n$ -bit page number, with  $n > m$ .

The hash function maps the  $n$ -bit page number into an  $m$ -bit quantity, which is used to index into the inverted page table.

Figure 8.5 Inverted Page Table Structure

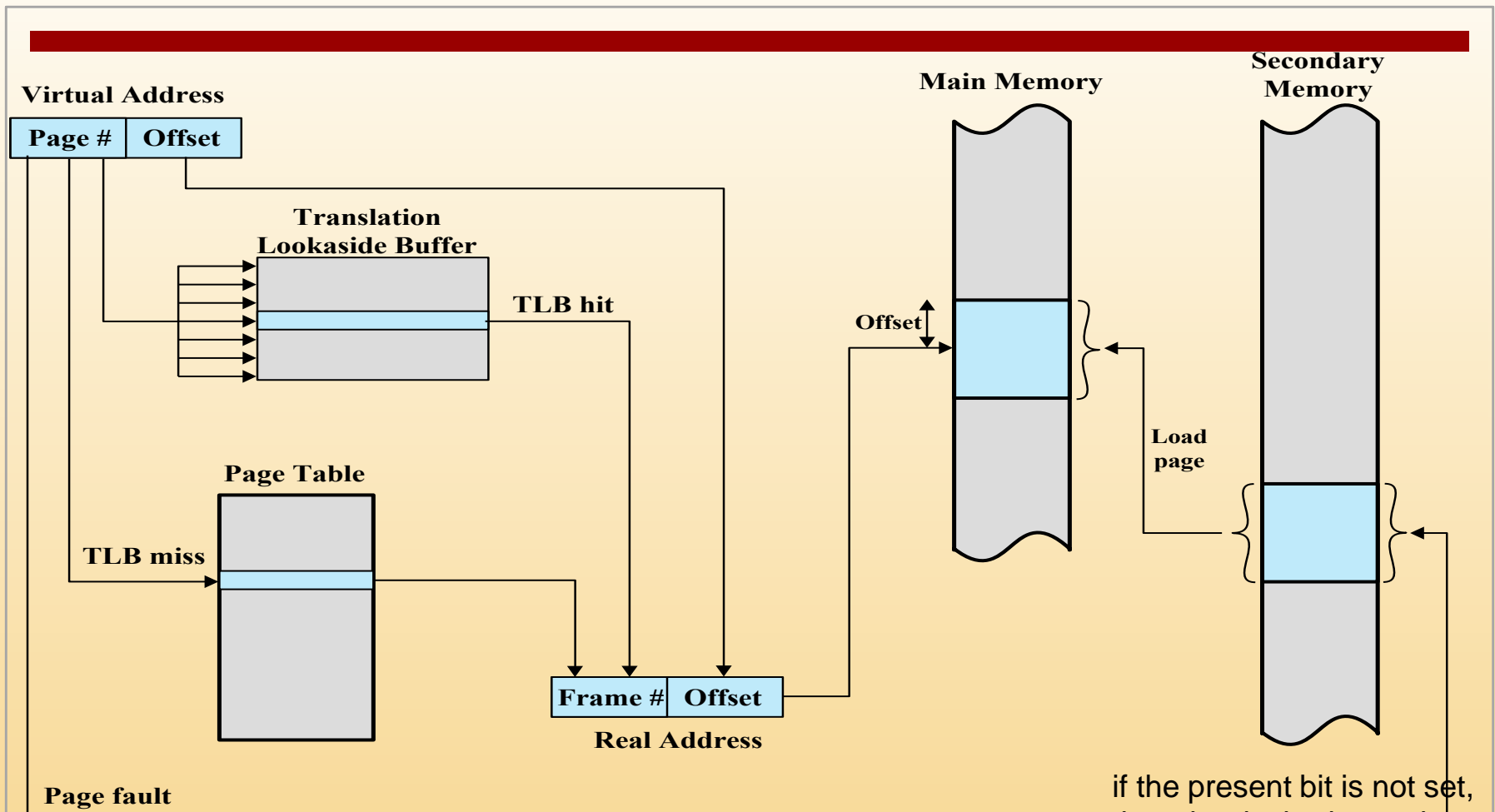
# Inverted Page Table

Each entry in the page table includes:



# Translation Lookaside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses:
  - One to fetch the page table entry
  - One to fetch the data
- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a *translation lookaside buffer* (TLB)
  - This cache functions in the same way as a memory cache and contains those page table entities that have been most recently used



**Figure 8.6 Use of a Translation Lookaside Buffer**

if the present bit is not set, then the desired page is not in main memory and a memory access fault, called a **page fault**, is issued. At this point, we leave the realm of hardware and invoke the operating system, which loads the needed page and updates the page table.

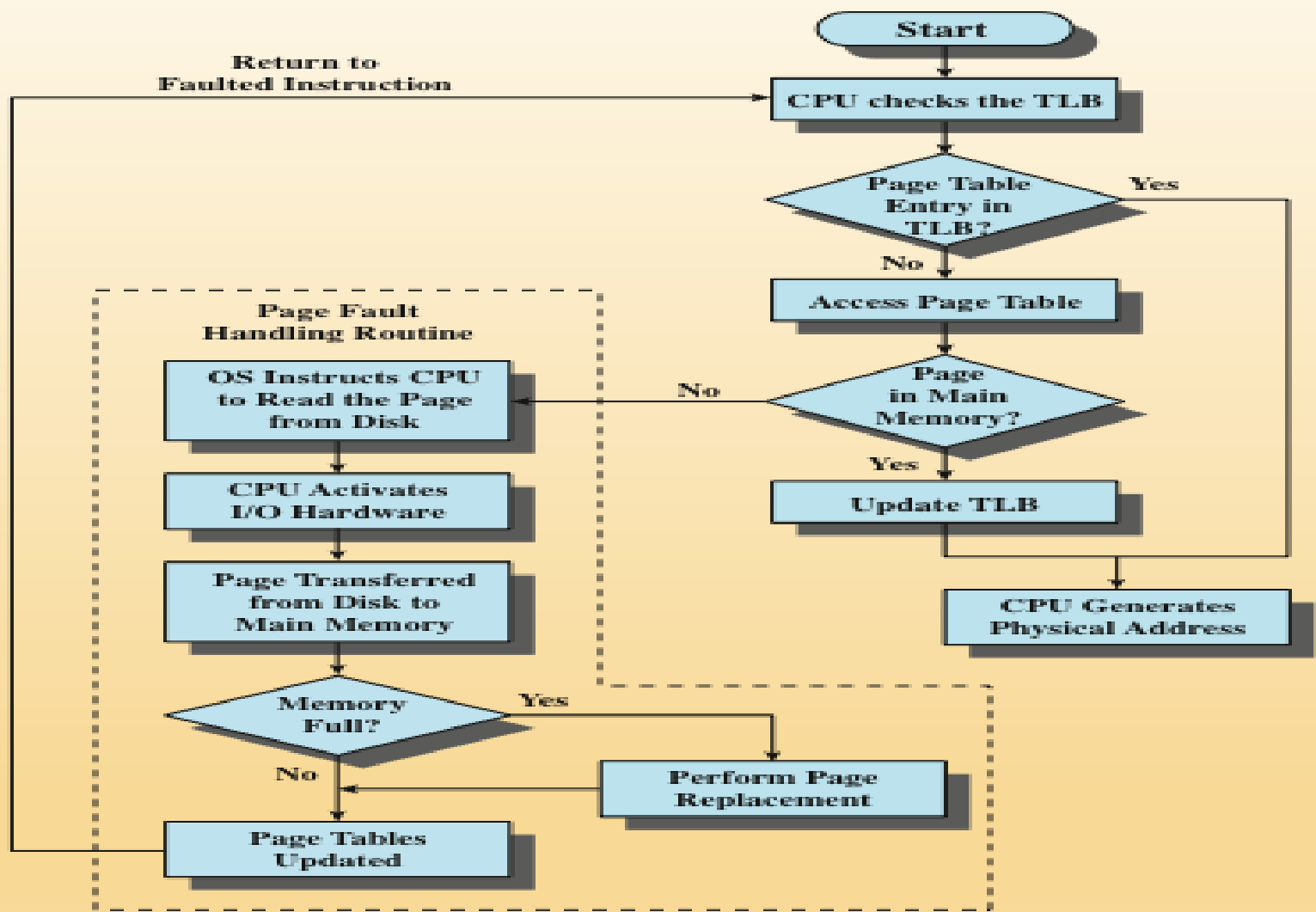


Figure 8.7 Operation of Paging and Translation Lookaside

(TLB) [FURH87]

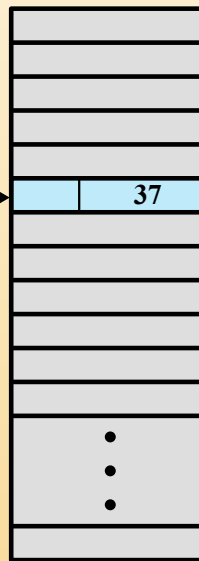
# Associative Mapping

- The TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number
  - Each TLB entry must include the page number as well as the complete page table entry
- The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number

Virtual Address

Page # Offset

5	502
---	-----



Page Table

37	502
----	-----

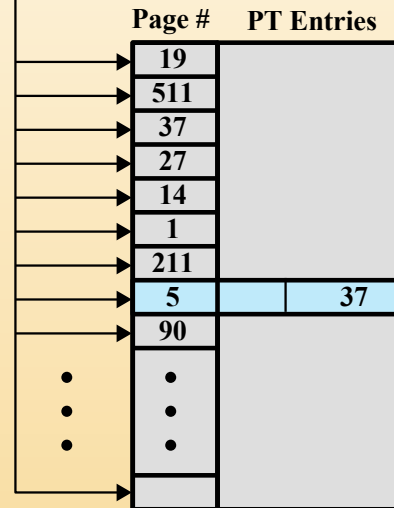
Frame # Offset  
Real Address

(a) Direct mapping

Virtual Address

Page # Offset

5	502
---	-----



Translation Lookaside Buffer

37	502
----	-----

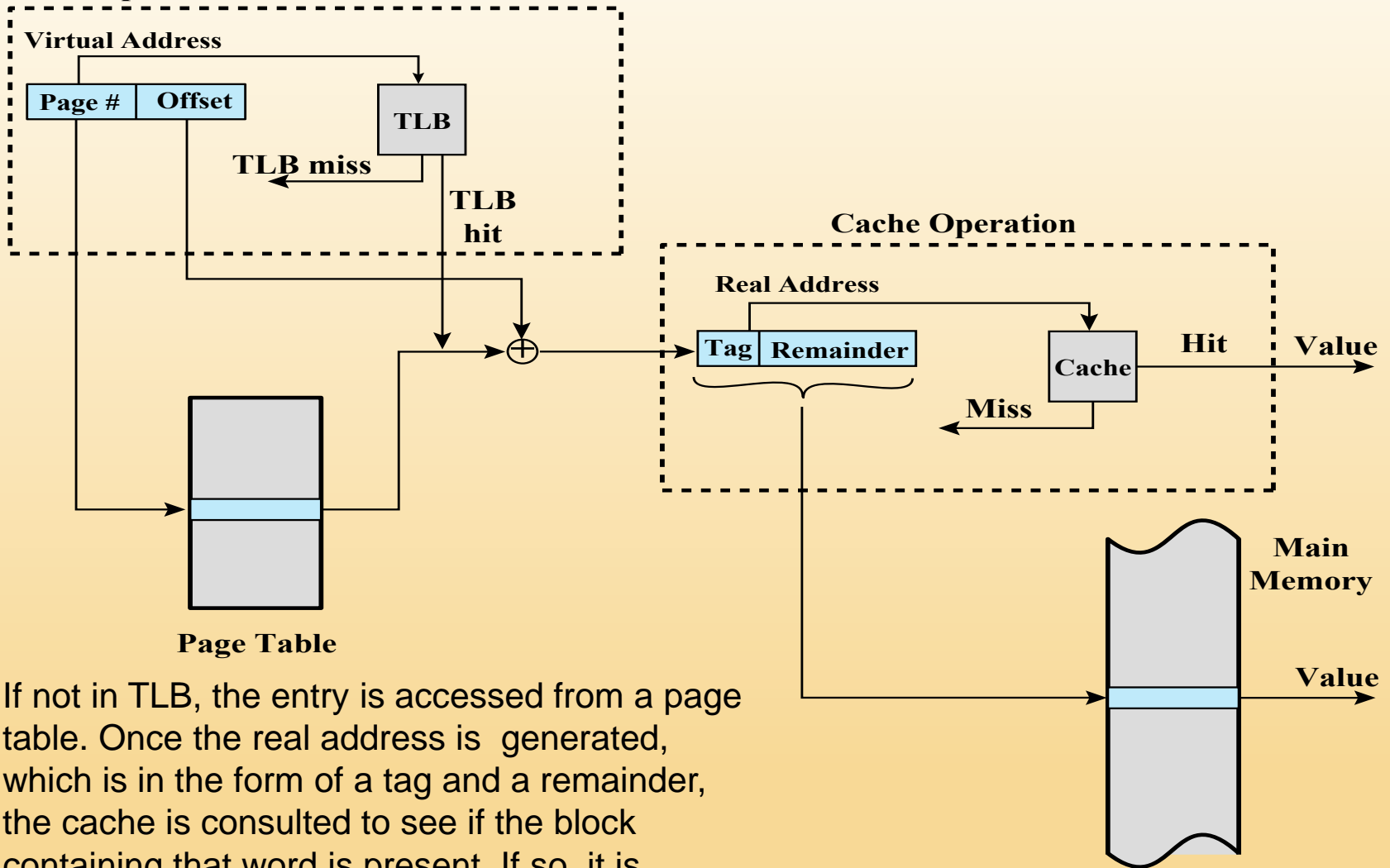
Frame # Offset  
Real Address

(b) Associative mapping

Figure 8.8 Direct Versus Associative Lookup for Page Table Entries



## TLB Operation

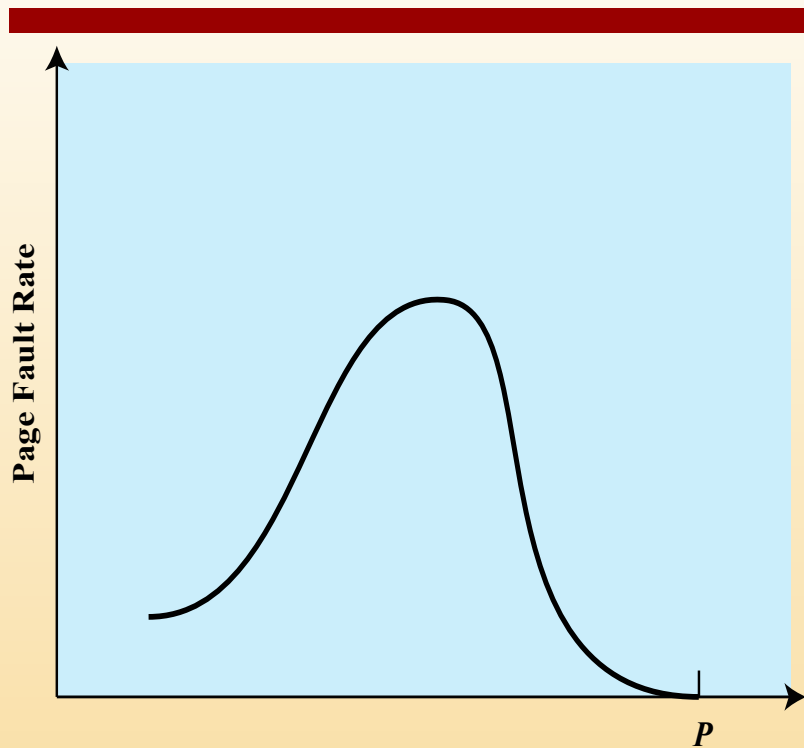


If not in TLB, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present. If so, it is returned to the CPU. If not, the word is retrieved from main memory.

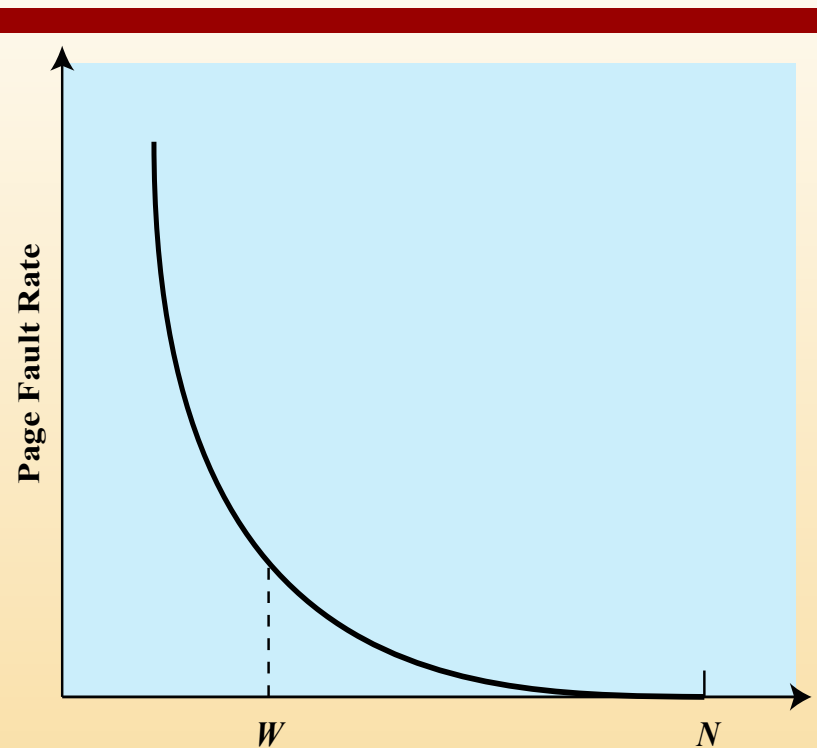
**Figure 8.9 Translation Lookaside Buffer and Cache Operation**

# Page Size

- The **smaller the page size**, the **lesser** the amount of internal fragmentation
  - However, more pages are required per process
  - **More pages** per process means **larger page tables**
  - For large programs in a heavily multiprogrammed environment some portion of the page tables of active processes must be in virtual memory instead of main memory
  - The physical characteristics of most secondary-memory devices favor a larger page size for more efficient block transfer of data



(a) Page Size



(b) Number of Page Frames Allocated

$P$  = size of entire process

$W$  = working set size

$N$  = total number of pages in process

**Figure 8.10 Typical Paging Behavior of a Program**

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Table 8.3

Example  
of Page  
Sizes

# Page Size

The design issue of page size is related to the size of physical main memory and program size



Main memory is getting larger and address space used by applications is also growing



Most obvious on personal computers where applications are becoming increasingly complex

- Contemporary programming techniques used in large programs tend to decrease the locality of references within a process

# Segmentation

- Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments

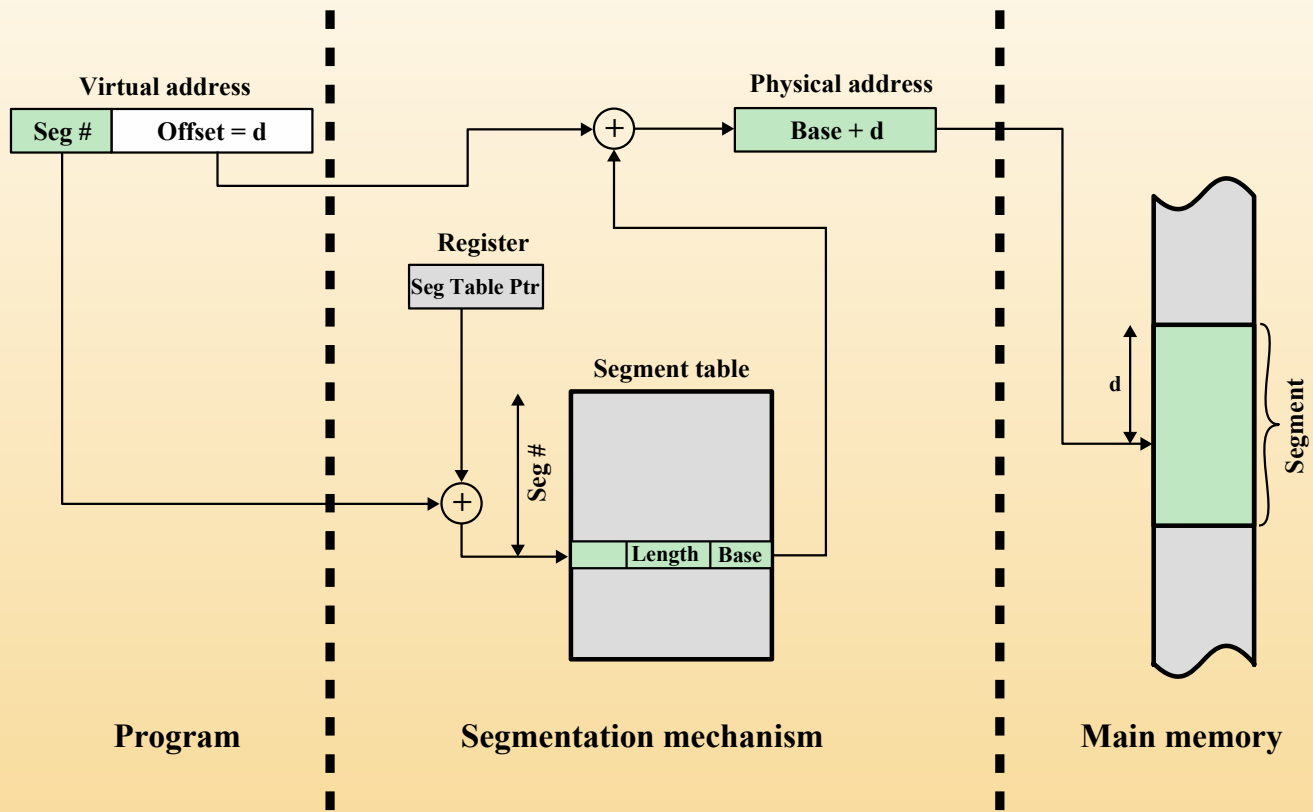
---

## Advantages:

- Simplifies handling of growing data structures
- Allows programs to be altered and recompiled independently
- Lends itself to sharing data among processes
- Lends itself to protection

# Segment Organization

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory



**Figure 8.11 Address Translation in a Segmentation System**

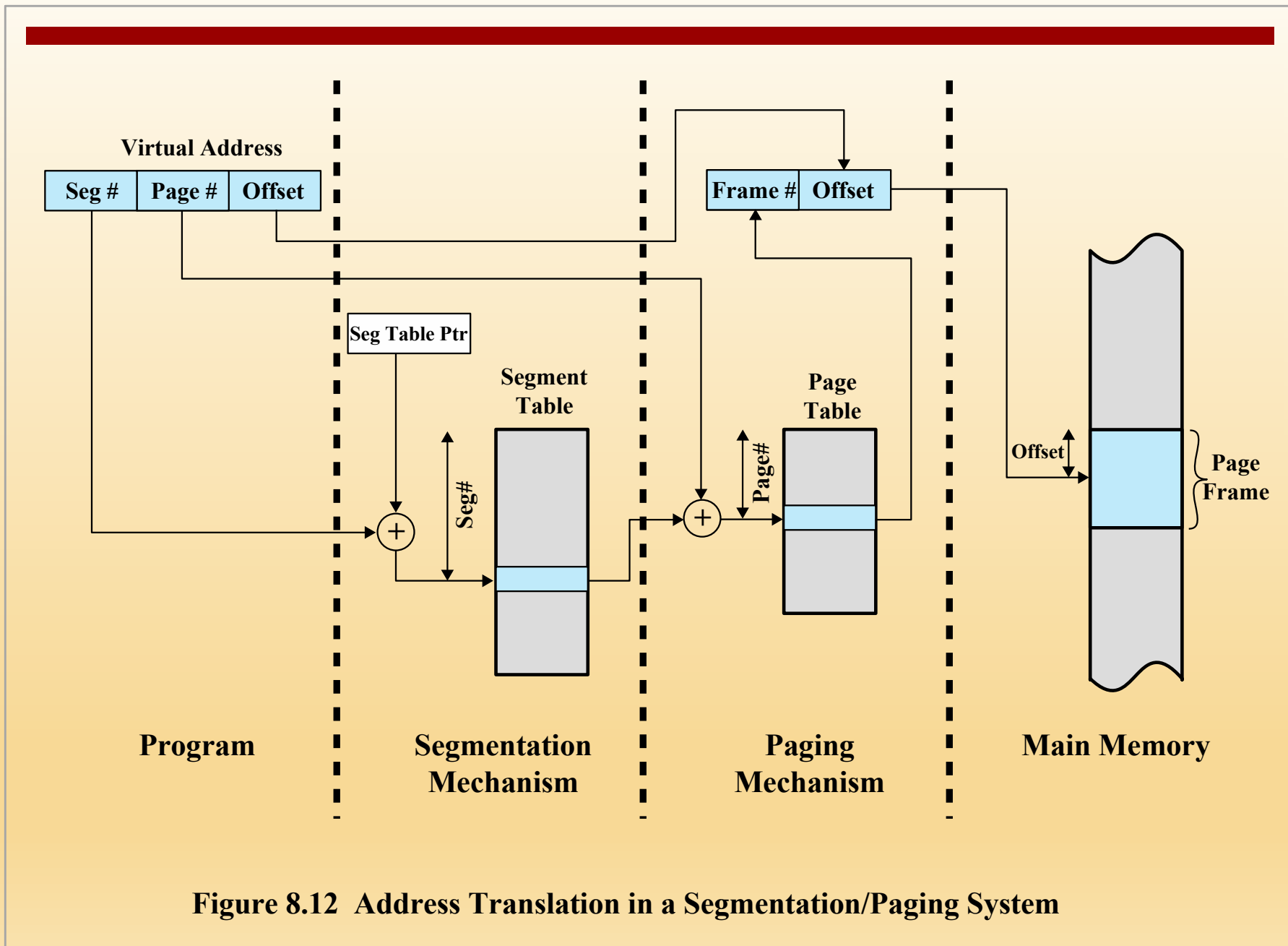


# Combined Paging and Segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

Segmentation is visible to the programmer

Paging is transparent to the programmer



**Figure 8.12 Address Translation in a Segmentation/Paging System**

Virtual Address



Segment Table Entry



Page Table Entry



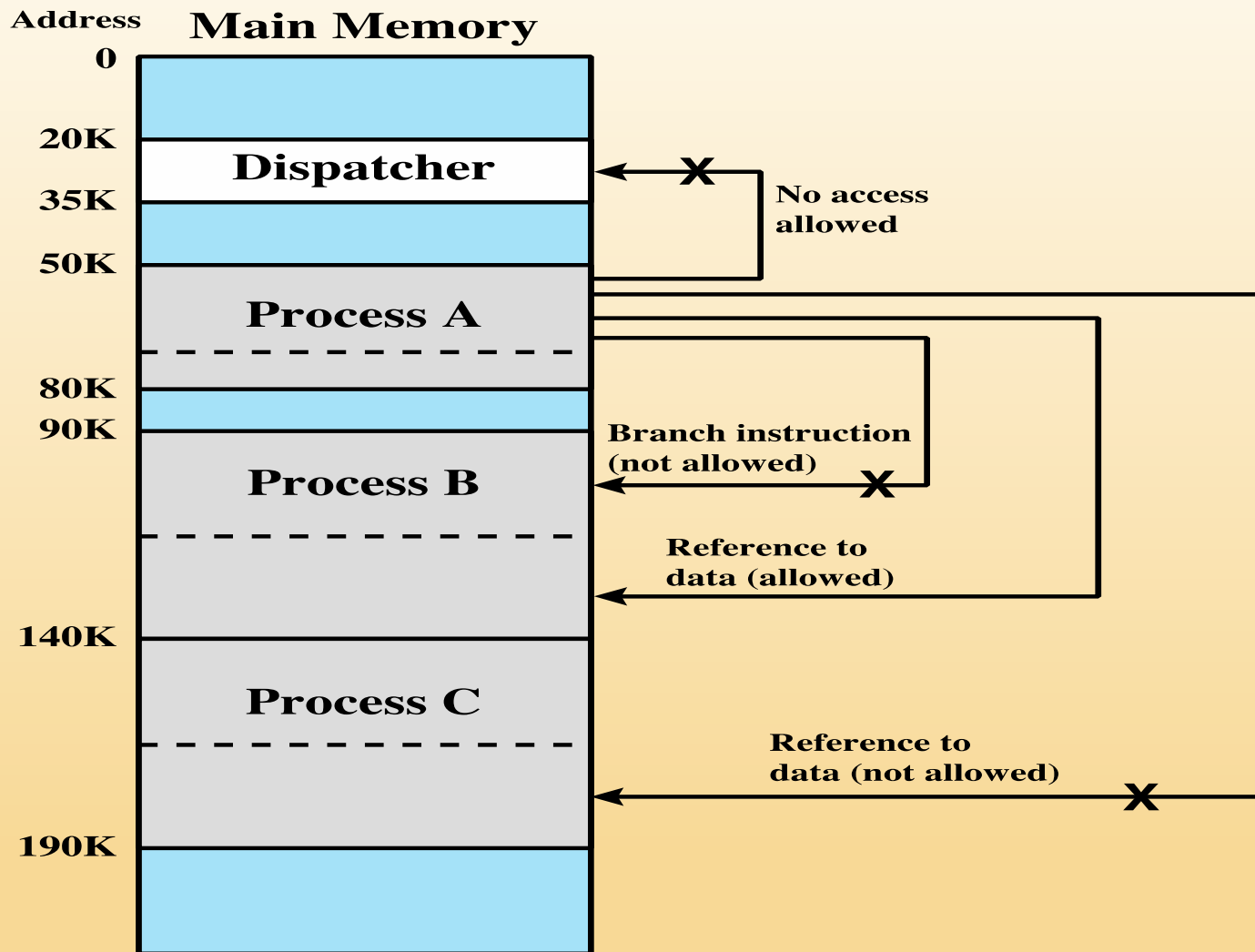
P= present bit  
M = Modified bit

**(c) Combined segmentation and paging**

Figure 8.1 Typical Memory Management Formats

# Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes



**Figure 8.13 Protection Relationships Between Segments**

# Operating System Software

The design of the memory management portion of an operating system depends on three fundamental areas of choice:

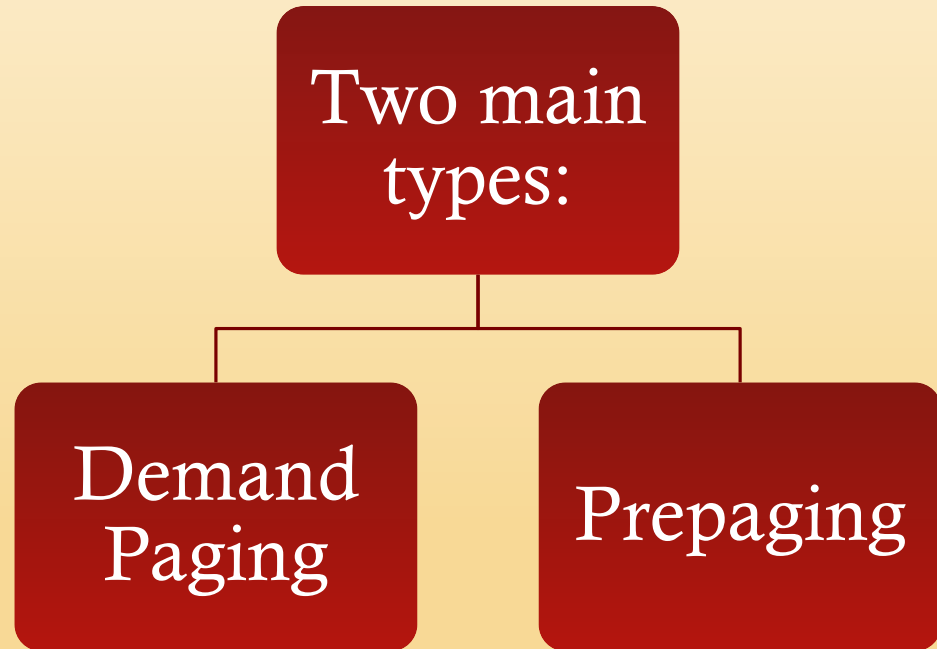
- Whether or not to **use virtual memory** techniques
- The use **of paging or segmentation** or both
- The algorithms employed for various aspects of memory management

<p><b>Fetch Policy</b></p> <ul style="list-style-type: none"> <li>Demand paging</li> <li>Prepaging</li> </ul> <p><b>Placement Policy</b></p> <p><b>Replacement Policy</b></p> <ul style="list-style-type: none"> <li>Basic Algorithms <ul style="list-style-type: none"> <li>Optimal</li> <li>Least recently used (LRU)</li> <li>First-in-first-out (FIFO)</li> <li>Clock</li> </ul> </li> <li>Page Buffering</li> </ul>	<p><b>Resident Set Management</b></p> <ul style="list-style-type: none"> <li>Resident set size <ul style="list-style-type: none"> <li>Fixed</li> <li>Variable</li> </ul> </li> <li>Replacement Scope <ul style="list-style-type: none"> <li>Global</li> <li>Local</li> </ul> </li> </ul> <p><b>Cleaning Policy</b></p> <ul style="list-style-type: none"> <li>Demand</li> <li>Precleaning</li> </ul> <p><b>Load Control</b></p> <ul style="list-style-type: none"> <li>Degree of multiprogramming</li> </ul>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Table 8.4 Operating System Policies for Virtual Memory**

# Fetch Policy

- Determines when a page should be brought into memory





# Demand Paging

## ■ Demand Paging

- Only **brings** pages into main memory **when a reference is made** to a location on the page
- **Many page faults** when process is **first started**
- **Principle of locality** suggests that as **more and more pages** are brought in, most future references will be to pages that have recently been brought in, and **page faults should drop to a very low level**

# Prepaging

## ■ Prepaging

- Pages other than the one demanded by a page fault are brought in
- Exploits the characteristics of most secondary memory devices
- If pages of a process are **stored contiguously** in secondary memory it is more efficient to **bring in a number of pages at one time**
- Ineffective if extra pages are not referenced
- Should not be confused with “swapping”

# Placement Policy

- Determines where in real memory a process piece is to reside
- Important design issue in a segmentation system
- Paging or combined paging with segmentation placing is irrelevant because hardware performs functions with equal efficiency
- For NUMA systems an automatic placement strategy is desirable

# Replacement Policy

- Deals with the **selection of a page in main memory to be replaced** when a new page must be brought in
  - Objective is that the page that is removed be **the page least likely to be referenced in the near future**
- The more elaborate the replacement policy the greater the hardware and software overhead to implement it

# Frame Locking

- When a frame is locked the page currently stored in that frame may not be replaced
  - Kernel of the OS as well as key control structures are held in locked frames
  - I/O buffers and time-critical areas may be locked into main memory frames
  - Locking is achieved by associating a lock bit with each frame

# Basic Algorithms



Algorithms used for the selection of a page to replace:

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock

# Optimal Policy

- The **optimal policy** selects for replacement that page for which **the time to the next reference is the longest**.
- It can be shown that this policy results in the fewest number of page faults [BELA66].
- Clearly, this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events.
- However, it does serve as a standard against which to judge real-world algorithms.

# Least Recently Used (LRU)

- Replaces the page that **has not been referenced for the longest time**
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Difficult to implement
  - One approach is to tag each page with the time of last reference
    - This requires a great deal of overhead

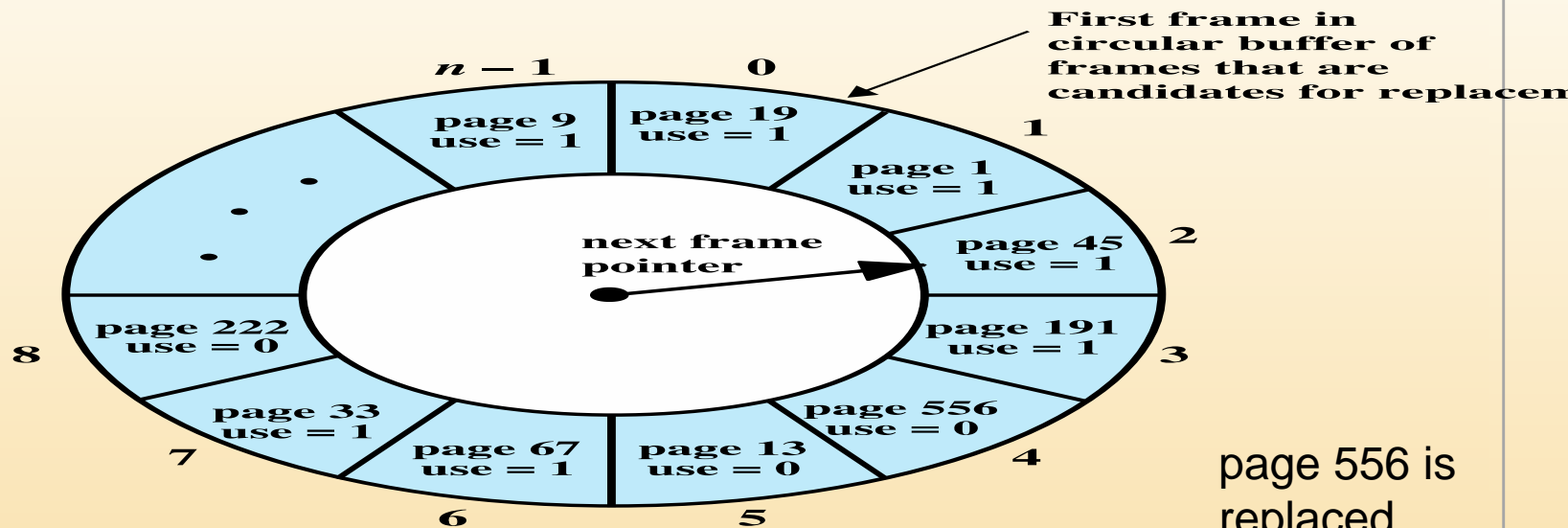


# First-in-First-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Simple replacement policy to implement
- Page that has been in memory the longest is replaced

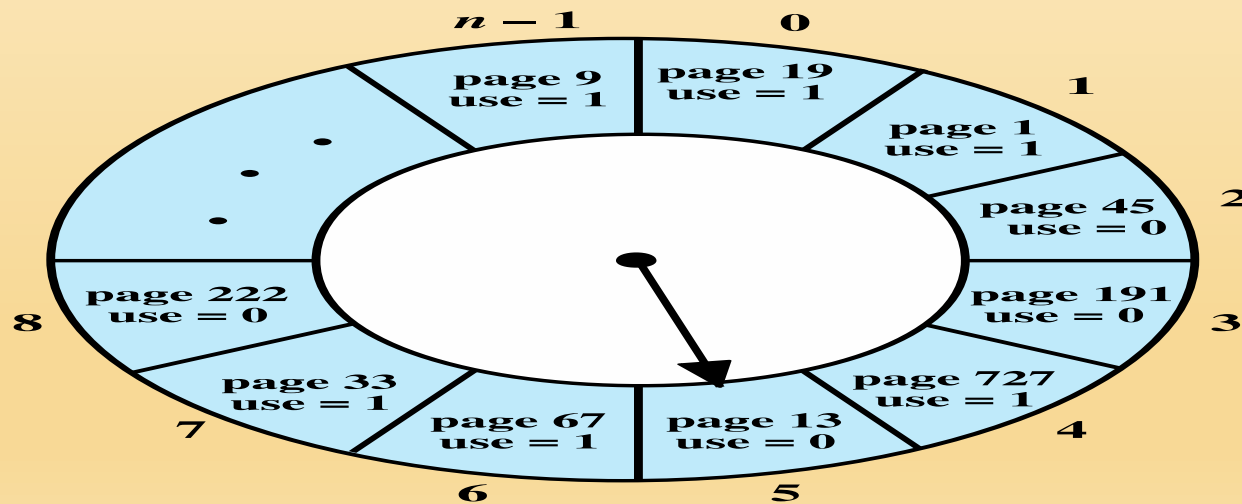
# Clock Policy

- Requires the association of **an additional bit** with each frame
  - Referred to as **the use bit**
- When **a page is first loaded** in memory or referenced, the **use bit is set to 1**. Whenever the page is **subsequently referenced** (after the reference that generated the page fault), its use bit is **set to 1**
- The set of frames is considered to be a circular buffer. Page frames visualized as laid out in a circle.
- When it comes time to replace a page, the operating system **scans the buffer to find a frame with a use bit set to 0**. Each time it encounters a frame with a use bit of 1, it resets that bit to 0 and continues on. If any of the frames in the buffer have a use bit of 0 at the beginning of this process, **the first such frame (with use bit 0) encountered is chosen for replacement**.



(a) State of buffer just prior to a page replacement

page 556 is replaced with page 727



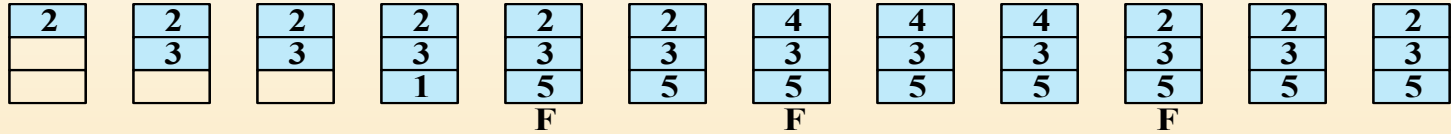
(b) State of buffer just after the next page replacement

**Figure 8.15 Example of Clock Policy Operation**

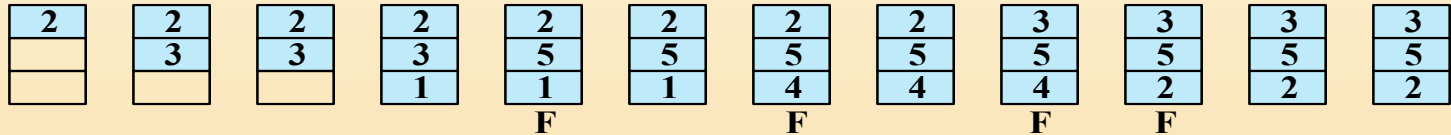
Page address stream

2 3 2 1 5 2 4 5 3 2 5 2

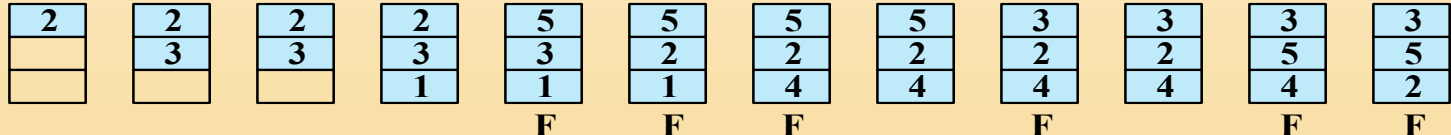
OPT



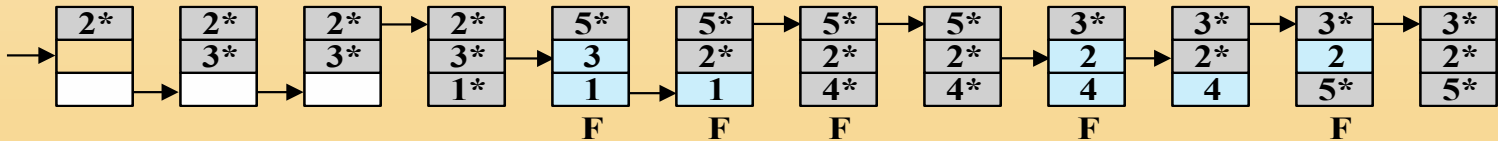
LRU



FIFO



CLOCK



F = page fault occurring after the frame allocation is initially filled

Figure 8.14 Behavior of Four Page-Replacement Algorithms

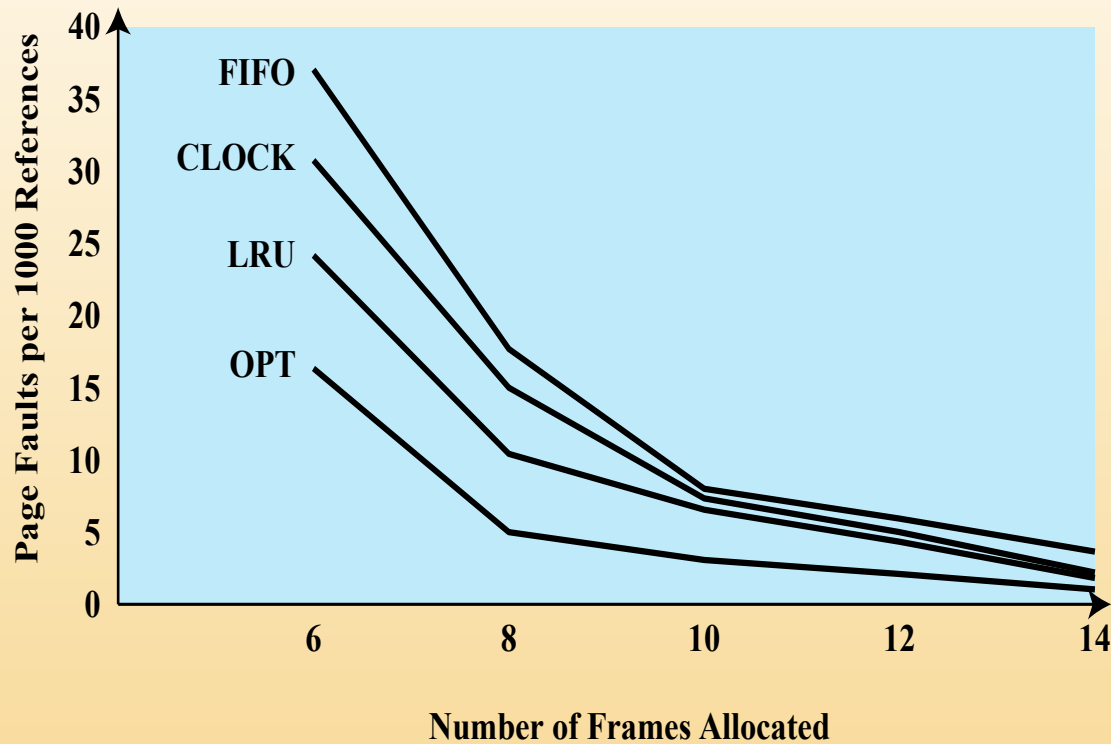
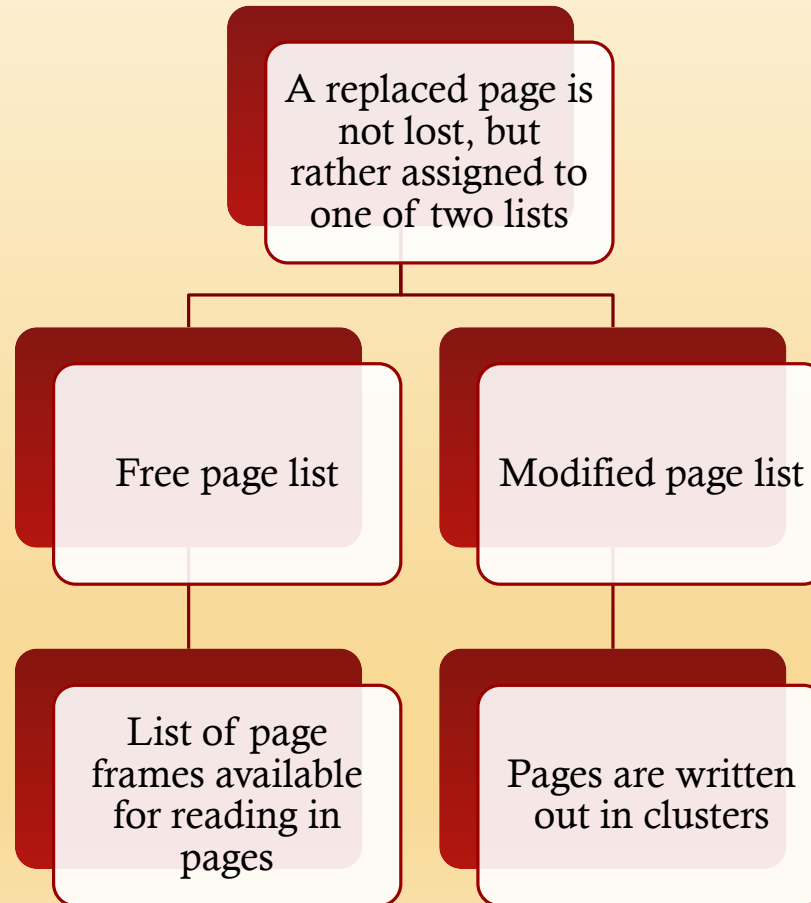


Figure 8.16 shows the results of an experiment reported in [BAER80], which compares the four algorithms that we have been discussing; it is assumed that the number of page frames assigned to a process is fixed. The results are based on the execution of  $0.25 \times 10^6$  references in a FORTRAN program, using a page size of 256 words. Baer ran the experiment with frame allocations of 6, 8, 10, 12, and 14 frames.

**Figure 8.16 Comparison of Fixed-Allocation, Local Page Replacement Algorithms**

# Page Buffering

- Improves paging performance and allows the use of a simpler page replacement policy



# Replacement Policy and Cache Size

- With large caches, replacement of pages can have a performance impact
  - If the page frame selected for replacement is in the cache, that cache block is lost as well as the page that it holds
  - In systems using page buffering, cache performance can be improved with a policy for page placement in the page buffer
  - Most operating systems place pages by selecting an arbitrary page frame from the page buffer

# Resident Set Management

- The OS must decide **how many pages** to bring into main memory
  - The **smaller the amount** of memory allocated **to each process**, the **more processes can reside** in memory
  - **Small number of pages loaded increases page faults**
  - **Beyond a certain size**, further allocations of pages will **not** noticeable effect on the page fault rate



# Resident Set Size

## Fixed-allocation

- Gives a process a fixed number of frames in main memory within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced

## Variable-allocation

- Allows the number of page frames allocated to a process to be varied over the lifetime of the process

# Replacement Scope

- The scope of a replacement strategy can be categorized as *global* or *local*
  - Both types are activated by a page fault when there are no free page frames

## Local

- Chooses only among the resident pages of the process that generated the page fault

## Global

- Considers all unlocked pages in main memory

### Local Replacement

### Global Replacement

#### Fixed Allocation

- Number of frames allocated to a process is fixed.
- Page to be replaced is chosen from among the frames allocated to that process.

- Not possible.

#### Variable Allocation

- The number of frames allocated to a process may be changed from time to time to maintain the working set of the process.
- Page to be replaced is chosen from among the frames allocated to that process.

- Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

**Table 8.5 Resident Set Management**

# Fixed Allocation, Local Scope

- Necessary to decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate

If allocation is too large, there will be too few programs in main memory

- Increased processor idle time
- Increased time spent in swapping

# Variable Allocation

## Global Scope

- Easiest to implement
  - Adopted in a number of operating systems
- OS maintains a list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no frames are available the OS must choose a page currently in memory
- One way to counter potential problems is to use page buffering

# Variable Allocation

## Local Scope

- When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set
- When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault
- Reevaluate the allocation provided to the process and increase or decrease it to improve overall performance

# Variable Allocation

## Local Scope

- Decision to increase or decrease a resident set size is based on the assessment of the likely future demands of active processes

### Key elements:

- Criteria used to determine resident set size
- The timing of changes

# Working Set Strategy

- The key elements of the variable-allocation, local-scope strategy are the criteria used to determine resident set size and the timing of changes.
- One specific strategy that has received much attention in the literature is known as the **working set strategy** .
- Although a true working set strategy would be **difficult to implement**, it is useful to examine it as a baseline for comparison.
- The working set is a concept introduced and popularized by Denning [DENN68, DENN70, DENN80b]; it has had a profound impact on virtual memory management design.
- The working set with parameter  $\Delta$  for a process at virtual time  $t$ , which we designate as  $W(t, \Delta)$ , is *the set of pages of that process that have* been referenced in the last  $\Delta$  virtual time units.

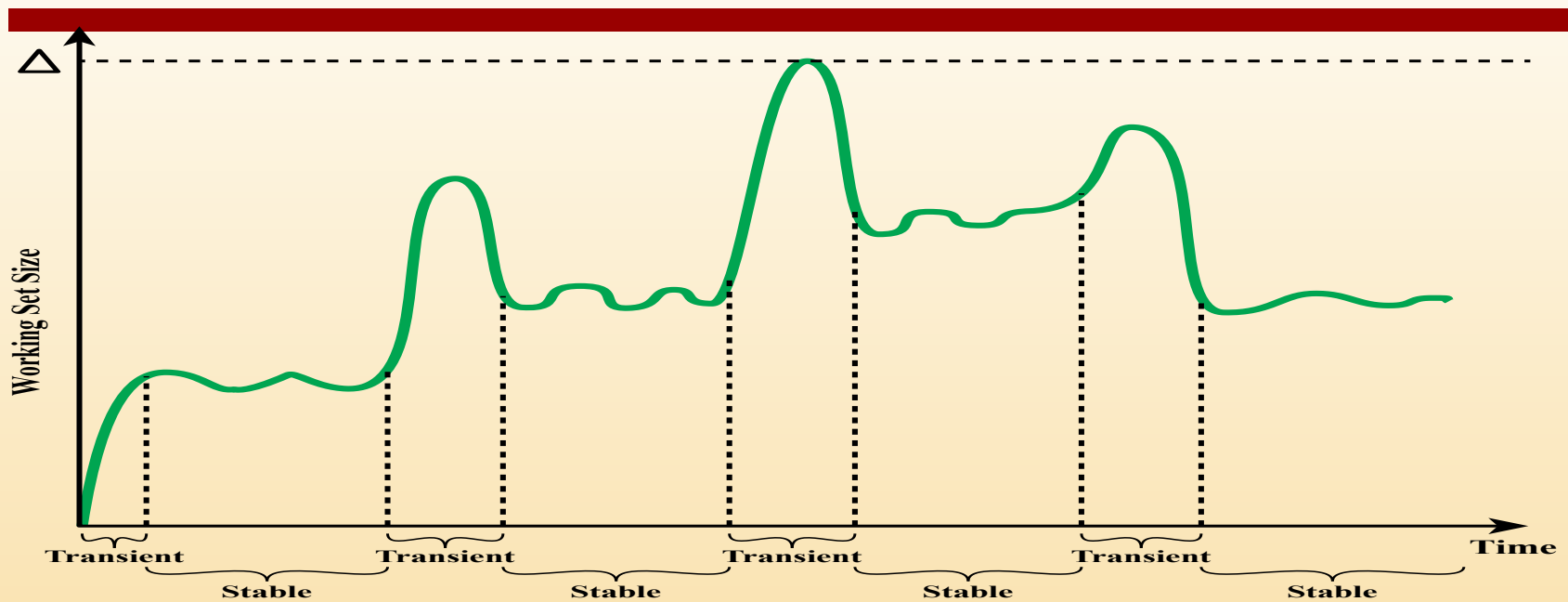


**Sequence of  
Page  
References**

**Window Size, D**

	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

**Figure 8.17 Working Set of Process as Defined by Window Size**



**Figure 8.18 Typical Graph of Working Set Size [MAEK87]**

Figure 8.18 indicates the way in which the working set size can vary over time for a fixed value of  $\Delta$ . For many programs, periods of relatively stable working set sizes alternate with periods of rapid change. When a process first begins executing, it gradually builds up to a working set as it references new pages. Eventually, by the principle of locality, the process should stabilize on a certain set of pages. Subsequent transient periods reflect a shift of the program to a new locality. During the transition phase, some of the pages from the old locality remain within the window,  $\Delta$ , causing a surge in the size of the working set as new pages are referenced. As the window slides past these page references, the working set size declines until it contains only those pages from the new locality.

# Page Fault Frequency (PFF)

- An algorithm that follows working set strategy is the page fault frequency (PFF) algorithm [CHU72, GUPT78]. It requires a use bit to be associated with each page in memory
- Requires a use bit to be associated with each page in memory
- Bit is set to 1 when that page is accessed
- When a page fault occurs, the OS notes the virtual time since the last page fault for that process.
- A threshold  $F$  is defined. If the amount of time since the last page fault is less than  $F$ , then a page is added to the resident set of the process. Otherwise, discard all pages with a use bit of 0, and shrink the resident set accordingly
- Does **not perform well during the transient periods** when there is a shift to a new locality

# Variable-Interval Sampled Working Set (VSWS)

- An approach that attempts to deal with the phenomenon of interlocality transition with a similar relatively low overhead to that of PFF is the variable-interval sampled working set (VSWS) policy [FERR83].
- Evaluates the working set of a process at sampling instances based on elapsed virtual time
- Driven by three parameters:

The minimum duration of the sampling interval

The maximum duration of the sampling interval

The number of page faults that are allowed to occur between sampling instances

# Cleaning Policy

- Concerned with determining when a modified page should be written out to secondary memory

## Demand Cleaning

A page is written out to secondary memory only when it has been selected for replacement

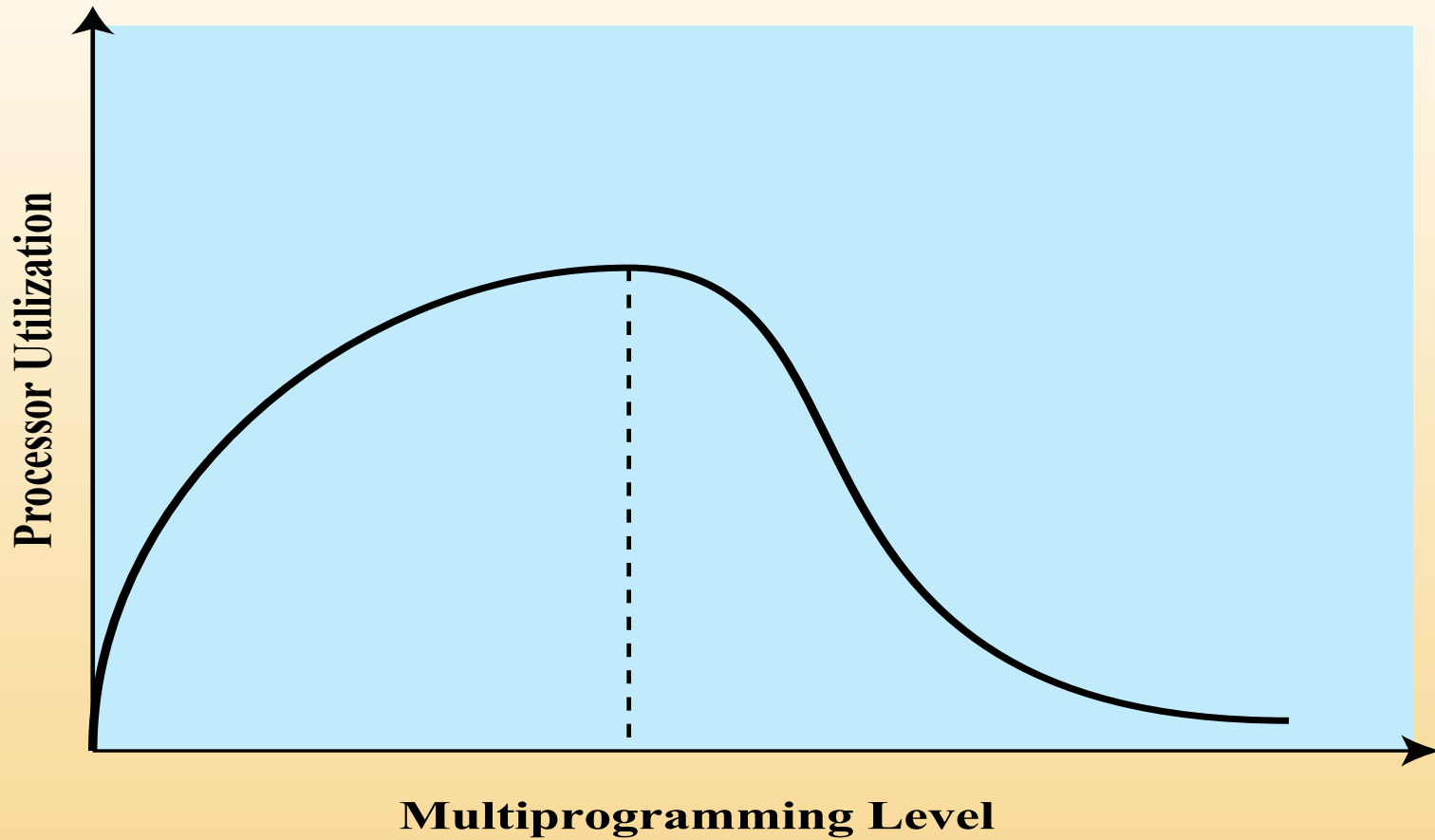


## Precleaning

Allows the writing of pages in batches

# Load Control

- Load control is concerned with determining the number of processes that will be resident in main memory, which has been referred to as the **multiprogramming level**. The load control policy is critical in effective memory management
- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping
- Too many processes will lead to thrashing



**Figure 8.19 Multiprogramming Effects**

# Process Suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out

Six possibilities exist:

- Lowest-priority process
- Faulting process
- Last process activated
- Process with the smallest resident set
- Largest process
- Process with the largest remaining execution window



# UNIX

- Intended to be machine independent so its memory management schemes will vary
  - Early UNIX: variable partitioning with no virtual memory scheme
  - Current implementations of UNIX and Solaris make use of paged

virtual memory  
**SVR4 and Solaris use  
two separate schemes:**

- Paging system
- Kernel memory allocator

# Paging System and Kernel Memory Allocator

## Paging System

Provides a virtual memory capability that allocates page frames in main memory to processes

Allocates page frames to disk block buffers

## Kernel Memory Allocator

Allocates memory for the kernel

Page frame number	Age	Copy on write	Modify	Reference	Valid	Protect
-------------------	-----	---------------	--------	-----------	-------	---------

(a) Page table entry

Swap device number	Device block number	Type of storage
--------------------	---------------------	-----------------

(b) Disk block descriptor

Page state	Reference count	Logical device	Block number	Pfdata pointer
------------	-----------------	----------------	--------------	----------------

(c) Page frame data table entry

Reference count	Page/storage unit number
-----------------	--------------------------

(d) Swap-use table entry

**Figure 8.20 UNIX SVR4 Memory Management Formats**

### Page Table Entry

**Page frame number**

Refers to frame in real memory.

**Age**

Indicates how long the page has been in memory without being referenced. The length and contents of this field are processor dependent.

**Copy on write**

Set when more than one process shares a page. If one of the processes writes into the page, a separate copy of the page must first be made for all other processes that share the page. This feature allows the copy operation to be deferred until necessary and avoided in cases where it turns out not to be necessary.

**Modify**

Indicates page has been modified.

**Reference**

Indicates page has been referenced. This bit is set to 0 when the page is first loaded and may be periodically reset by the page replacement algorithm.

**Valid**

Indicates page is in main memory.

**Protect**

Indicates whether write operation is allowed.

### Disk Block Descriptor

**Swap device number**

Logical device number of the secondary device that holds the corresponding page. This allows more than one device to be used for swapping.

**Device block number**

Block location of page on swap device.

**Type of storage**

Storage may be swap unit or executable file. In the latter case, there is an indication as to whether the virtual memory to be allocated should be cleared first.

# Table 8.6

## UNIX SVR4 Memory Management Parameters (page 1 of 2)

(Table can be found on page 381  
in the textbook)

## Page Frame Data Table Entry

### Page state

Indicates whether this frame is available or has an associated page. In the latter case, the status of the page is specified: on swap device, in executable file, or DMA in progress.

### Reference count

Number of processes that reference the page.

### Logical device

Logical device that contains a copy of the page.

### Block number

Block location of the page copy on the logical device.

### Pfdata pointer

Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.

## Swap-Use Table Entry

### Reference count

Number of page table entries that point to a page on the swap device.

### Page/storage unit number

Page identifier on storage unit.

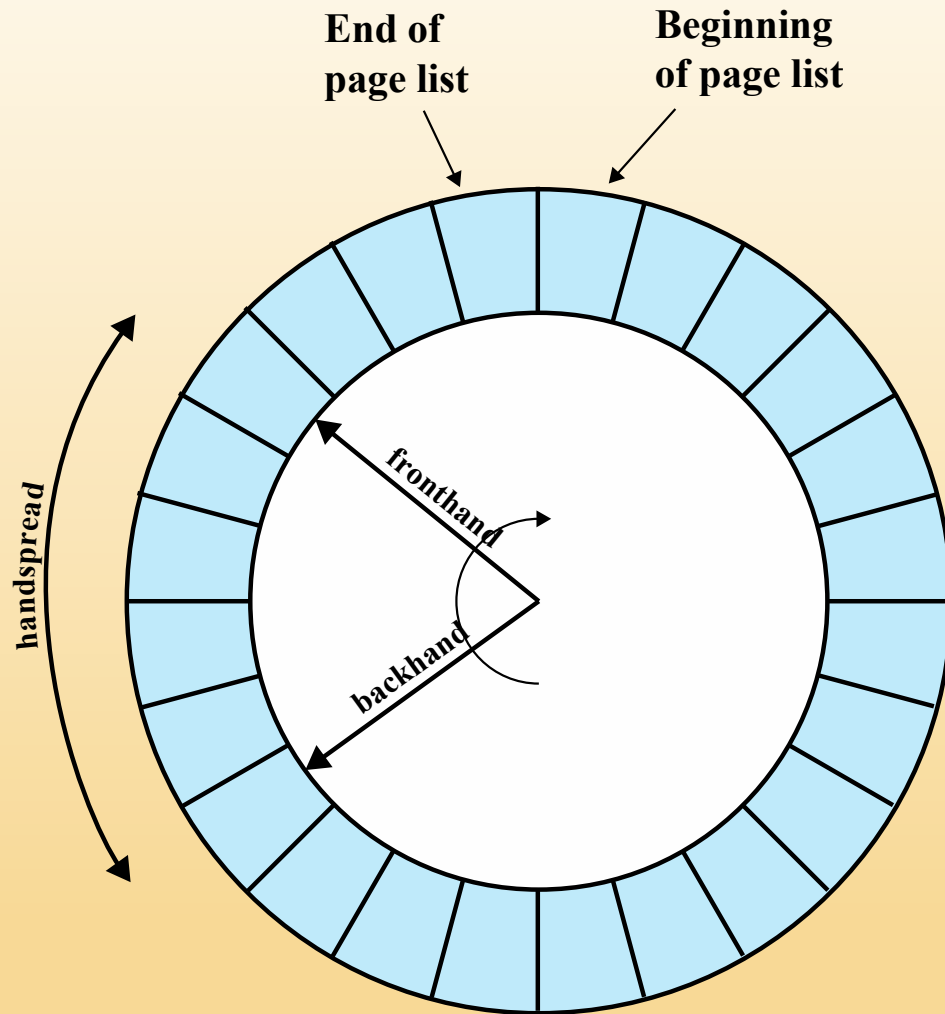
## Table 8.6

### UNIX SVR4 Memory Management Parameters (page 2 of 2)

(Table can be found on page 381 in the textbook)

# Page Replacement

- The page frame data table is used for page replacement
- Pointers are used to create lists within the table
  - All available frames are linked together in a list of free frames available for bringing in pages
  - When the number of available frames drops below a certain threshold, the kernel will steal a number of frames to compensate



**Figure 8.21 Two-Handed Clock Page-Replacement Algorithm**

# Kernel Memory Allocator

- The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation.
- Most of these blocks are significantly smaller than typical pages (therefore paging would be inefficient)
- Allocations and free operations must be made as fast as possible



# Lazy Buddy

- Technique adopted for SVR4
- UNIX often exhibits steady-state behavior in kernel memory demand
  - i.e. the amount of demand for blocks of a particular size varies slowly in time
- Defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible

Initial value of  $D_i$  is 0

After an operation, the value of  $D_i$  is updated as follows

**(I)** if the next operation is a block allocate request:

if there is any free block, select one to allocate

if the selected block is locally free

then  $D_i := D_i + 2$

else  $D_i := D_i + 1$

otherwise

first get two blocks by splitting a larger one into two (recursive operation)

allocate one and mark the other locally free

$D_i$  remains unchanged (but  $D$  may change for other block sizes because of the recursive call)

**(II)** if the next operation is a block free request

Case  $D_i \geq 2$

mark it locally free and free it locally

$D_i := D_i - 2$

Case  $D_i = 1$

mark it globally free and free it globally; coalesce if possible

$D_i := 0$

Case  $D_i = 0$

mark it globally free and free it globally; coalesce if possible

select one locally free block of size  $2^i$  and free it globally; coalesce if possible

$D_i := 0$

**Figure 8.22 Lazy Buddy System Algorithm**

# Linux

## Memory Management

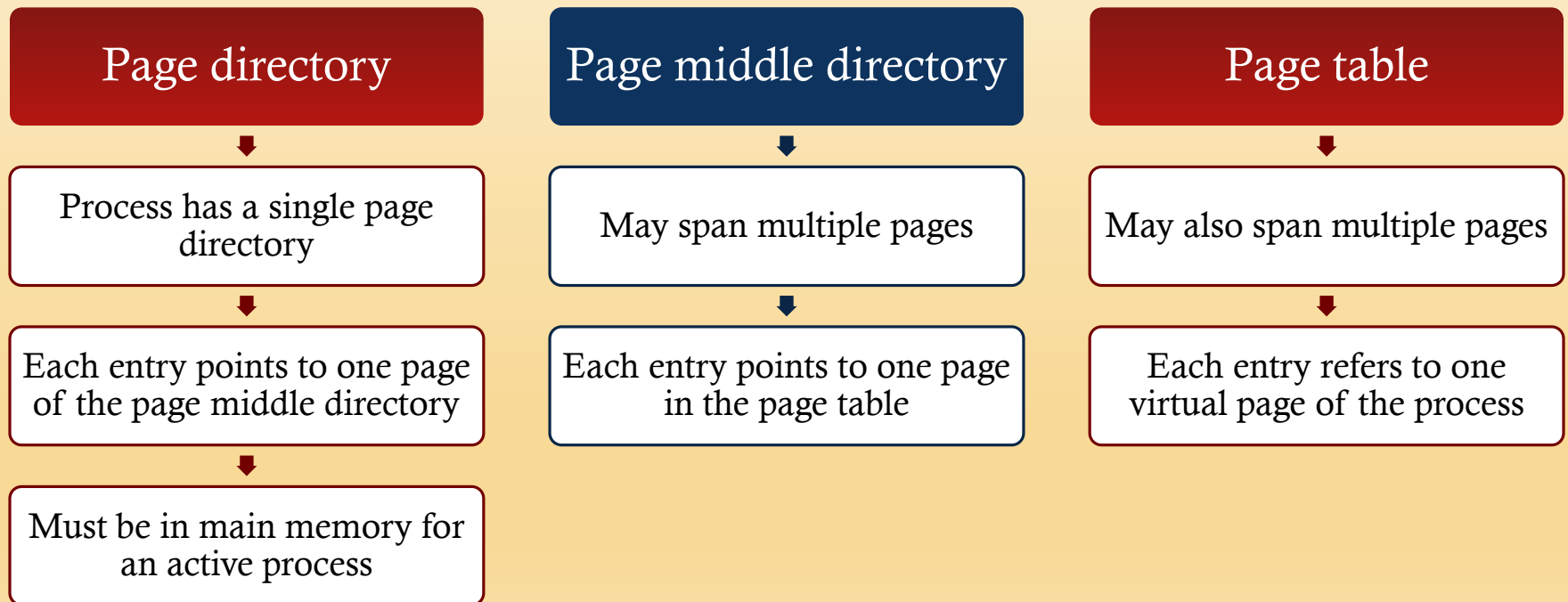
- Shares many characteristics with UNIX
- Is quite complex

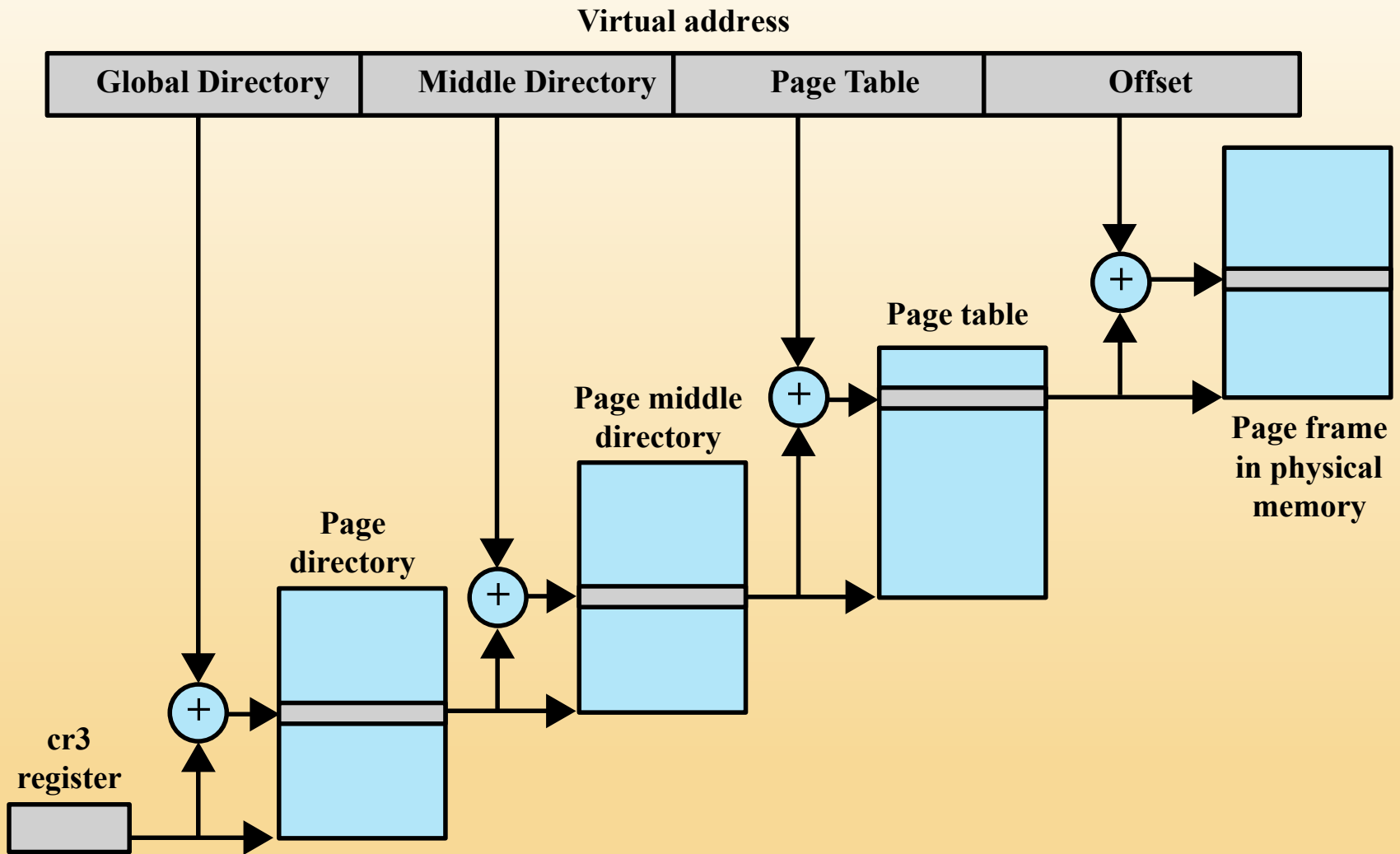
Two main  
aspects

- Process virtual memory
- Kernel memory allocation

# Linux Virtual Memory

## ■ Three level page table structure:





**Figure 8.23 Address Translation in Linux Virtual Memory Scheme**

# Linux Page Replacement

- Based on the clock algorithm
- The use bit is replaced with an 8-bit age variable
  - Incremented each time the page is accessed
- Periodically decrements the age bits
  - A page with an age of 0 is an “old” page that has not been referenced in some time and is the best candidate for replacement
- A form of least frequently used policy

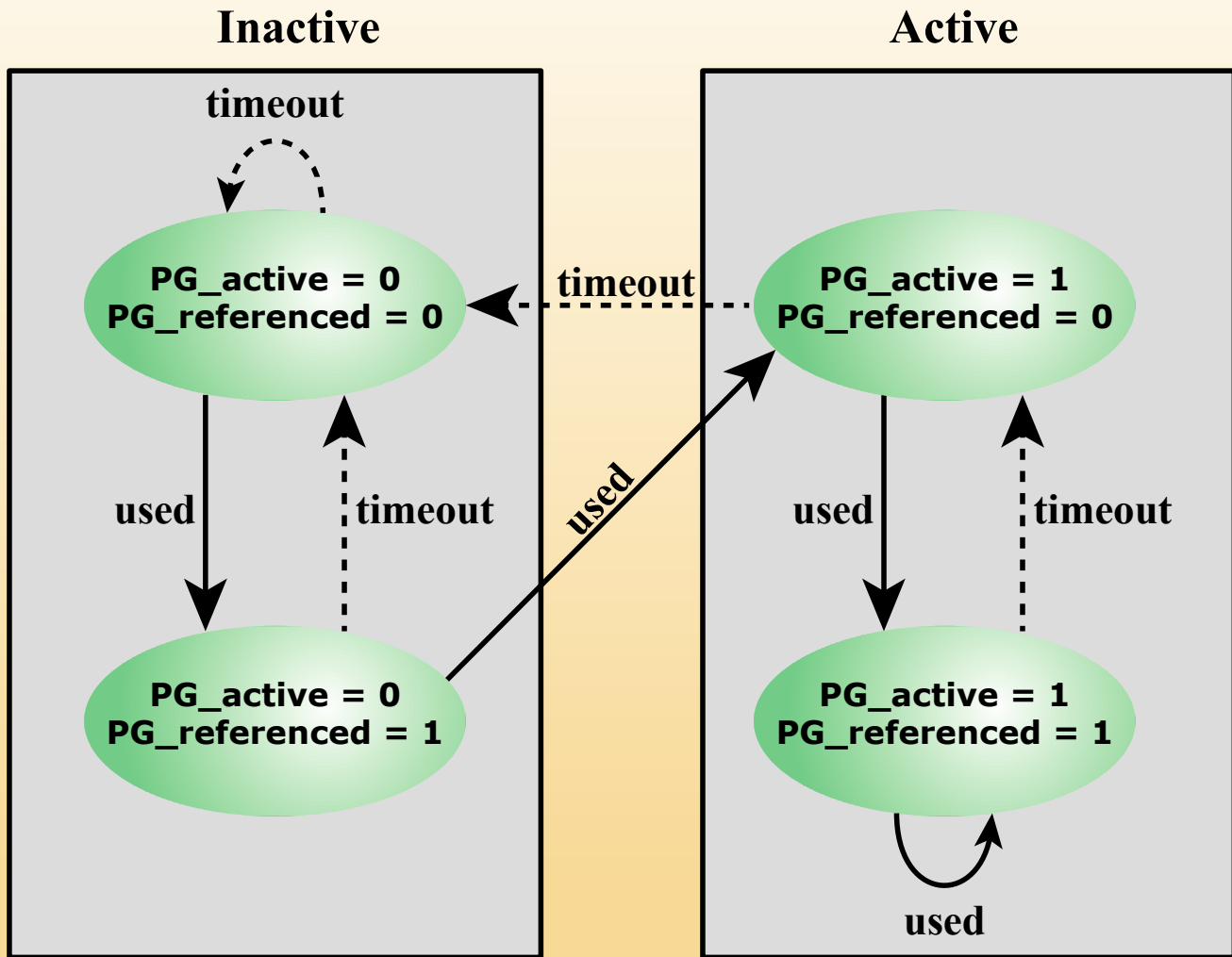


Figure 8.24 Linux Page Reclaiming

# Kernel Memory Allocation

- Kernel memory capability manages physical main memory page frames
  - Primary function is to allocate and deallocate frames for particular uses

## Possible owners of a frame include:

- User-space processes
- Dynamically allocated kernel data
- Static kernel code
- Page cache

- A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages
- Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes
- Slab allocation
  - Used by Linux to accommodate small chunks

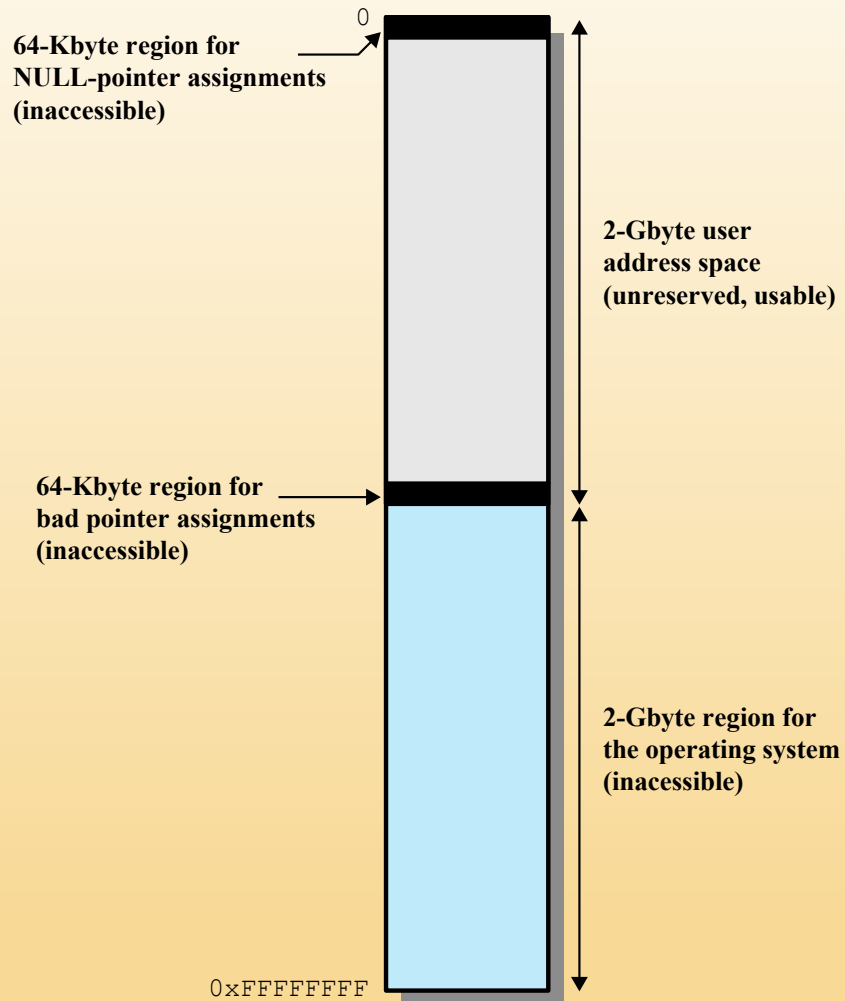


# Windows Memory Management

- Virtual memory manager controls how memory is allocated and how paging is performed
- Designed to operate over a variety of platforms
- Uses page sizes ranging from 4 Kbytes to 64 Kbytes

# Windows Virtual Address Map

- On 32 bit platforms each user process sees a separate 32 bit address space allowing 4 GB of virtual memory per process
  - By default half is reserved for the OS
- Large memory intensive applications run more effectively using 64-bit Windows
- Most modern PCs use the AMD64 processor architecture which is capable of running as either a 32-bit or 64-bit system



**Figure 8.25 Windows Default 32-bit Virtual Address Space**

# Windows Paging

- On creation, a process can make use of the entire user space of almost 2 GB
- This space is divided into fixed-size pages managed in contiguous regions allocated on 64 KB boundaries
- Regions may be in one of three states:



# Resident Set Management System

- Windows uses variable allocation, local scope
- When activated, a process is assigned a data structure to manage its working set
- Working sets of active processes are adjusted depending on the availability of main memory

# Android Memory Management

- Android includes a number of extensions to the normal Linux kernel memory management facility
- These include:
  - ASHMem
    - This feature provides anonymous shared memory, which abstracts memory as file descriptors
    - A file descriptor can be passed to another process to share memory
  - Pmem
    - This feature allocates virtual memory so that it is physically contiguous
    - This feature is useful for hardware that does not support virtual memory
  - Low Memory Killer
    - This feature enables the system to notify an app or apps that they need to free up memory
    - If an app does not cooperate, it is terminated

# Summary

- Hardware and control structures
  - Locality and virtual memory
  - Paging
  - Segmentation
  - Combined paging and segmentation
  - Protection and sharing
- OS software
  - Fetch policy
  - Placement policy
  - Replacement policy
  - Resident set management
  - Cleaning policy
  - Load control
- UNIX and Solaris memory management
  - Paging system
  - Kernel memory allocator
- Linux memory management
  - Linux virtual memory
  - Kernel memory allocation
- Windows memory management
  - Windows virtual address map
  - Windows paging
  - Windows swapping
- Android memory management