Chapter 8
Virtual Memory

The Holodeck

Creates your surroundings on-demand

Virtual Memory

Loads the program's code and data on-demand.

Virtual Memory

Loads the program's code and data on-demand:

- Upon reads, load memory from disk.
- When memory is unused, write it to disk.

Enablers of Virtual Memory

- Addresses are logical, not physical.
- Run-time address translation (paging and segmentation) enable non-contiguous memory layouts.

Implication

A process can successfully run without being fully resident in memory.

What do we need?

Two things:

- Next instruction to execute
- Next data to be accessed

With these items, you're good to go.

Pulling it off

- Load some part of the program.
- Start executing, till program tries to read or execute something not in RAM (page fault).
- Upon page fault, block the process, read in data, resume the process.

Resident set:

portion of process that is in main memory

Page Faults

How does the OS know when to load data?

CPU triggers a page fault upon access to nonresident data/code.

Performance Implications

Is virtual memory harmful to performance?

- Lazy loading is often a win; but
- Disks load batches of data faster.

Functionality WIN

- Can juggle more processes in memory at once.
- A process can exceed system's RAM size.

Real memory: backed by RAM chips.

Virtual memory: RAM and swap space.

Is virtual memory too slow?

Nope.

Key win of lazy loading: CPU keeps busy, hence less wasted time.

Only a small part of each process is active at a time (we hope).

Thrashing

When you put something in, you have to take something out (eventually).

Thrashing

If you throw something out, but need it right away, you lose.

Researchers in the 70s devised algorithms to figure out what to throw out.

http://www.paulgraham.com/stuff.html

Principle of Locality

Paul Graham claims that you never need stuff.

In computers, though, we use the principle of locality:

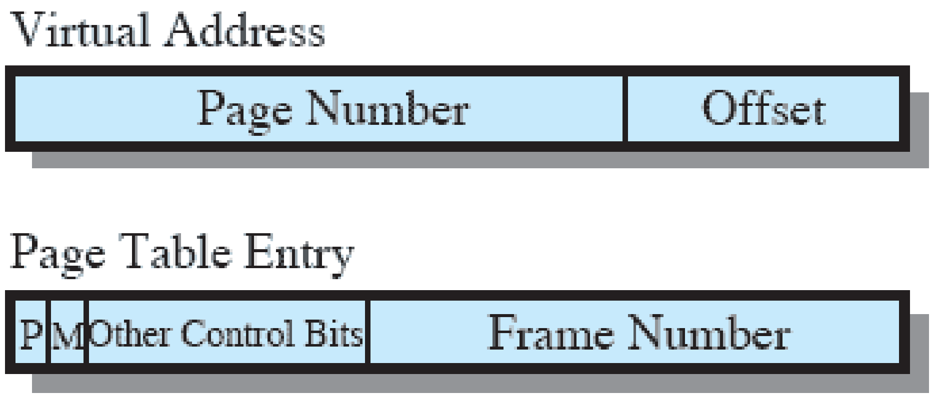
Stuff you need in the future is close to stuff you needed in the past.

Requirements for Virtual Memory

- Hardware must support paging and segmentation
- Operating system support for putting pages on disk and reloading them from disk.

We'll look at hardware support next.

Paging



(a) Paging only

Address Translation

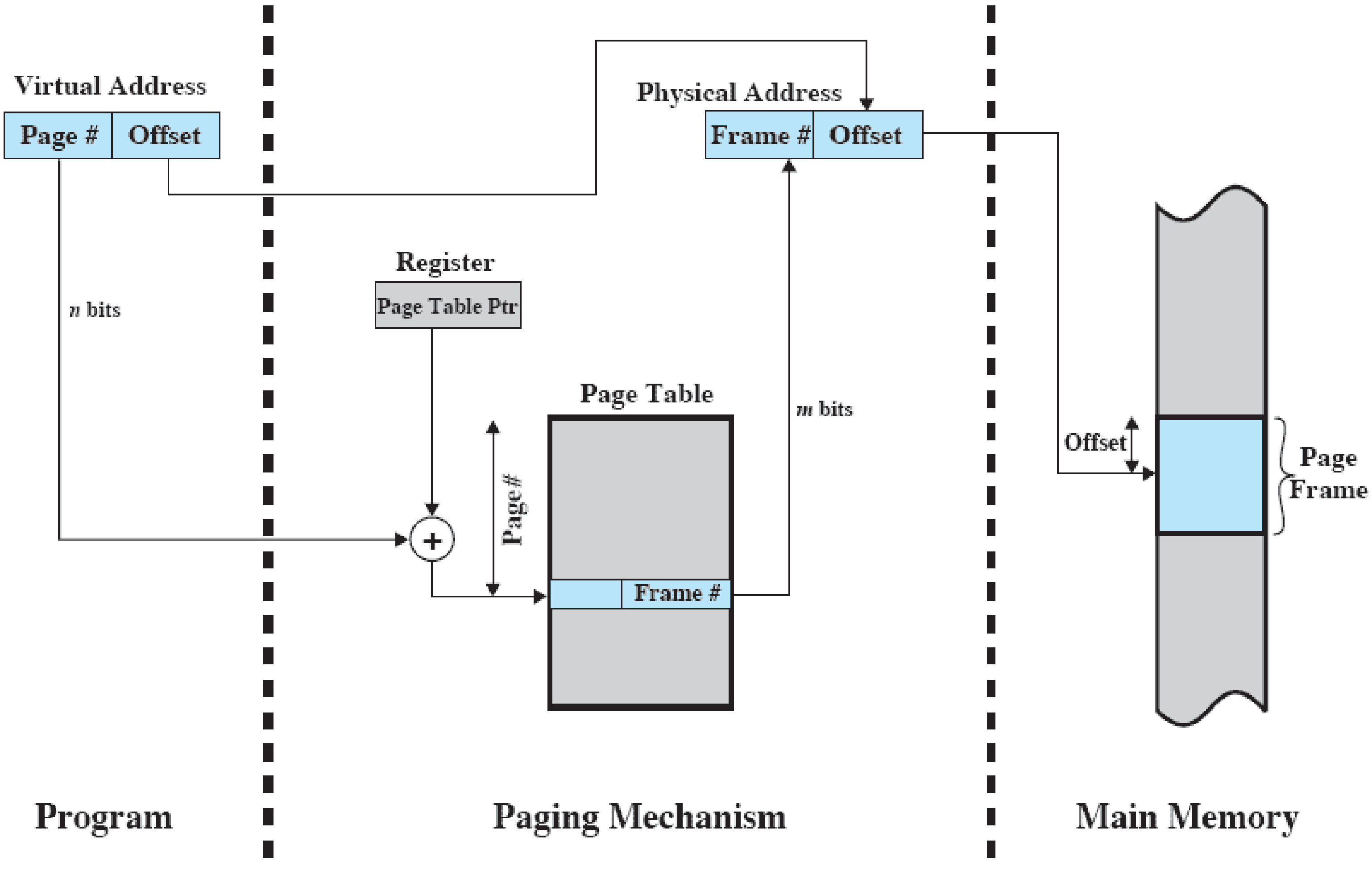


Figure 8.3 Address Translation in a Paging System

Are We Happy Yet?

Not quite: page tables will eat all your RAM!

(e.g. 4MB of page tables per process
for 4GB RAM)

Paging Page Tables

Put the page tables in virtual memory.

Obviously, must keep page table entry of the currently-executing page in main memory.

Two-Level Hierarchical Page Table

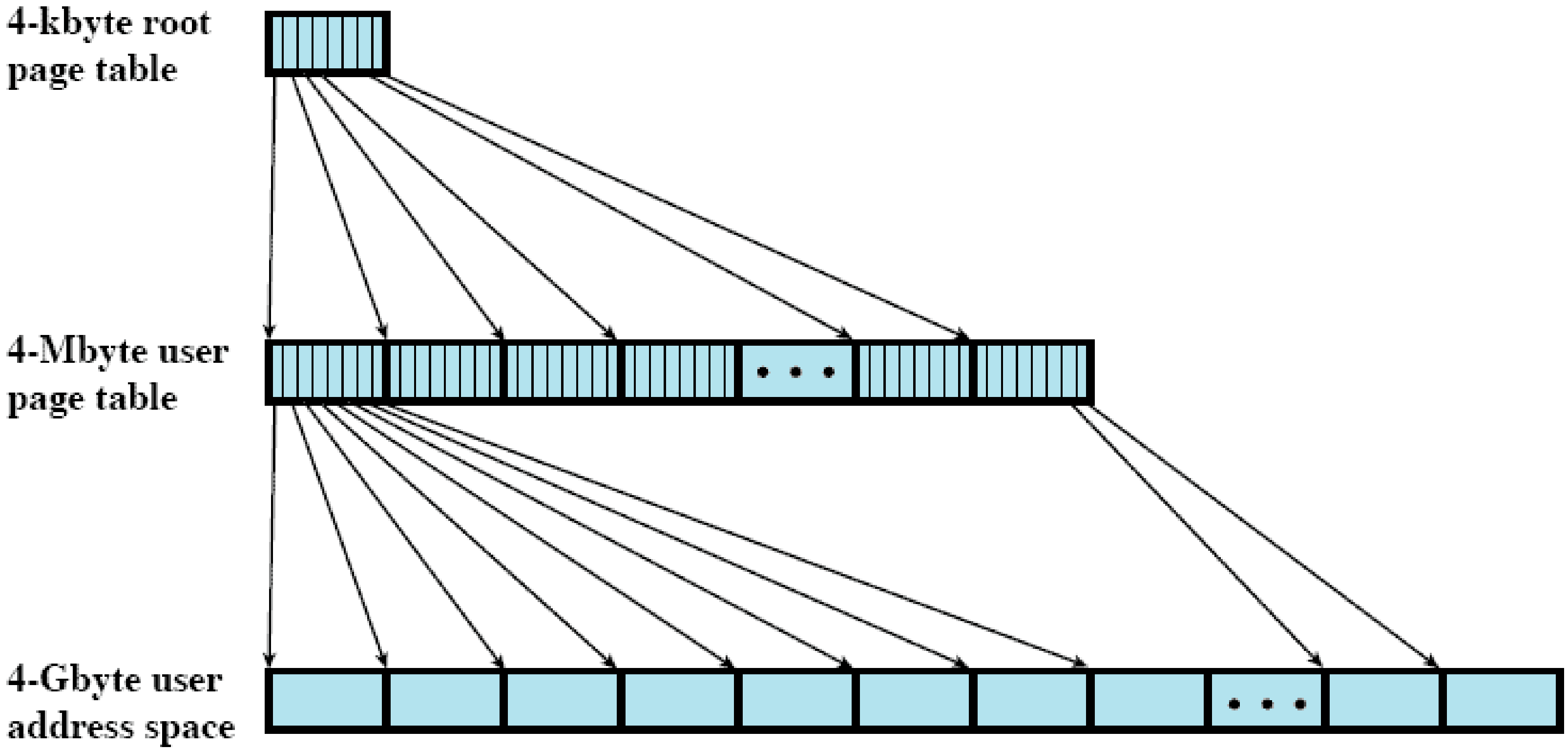


Figure 8.4 A Two-Level Hierarchical Page Table

Address Translation

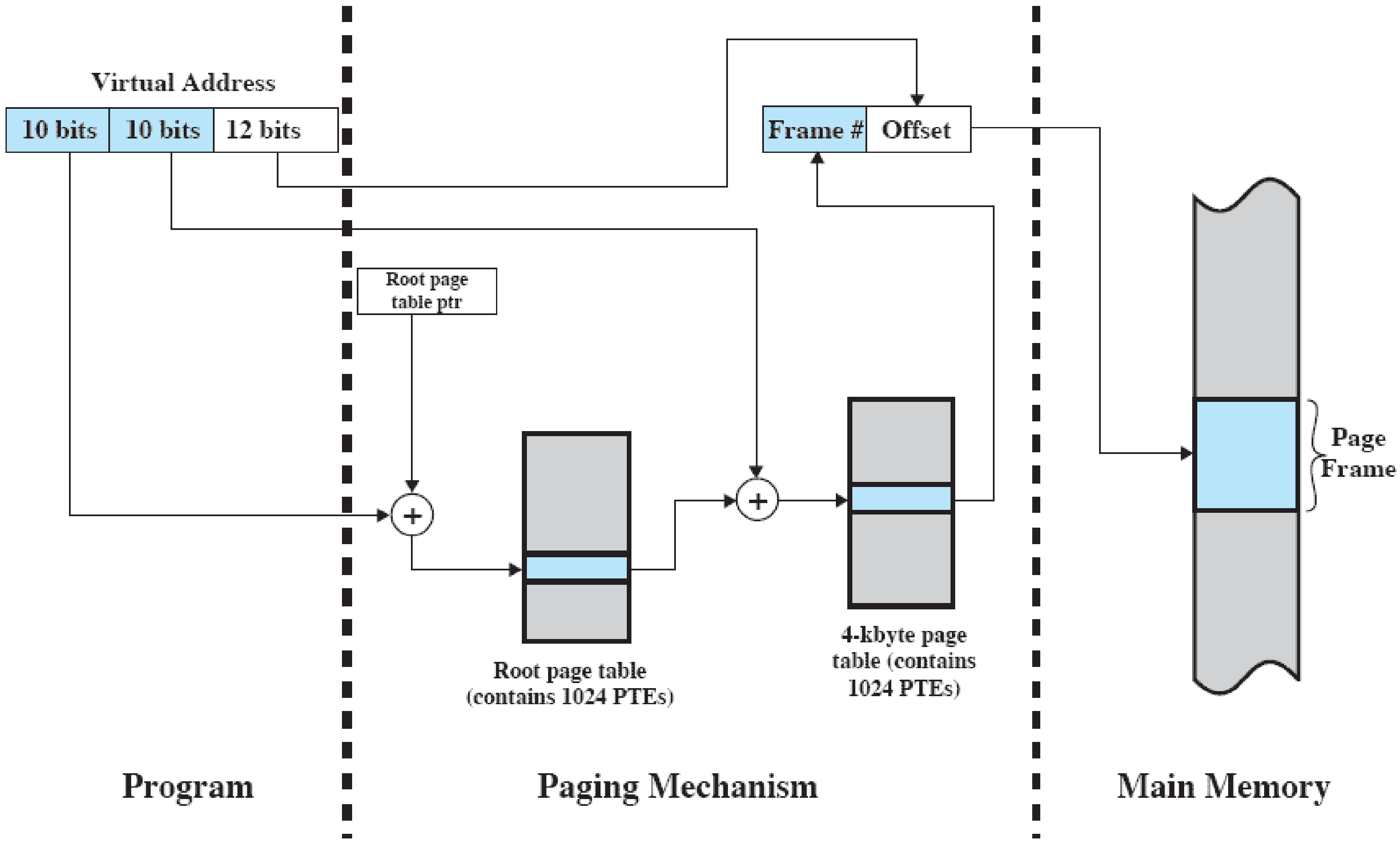


Figure 8.5 Address Translation in a Two-Level Paging System

Drawback of Page tables

- Page table size is proportional to the virtual address space.

Alternative:

- Inverted page table: use a hash table.

Inverted Page Tables

Inverted = index on frame #, not virtual address.

Inverted tables grow with size of physical memory.

Inverted Page Table

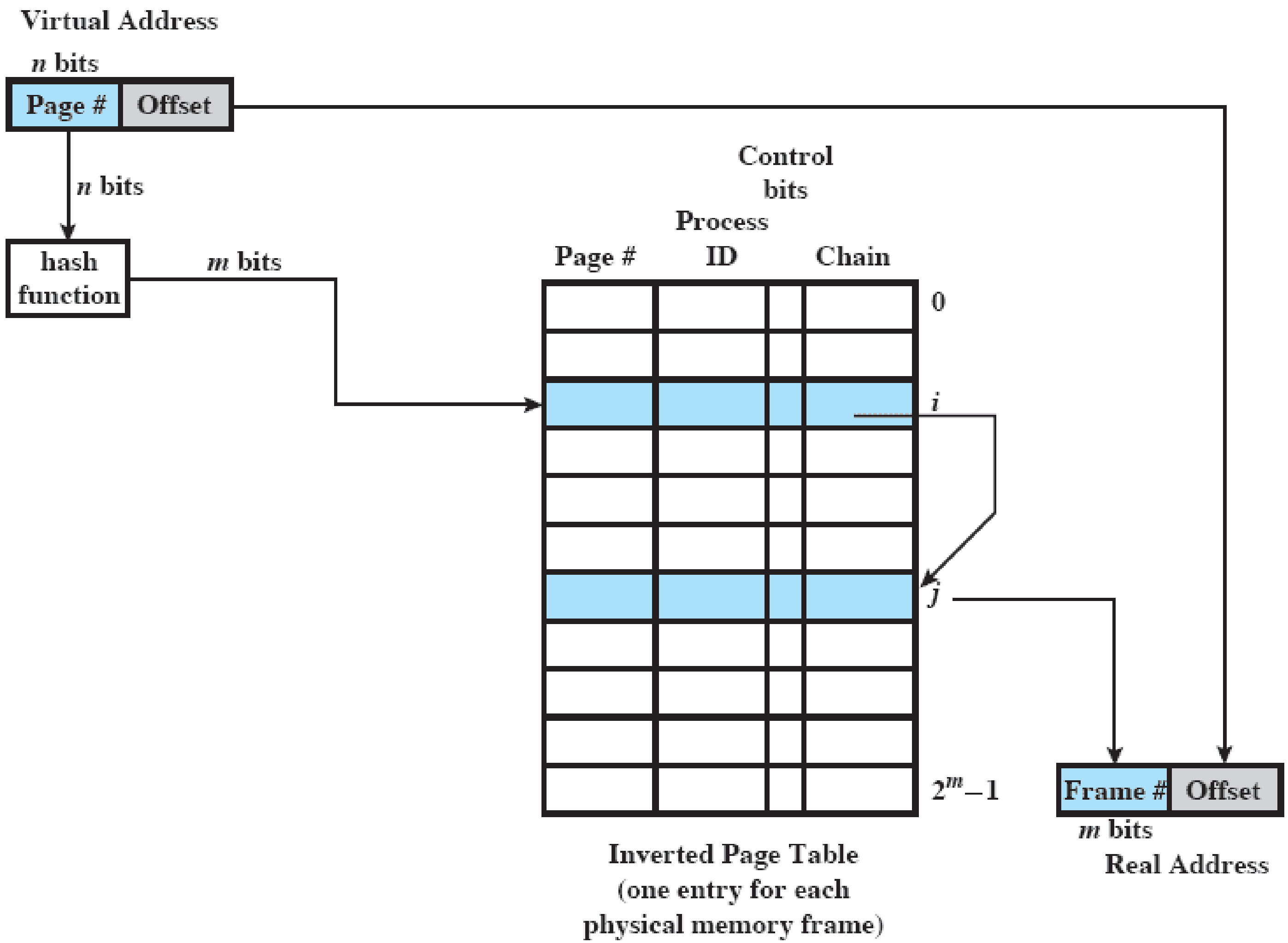


Figure 8.6 Inverted Page Table Structure

Translation Lookaside Buffer

Each virtual memory reference can cause two physical memory accesses

- One to fetch the page table entry
- One to fetch the data

What do you do when a system is too slow?

Add caches!

Translation Lookaside Buffer

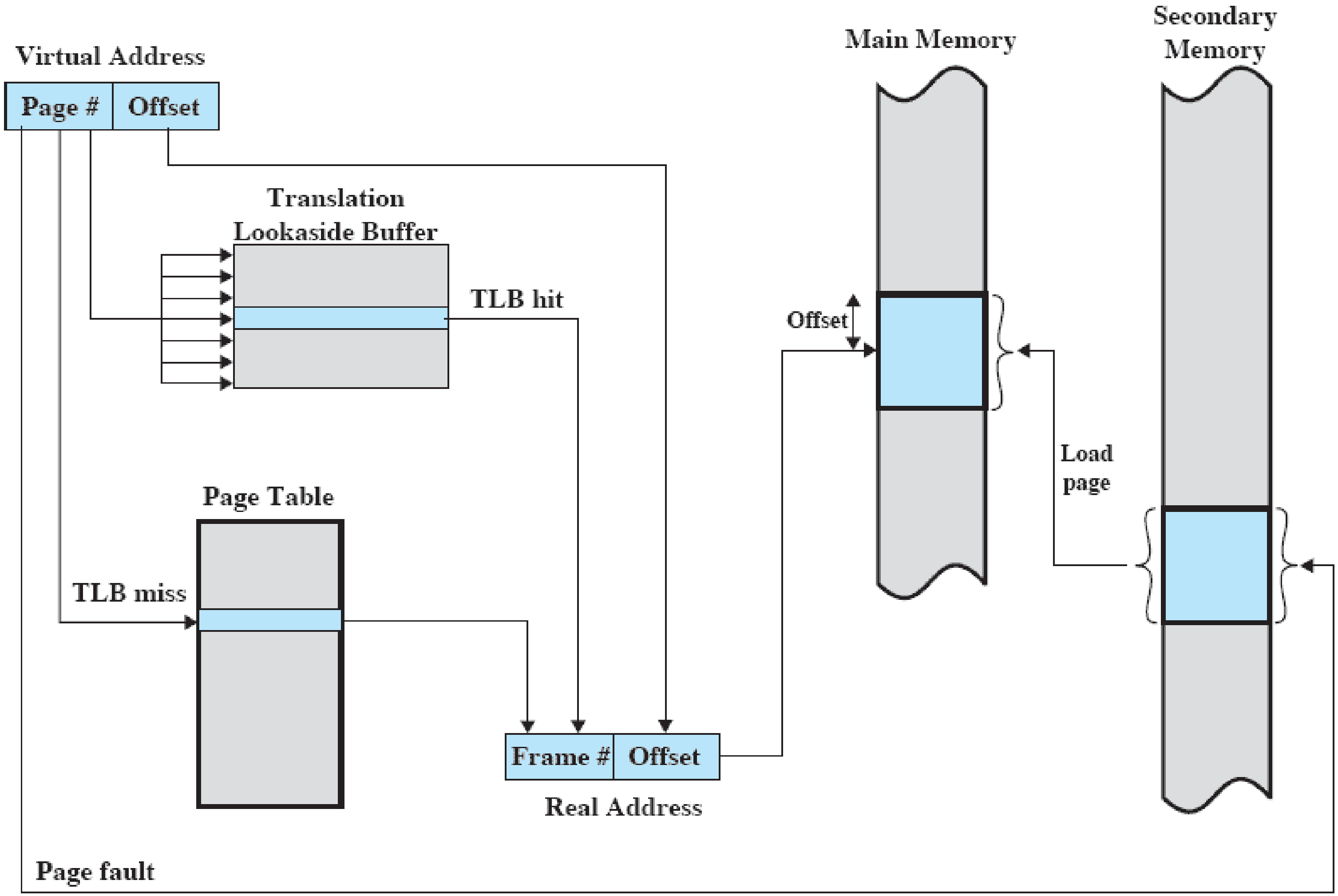


Figure 8.7 Use of a Translation Lookaside Buffer

Translation Lookaside Buffer

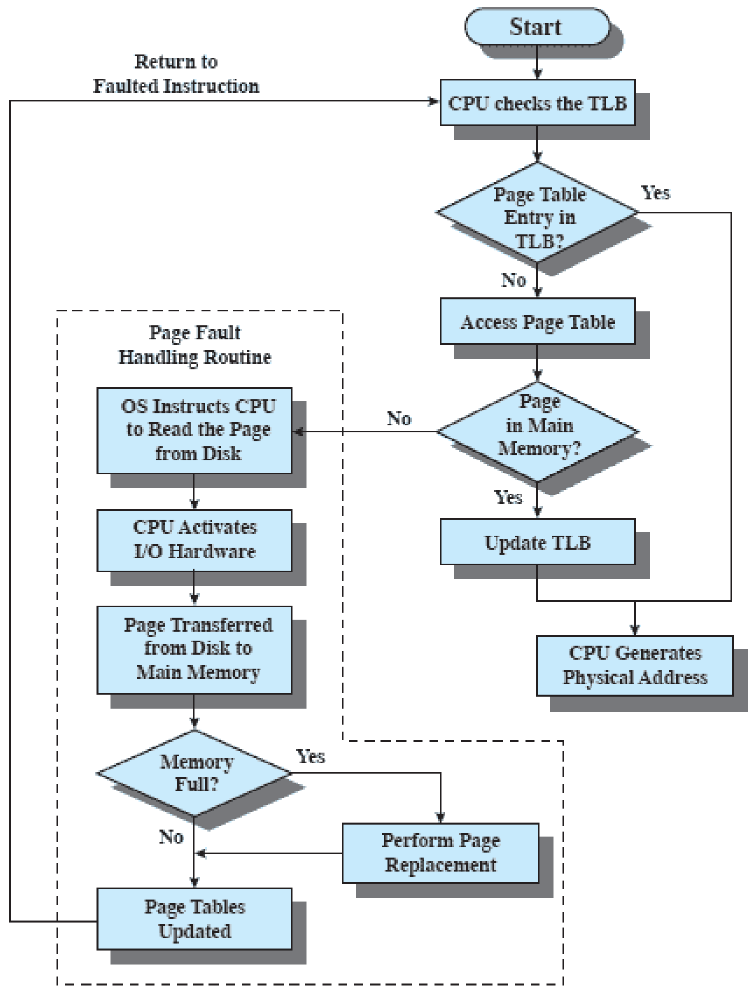


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

Translation Lookaside Buffer

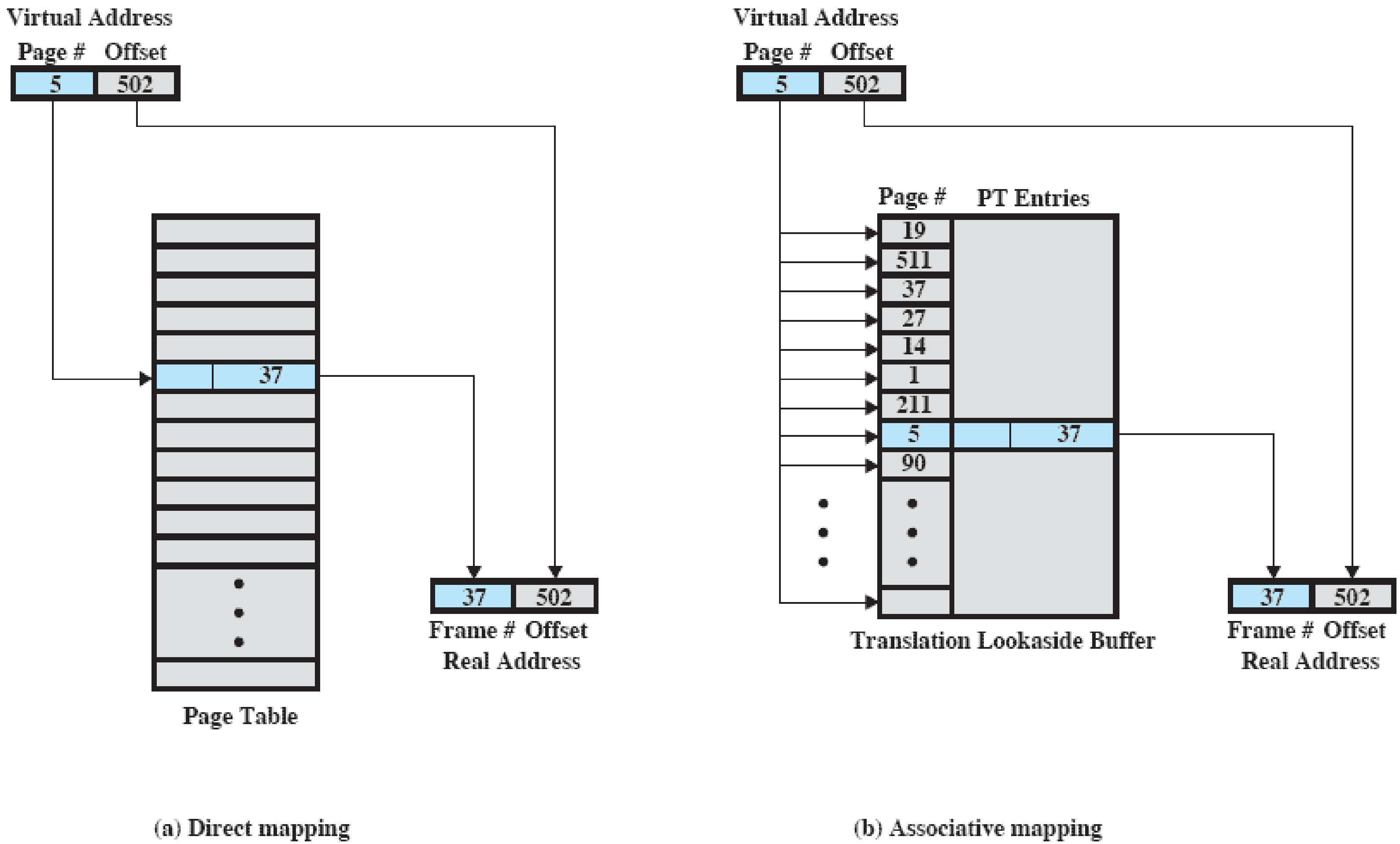


Figure 8.9 Direct Versus Associative Lookup for Page Table Entries

Translation Lookaside Buffer

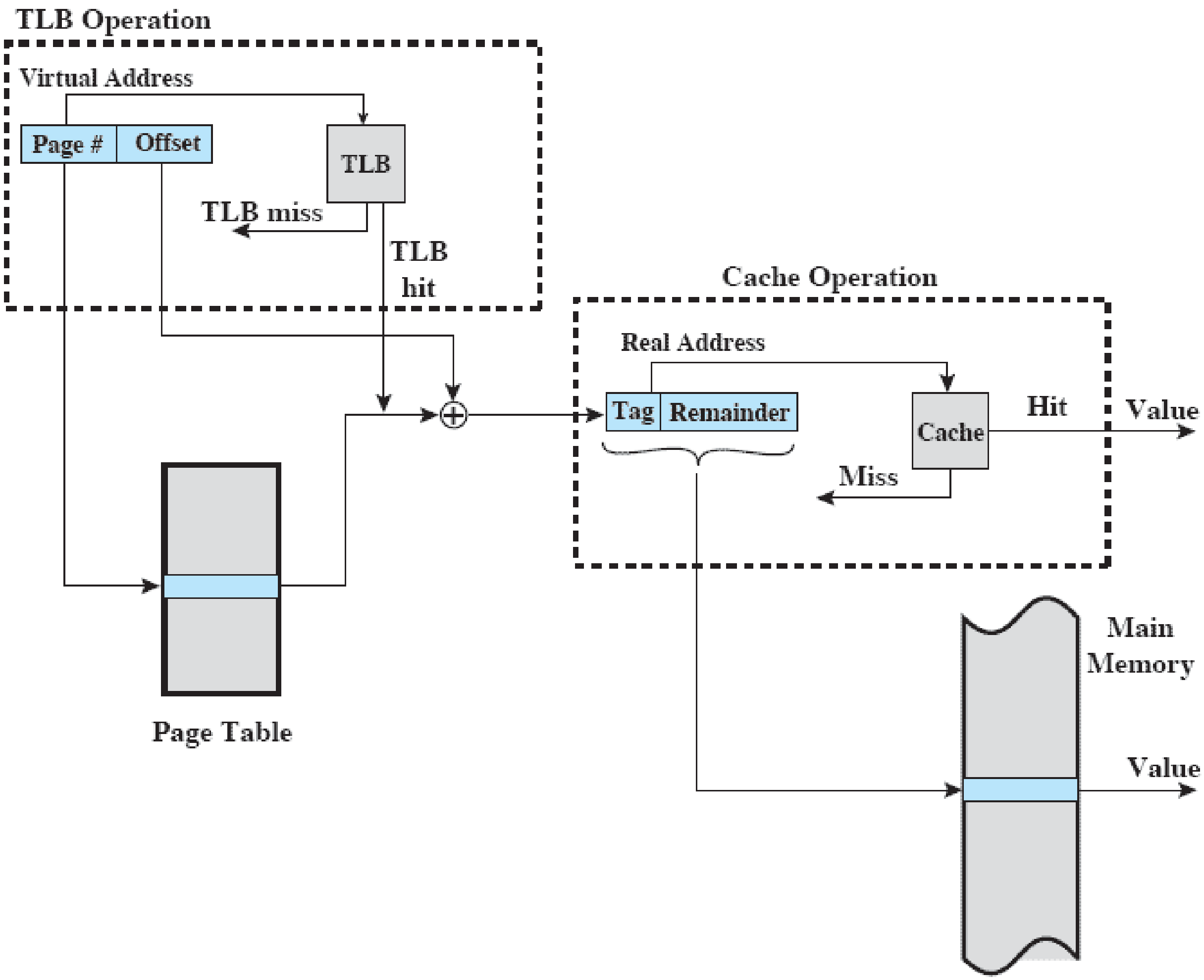


Figure 8.10 Translation Lookaside Buffer and Cache Operation

Page Size: Large?

- Smaller page size, less amount of internal fragmentation (you can only allocate pages [story about FS blocks])
- Smaller page size, more pages required per process
- More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory

Page Size: Small?

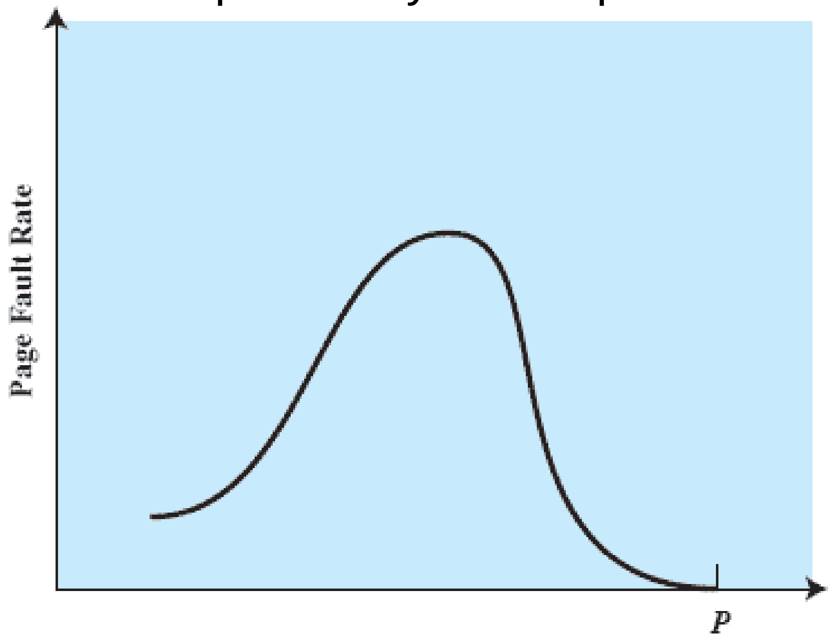
- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better
  - Disk produces streams

Page Size: Small?

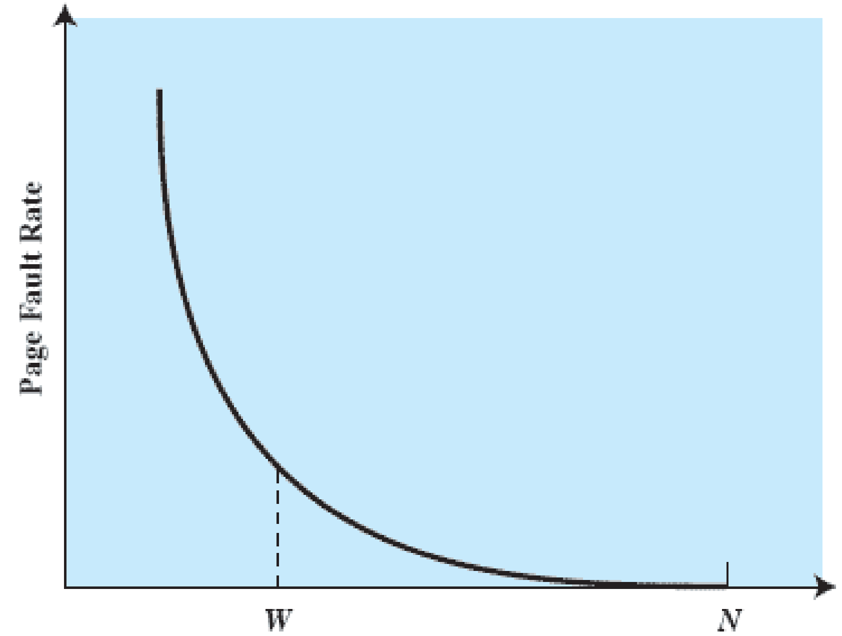
- Small page size, large number of pages will be found in main memory
- As execution of the program progresses, the pages in memory will all contain portions of the process near recent references
  → Page faults low
- Increased page size causes pages to contain locations further from any recent reference
  → Page faults rise

Page Size

Figure for the previously shown phenomenon.



(a) Page Size



(b) Number of Page Frames Allocated

The more pages for a process, the less page faults.

P = size of entire process
W = working set size
N = total number of pages in process

Figure 8.11 Typical Paging Behavior of a Program

Page Size Interactions

- Small page size => less internal fragmentation
- Small page size => more pages, larger page tables (possible double whammy on page table & page miss)
- Small page size => more pages, less TLB entries, more TLB misses
- Small page size => more disk access

Segmentation

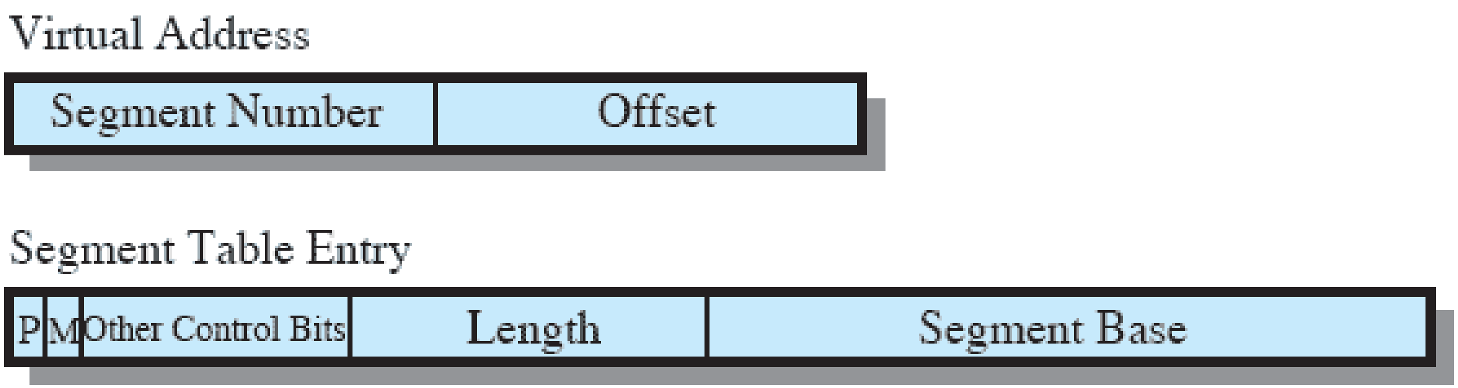
… allows the programmer to view memory as consisting of multiple address spaces or segments.

- Advantages:
  - Simplifies handling of growing data structures (→ put the whole structure into one segment)
  - Allows programs to be altered and recompiled independently (→ code, data have their own segments)
  - Lends itself to sharing data among processes (→ share a segment)
  - Lends itself to protection (→ protect segments)

Segment Tables

- Starting address corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if 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

Segment Table Entries



(b) Segmentation only

Segmentation

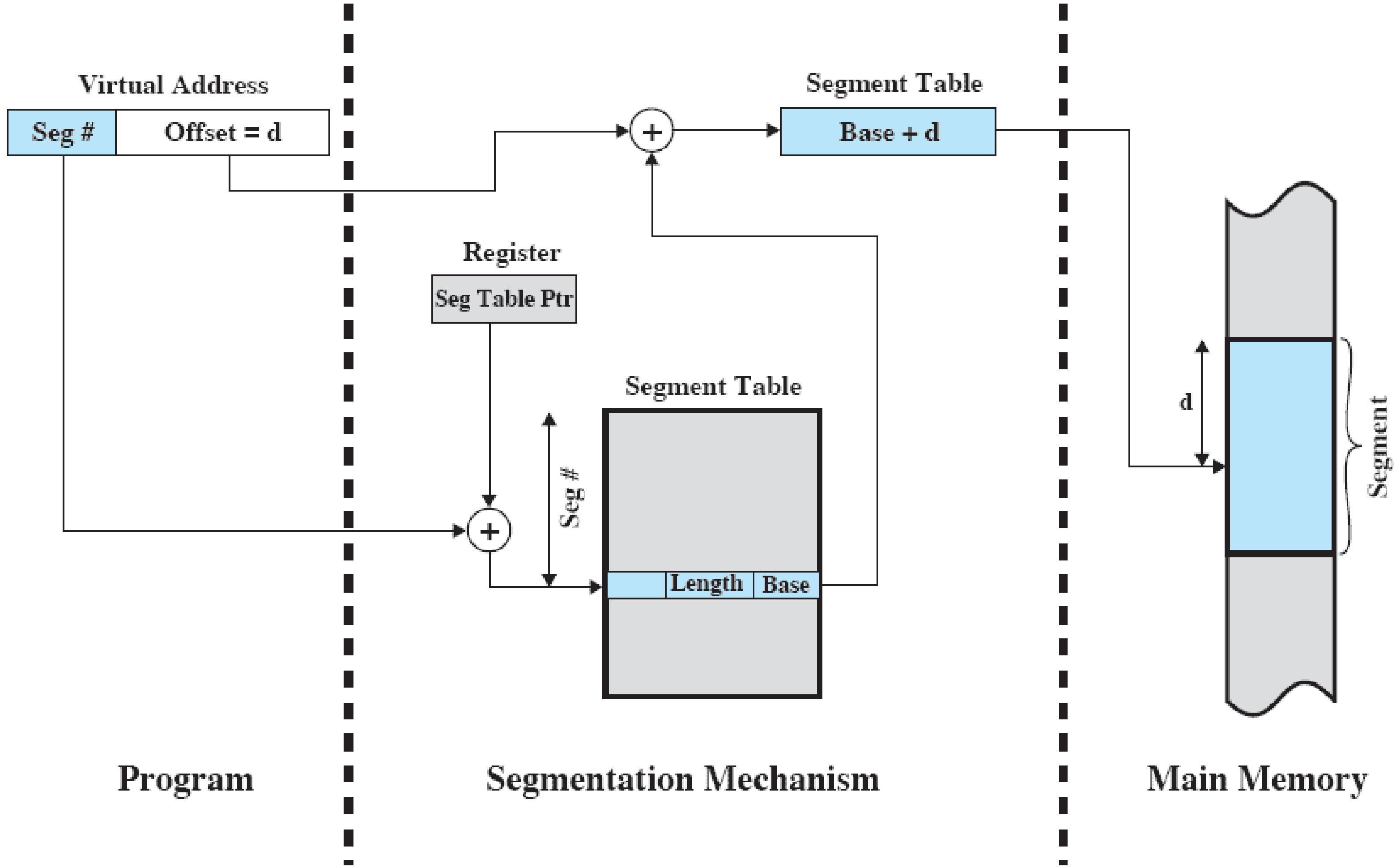
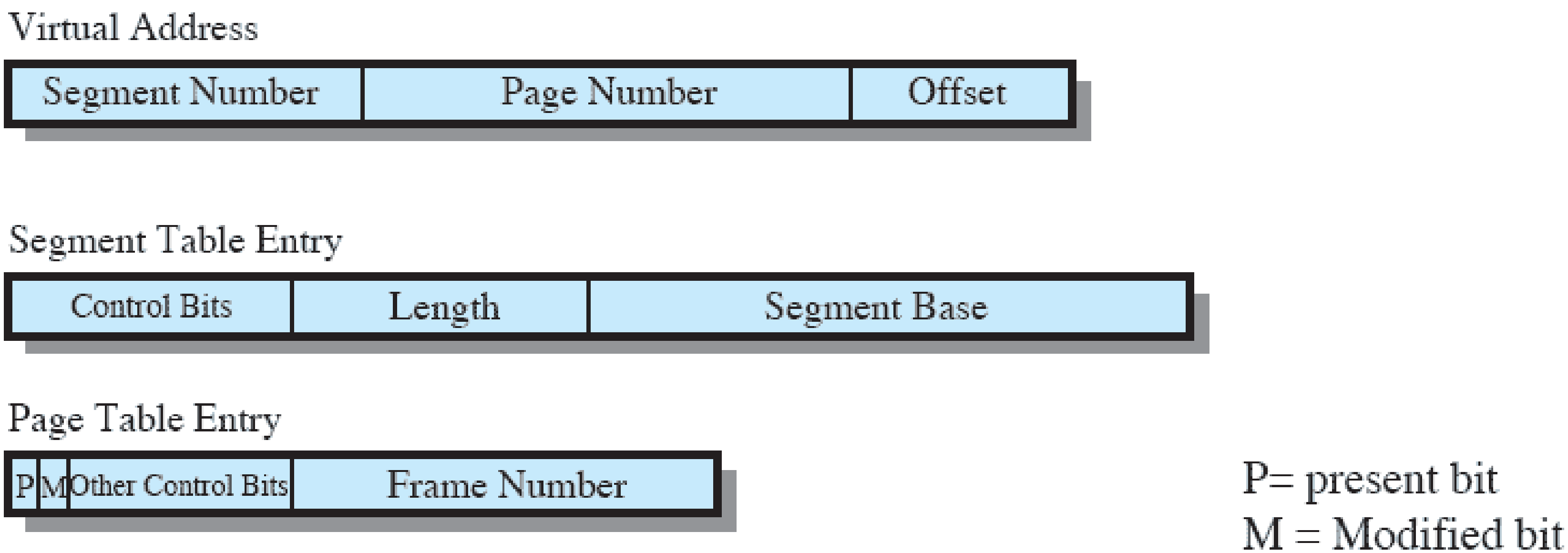


Figure 8.12 Address Translation in a Segmentation System

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Elements:
  - 1 process
  - 1 segment table per process
  - 1 page table per segment

Combined Paging and Segmentation



(c) Combined segmentation and paging

Address Translation

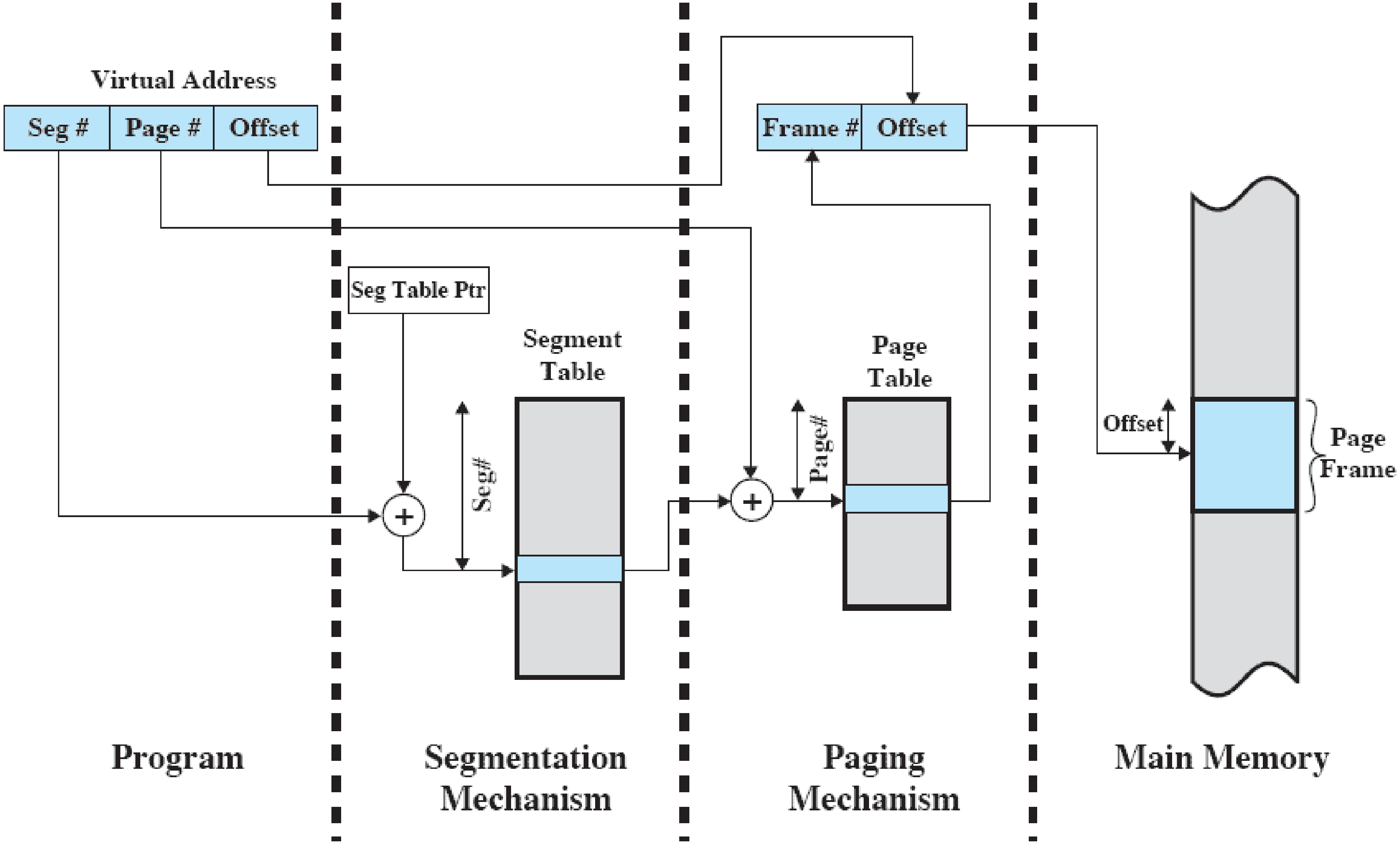


Figure 8.13 Address Translation in a Segmentation/Paging System

Protection Relationships

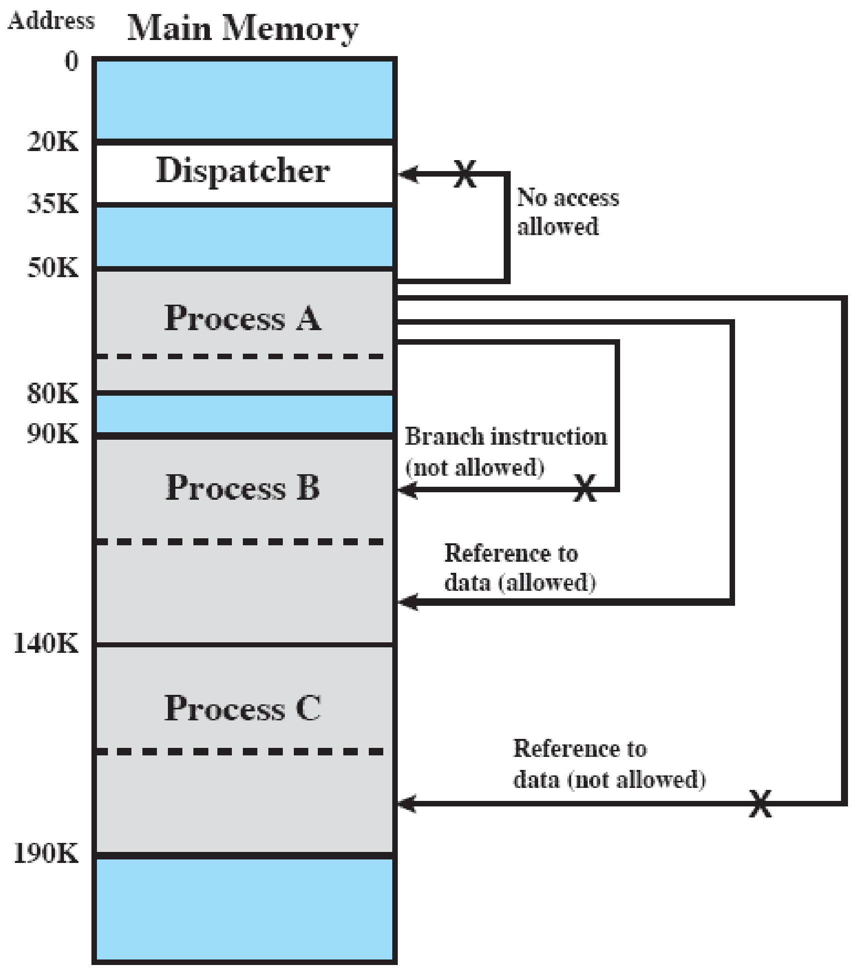


Figure 8.14 Protection Relationships Between Segments

Operating System Software

OS Designer needs to make three choices:

- Does the system need virtual memory?
- Paging, segmentation or both?
- Which algorithms for memory management?

- Required algorithms:
  - Fetch policy
  - Placement policy
  - Replacement policy
  - Cleaning policy
  - (Load control)

Fetch Policy

- Determines when a page should be brought into memory
- Demand paging only brings pages into main memory when a reference is made to a location on the page
  - Many page faults when process first started
- Prepaging brings in more pages than needed
  - More efficient to bring in pages that reside contiguously on the disk

Placement Policy

- Determines where in real memory a process piece is to reside
- Important in a segmentation system (minimize fragmentation)
- Paging or combined paging with segmentation hardware performs address translation

Replacement Policy

- Which page is replaced?
- Page removed should be the page least likely to be referenced in the near future
  - Obey principle of locality (→ high correlation between recent referencing history & near-future referencing pattern)
- Most policies predict the future behavior on the basis of past behavior

Replacement Policy

- Basic concepts for the policy:
  - How many page frames are to be allocated to each active process (fixed-size, variable)?
  - Local vs global scope when replacing page frames in main memory.
  - From the many candidate pages, which one?

Replacement Policy

- Frame Locking
  - Restricts the placement policy
  - If frame is locked, it may not be replaced
  - Used for example in:
    - Kernel of the operating system
    - Key control structures
    - I/O buffers
    - Time critical elements
  - Associate a lock bit with each frame

Basic Replacement Algorithms

- Optimal policy
  - Selects for replacement that page for which the time to the next reference is the longest
  - Impossible to have perfect knowledge of future events

Basic Replacement Algorithms

- Least Recently Used (LRU)
  - Uses the assumption of the principle of locality
  - Nearly as good as the optimal algorithm
  - 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
  - Each page could be tagged with the time of last reference. This would require a great deal of overhead. → clock algorithms

Basic Replacement Algorithms

- First-in, first-out (FIFO)
  - Page that has been in memory the longest is replaced
  - Bases on the assumption that an ‘old’ page will not be referenced soon again.
  - Treats page frames allocated to a process as a circular buffer
  - Pages are removed in round-robin style
  - Simplest replacement policy to implement

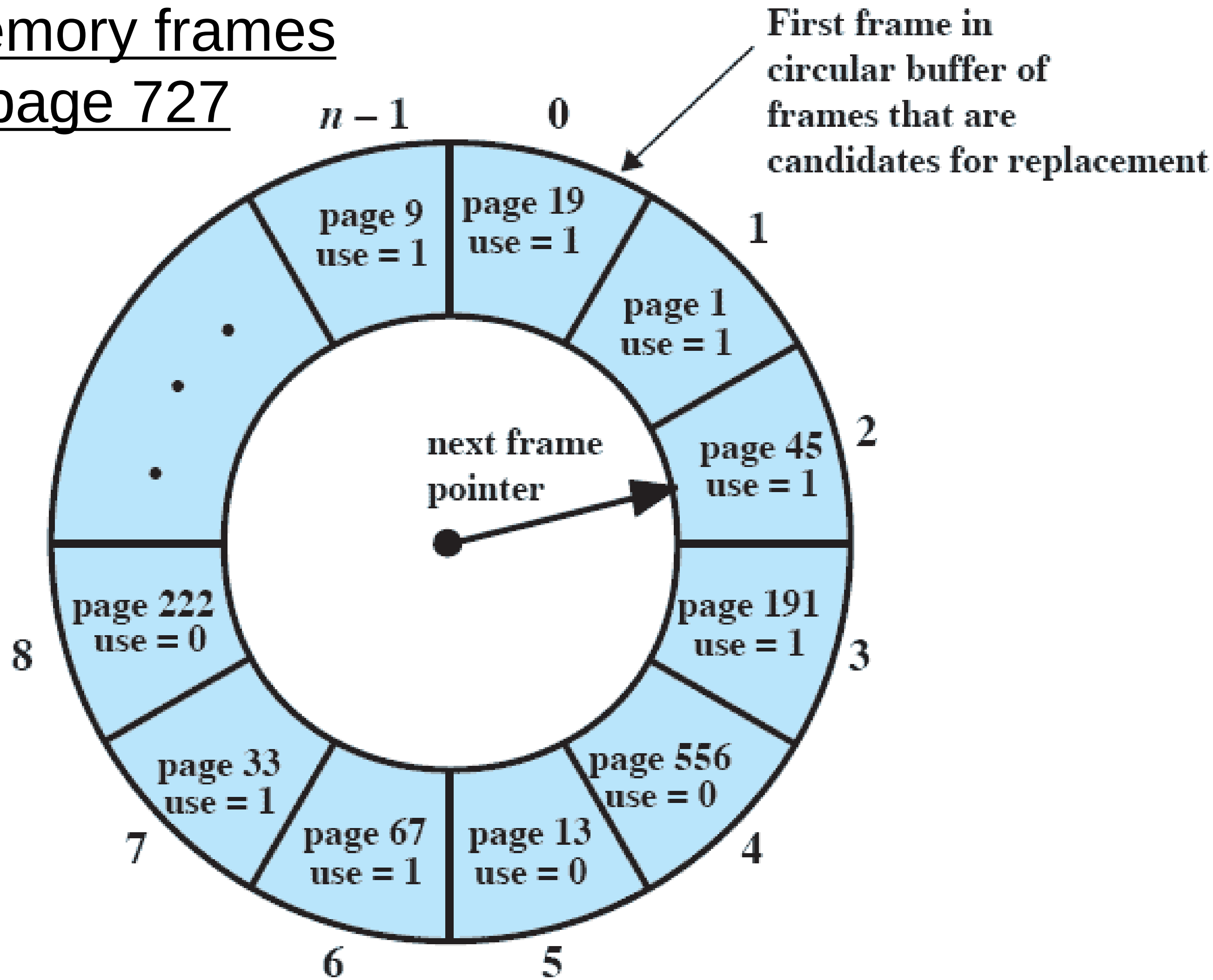
Basic Replacement Algorithms

- Clock Policy … approximates LRU
  - Additional bit called a use bit
  - When a page is first loaded in memory, the use bit is set to 1
  - When the page is referenced, the use bit is set to 1
  - When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
  - During the search for replacement, each use bit set to 1 is changed to 0

Clock Policy

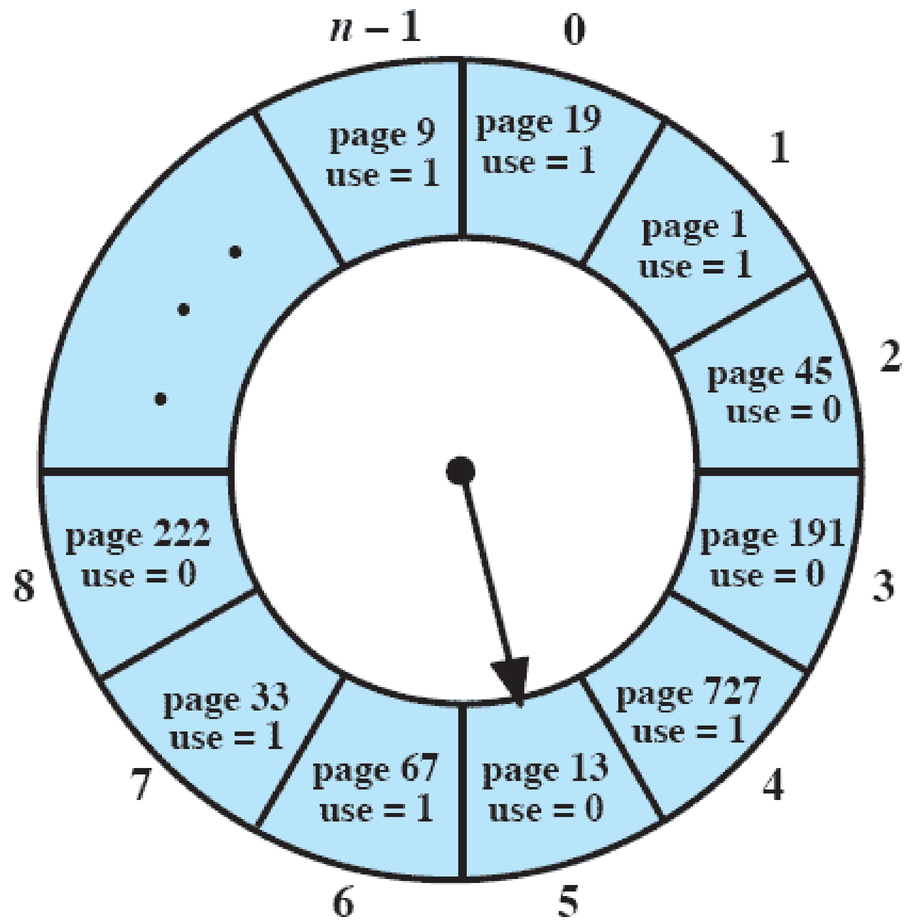
n main memory frames

incoming page 727



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

Clock Policy



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

Figure 8.16 Example of Clock Policy Operation

Clock Policy

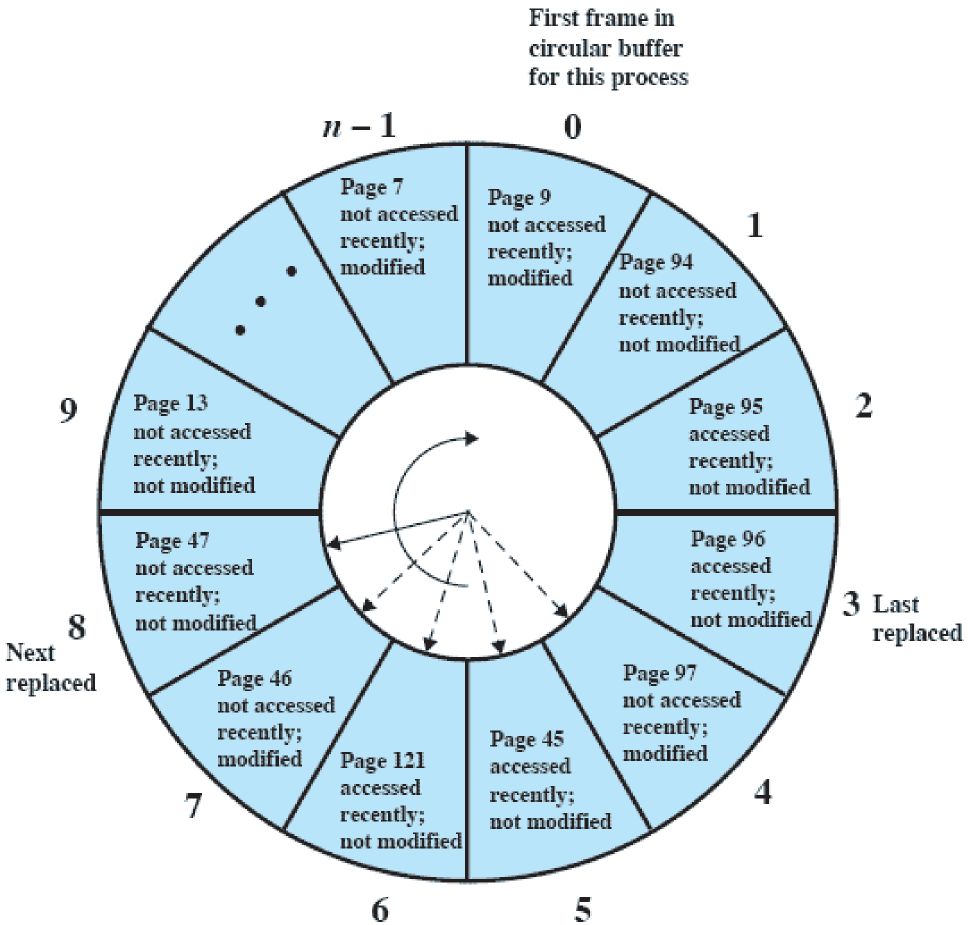
Use bit, modified bit

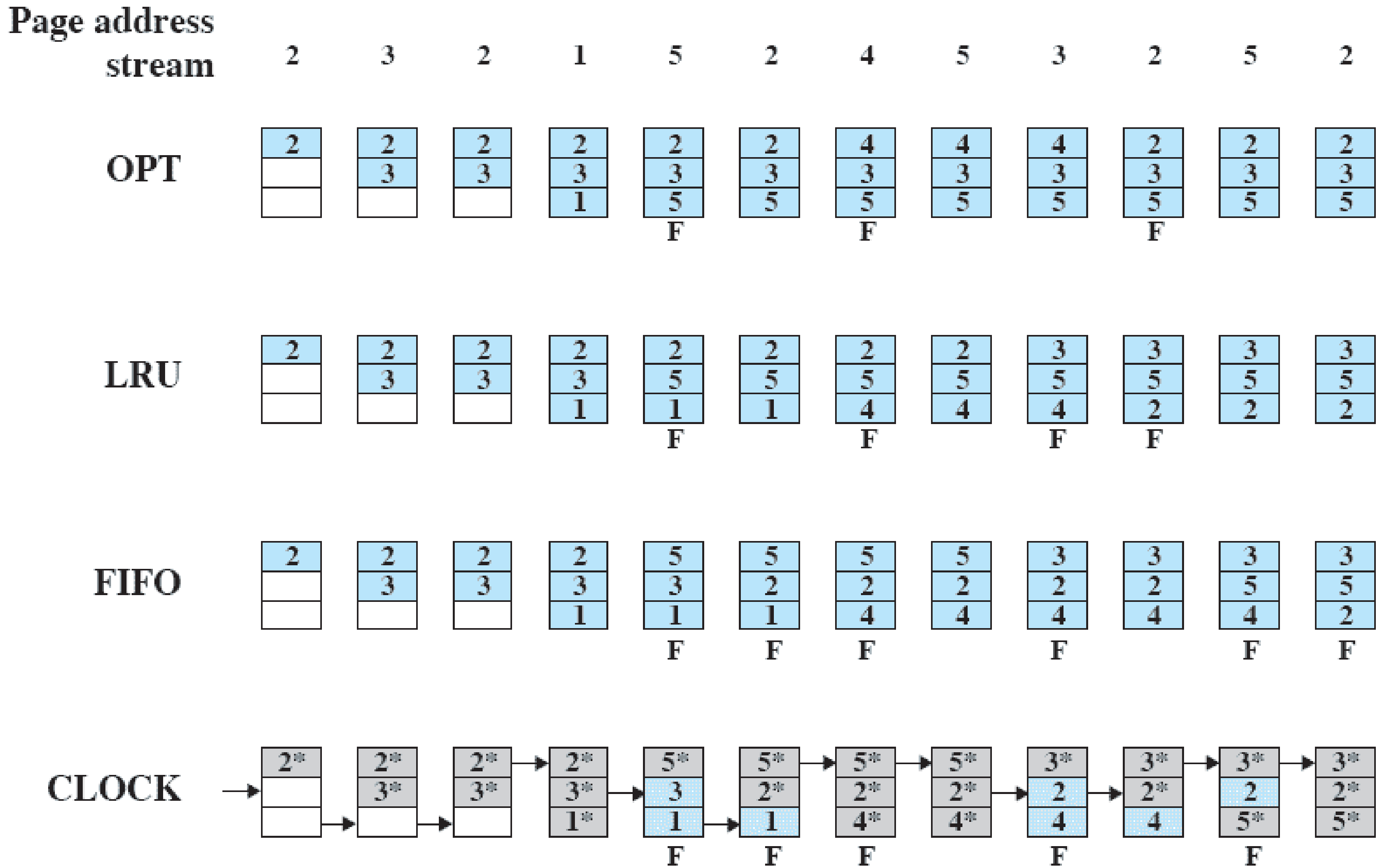
1.Look for u=0, m=0

2.Cycle & look for u=0, m=1

u--

3.Goto 1





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

Figure 8.15 Behavior of Four Page-Replacement Algorithms

Basic Replacement Algorithms

- Page Buffering
  – Implements a cache for memory pages
  – Replaced page is added to one of two lists
    - Free page list if page has not been modified
    - Modified page list
  - The idea is that the OS can revive these pages from the list if space becomes available.
  - Supports bursty block writes of I/O

Resident Set

- The pages of a process in memory

- Important factors
  - The smaller the amount, the more processes
  - Too small → high page fault rate
  - Too big → no real gain

Resident Set Size

- Fixed-allocation
  - Gives a process a fixed number of pages within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
  - Number of pages allocated to a process varies over the lifetime of the process

Fixed Allocation, Local Scope

- Decide ahead of time the amount of allocation to give a process. <u>(hard)</u>
- 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
  - Processor idle time
  - Swapping

Variable Allocation, Global Scope

- Easiest to implement
- Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces anyone
- Risk: a process can suffer reduction in its resident set size

Variable Allocation, Local Scope

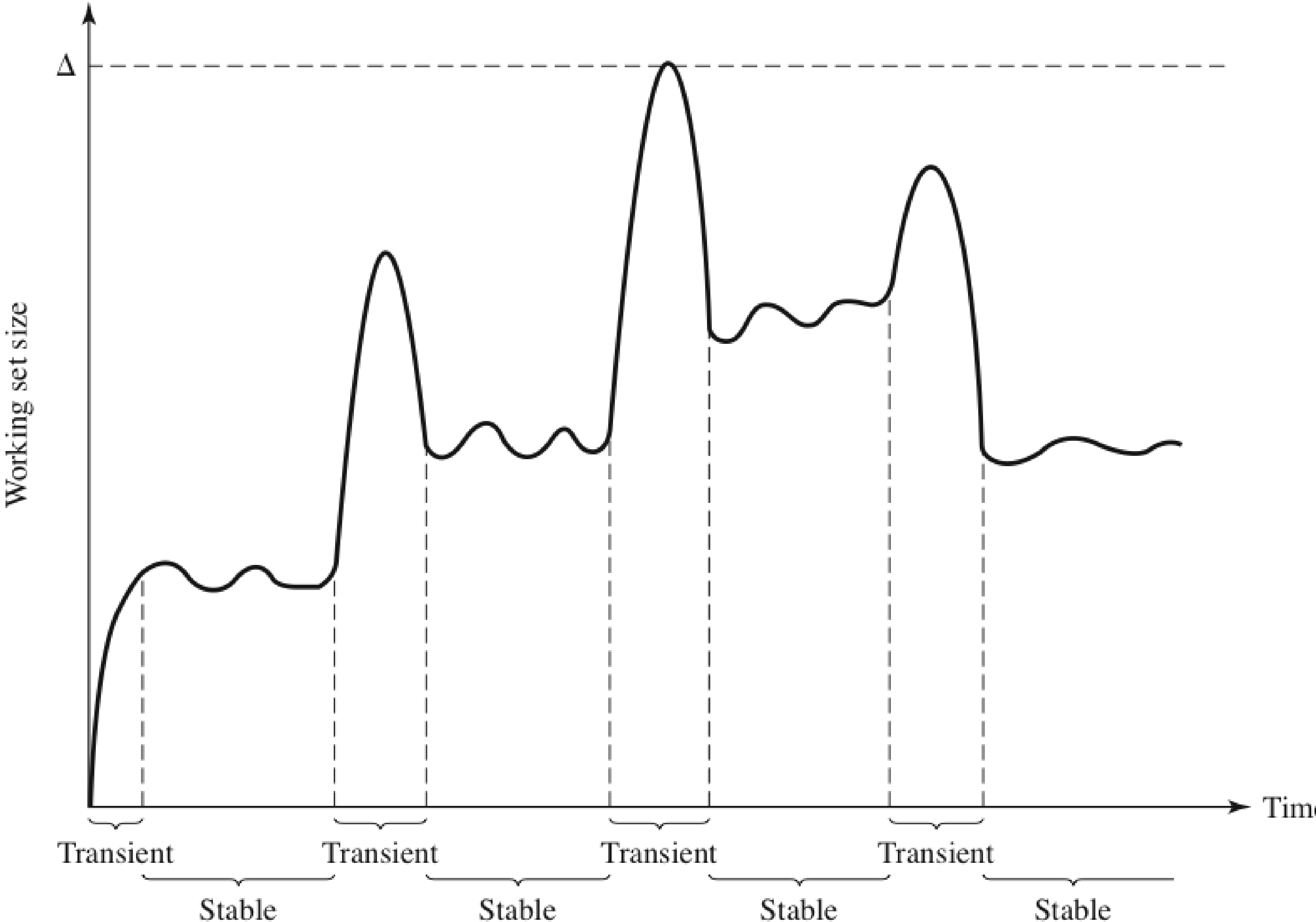
- When new process added, allocate number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time

Working Set

- The set of pages of the process that have been referenced in the last t logical time units.

Sequence of Page References	Window Size, Δ 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 8.19 Working Set of Process as Defined by Window Size


Δ
Working set size
Time
Transient
Transient
Transient
Transient
Stable
Stable
Stable
Stable

Use of the Working Set For Resident Set Size

1. Monitor the working set of each process.
2. Remove pages from the resident set but not in working set (= LRU).
3. A process may only execute if its working set is in main memory.

- Problems:
  - Past does not predict the future
  - Impractical to implement
  - Optimal window size is unknown and varies

→ observe page fault frequency

Cleaning Policy

- Demand cleaning
  - A page is written out only when it has been selected for replacement
  - Requires two page transfers per page fault
- Pre-cleaning
  - Pages are written out in batches
  - Allows bursty long (large block) I/O
  - … but write may be unnecessary

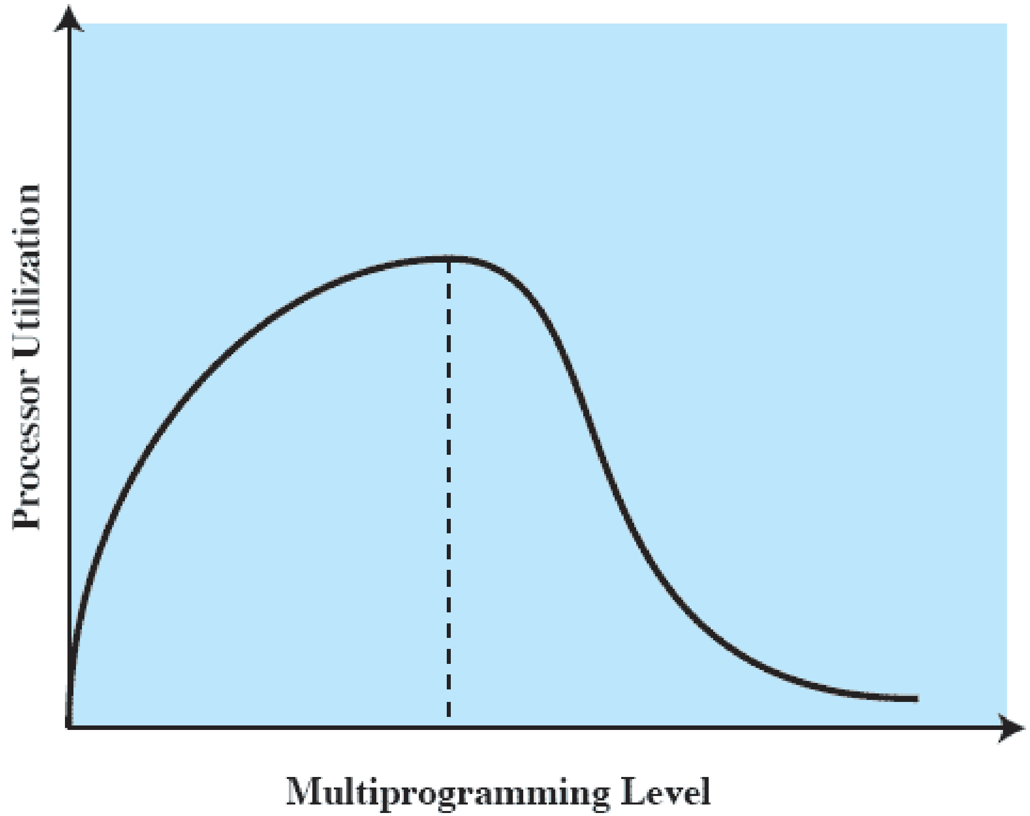
Cleaning Policy

- Best approach uses page buffering
  - Replaced pages are placed in two lists
    - Modified and unmodified
  - Pages in the modified list are periodically written out in batches
  - Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

Load Control

- Determines the number of processes that will be resident in main memory
- 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

Multiprogramming



How to find the sweet spot?

- page fault frequency monitoring
- L=S criterion (mean time between faults = time to process a fault)
- 50% load of page device criterion (similar to L=S)
- monitor clock algorithm

Process Suspension

- If multiprogramming is to be reduced, then which process will be swapped out?
- Lowest priority process
- Faulting process
  - This process does not have its working set in main memory so it will be blocked anyway
- Last process activated
  - This process is least likely to have its working set resident

Process Suspension

- Process with smallest resident set
  - This process requires the least future effort to reload (→ disobeys principle of locality)
- Largest process
  - Obtains the most free frames