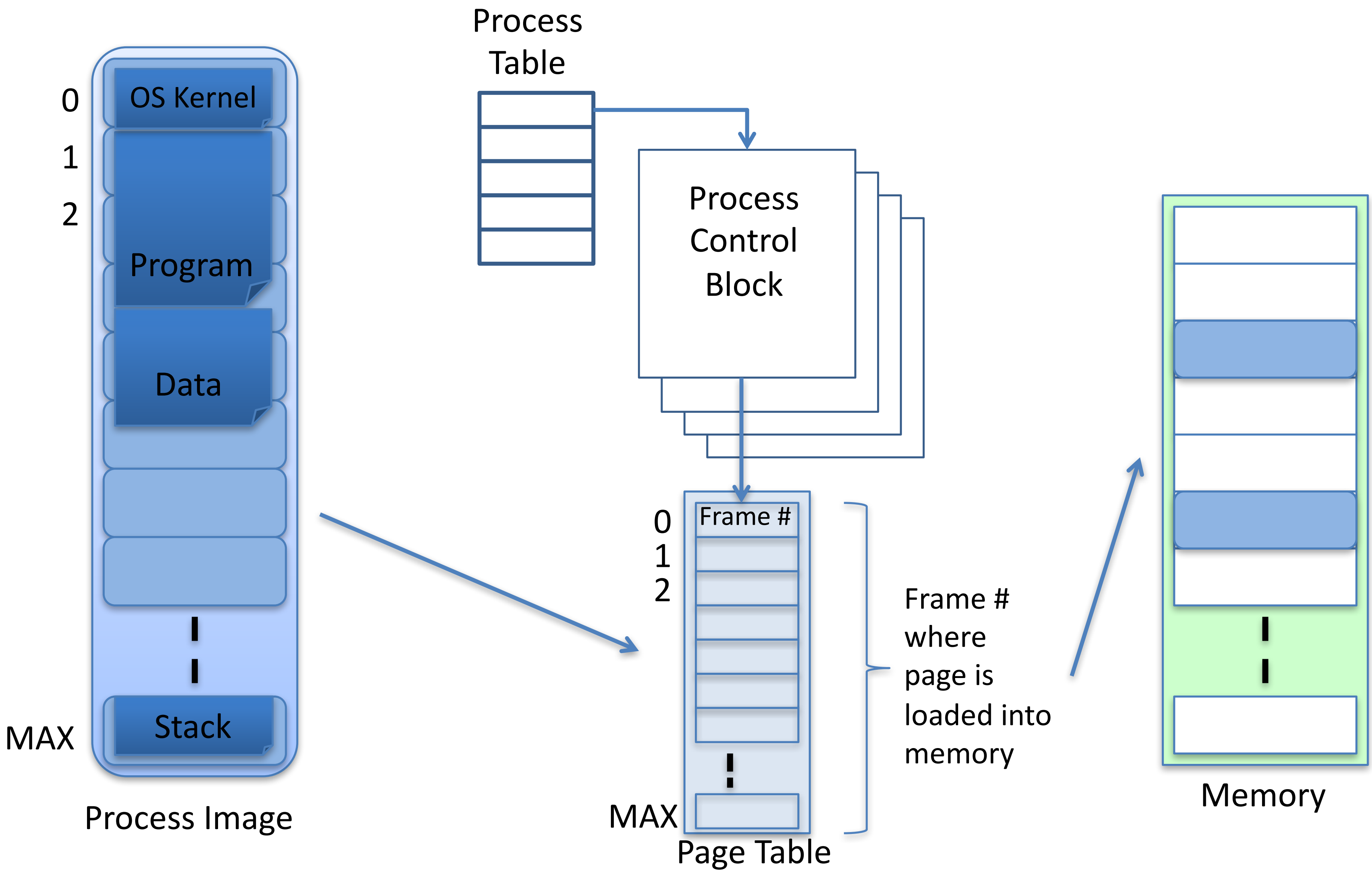
Memory Management

CS3026 Operating Systems

Lecture 09

Page Table of a process



Page Table Management

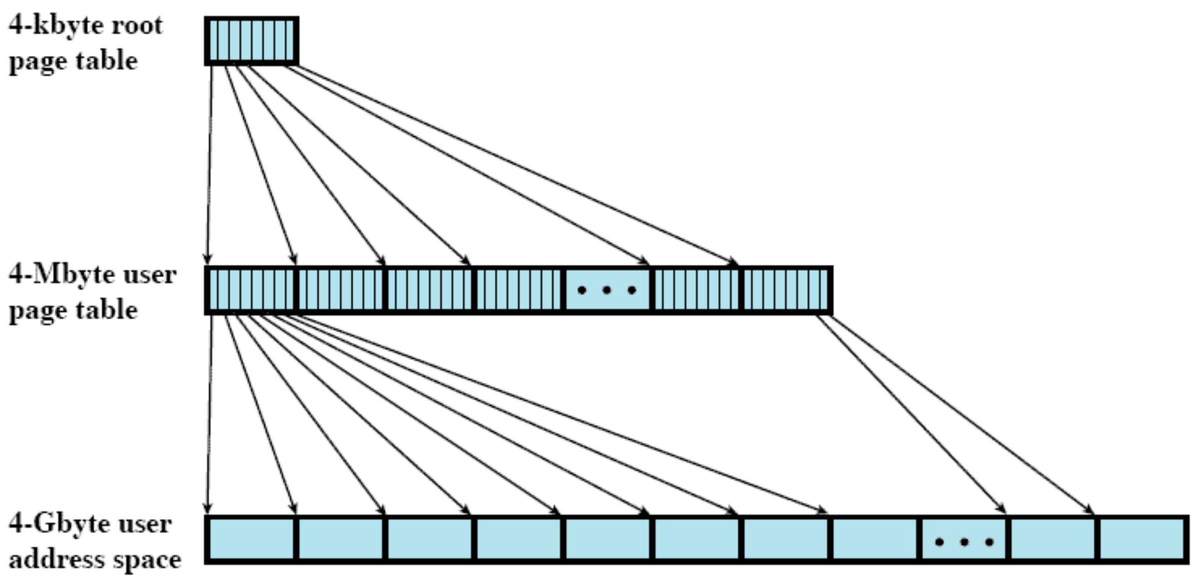
- Size of address space of a process influences the maximum size of the page table
  - Can be huge for address spaces 2^{32} (4GByte) or 2^{64} (16 Exabytes!)
  - Example:
    - Address space: 32-bit logical address space, 4GB
    - Page size: 4kb (2^{12}), address space is composed of 2^{20} (2^{32} / 2^{12}) pages (1 Mio)
    - page table may contain up to 1 Mio entries (as many as the process needs)
    - If each entry in page table is 4 bytes (32-bit), then we may need up to 4MB of physical memory just to hold this page table !
- We many not be able to allocate a contiguous area of physical memory
- The page table itself may be subject to paging

Problems with Page Tables

- Page tables themselves are held in virtual memory and are subject to paging
  - When a process is executing, those parts that are currently needed have to be in memory
    - Which part of the page table?
  - Possibly two page faults: reference to page not in memory, reference to page table part not in memory
- Approaches
  - Hierarchical page tables
  - Inverted page tables
  - Hashed page tables
  - Hardware solution: use of associative memory to cache page table

Hierarchical Page Tables

- Multilevel Page Table
  - Page table may not fit into a single page, may occupy multiple pages itself
    - Create hierarchy of sub-tables
  - Extra page faults to load pages that contain page table information



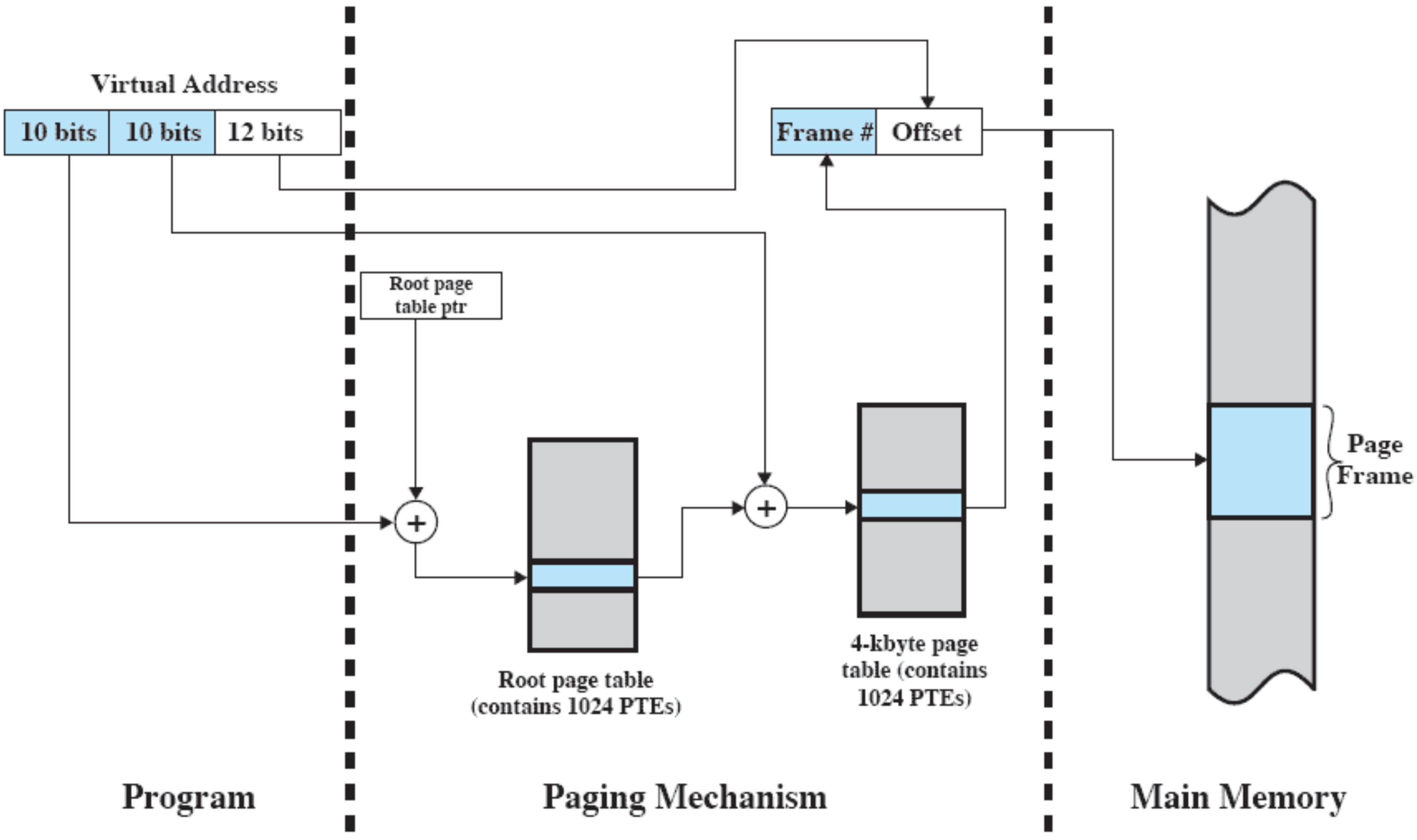
Hierarchical Page Tables

- If we assume a hierarchy of two tables:
  - Virtual address split into three parts
    - Root table index
    - Page table index
    - Page offset
- Consider a 32-bit memory address
  - 10 bits for root table: has $2^{10} = 1024$ entries
  - 10 bits for page table: has $2^{10} = 1024$ entries
  - 4kB per page: we need 12 bits ($2^{12} = 4096$) to address a location within the page

			Page offset
32-bit Virtual Address	10 bit	10 bit	12 bit
	Root Index	Page table index	

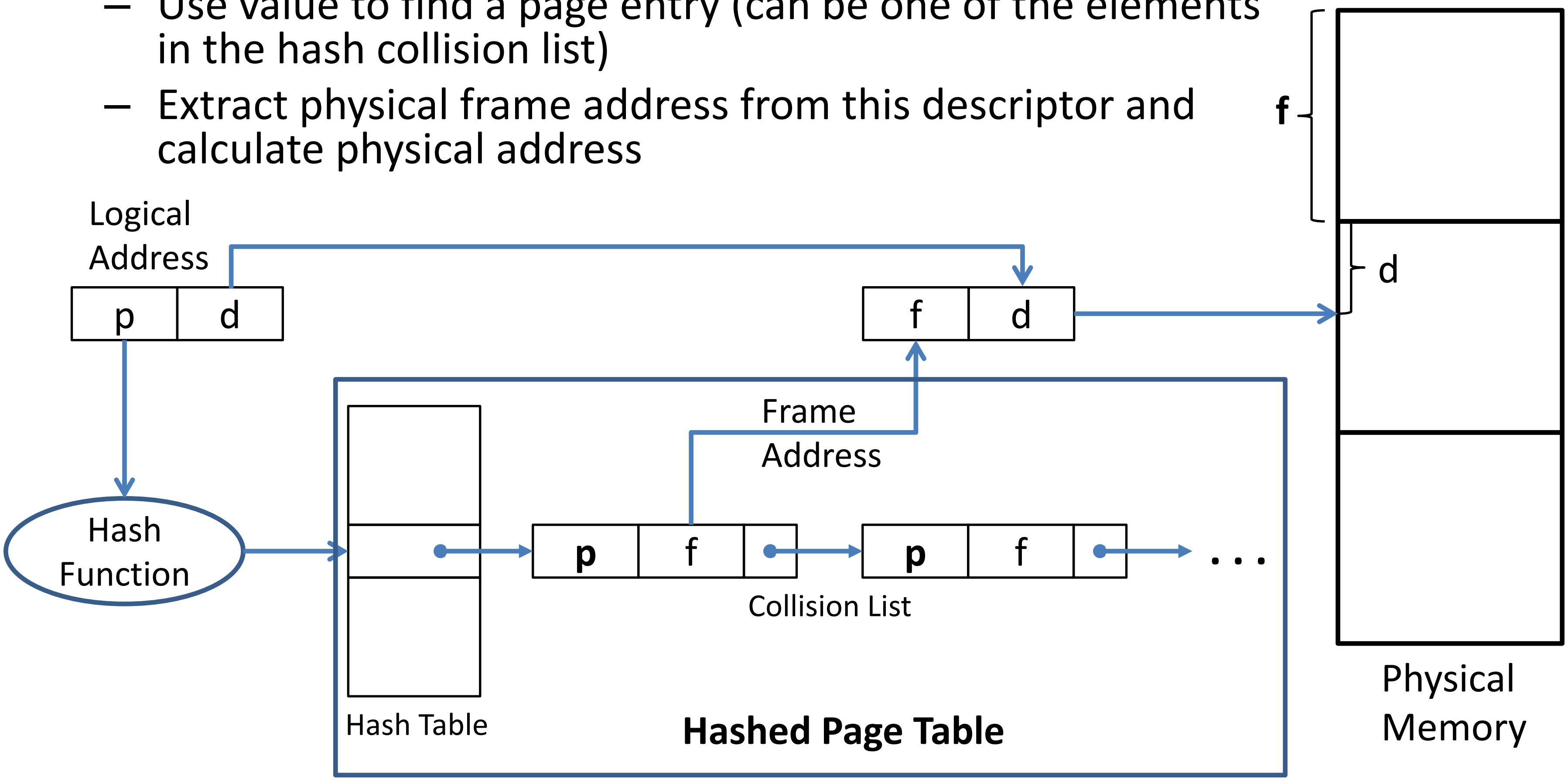
Hierarchical Page Table

Address Translation



Hashed Page Tables

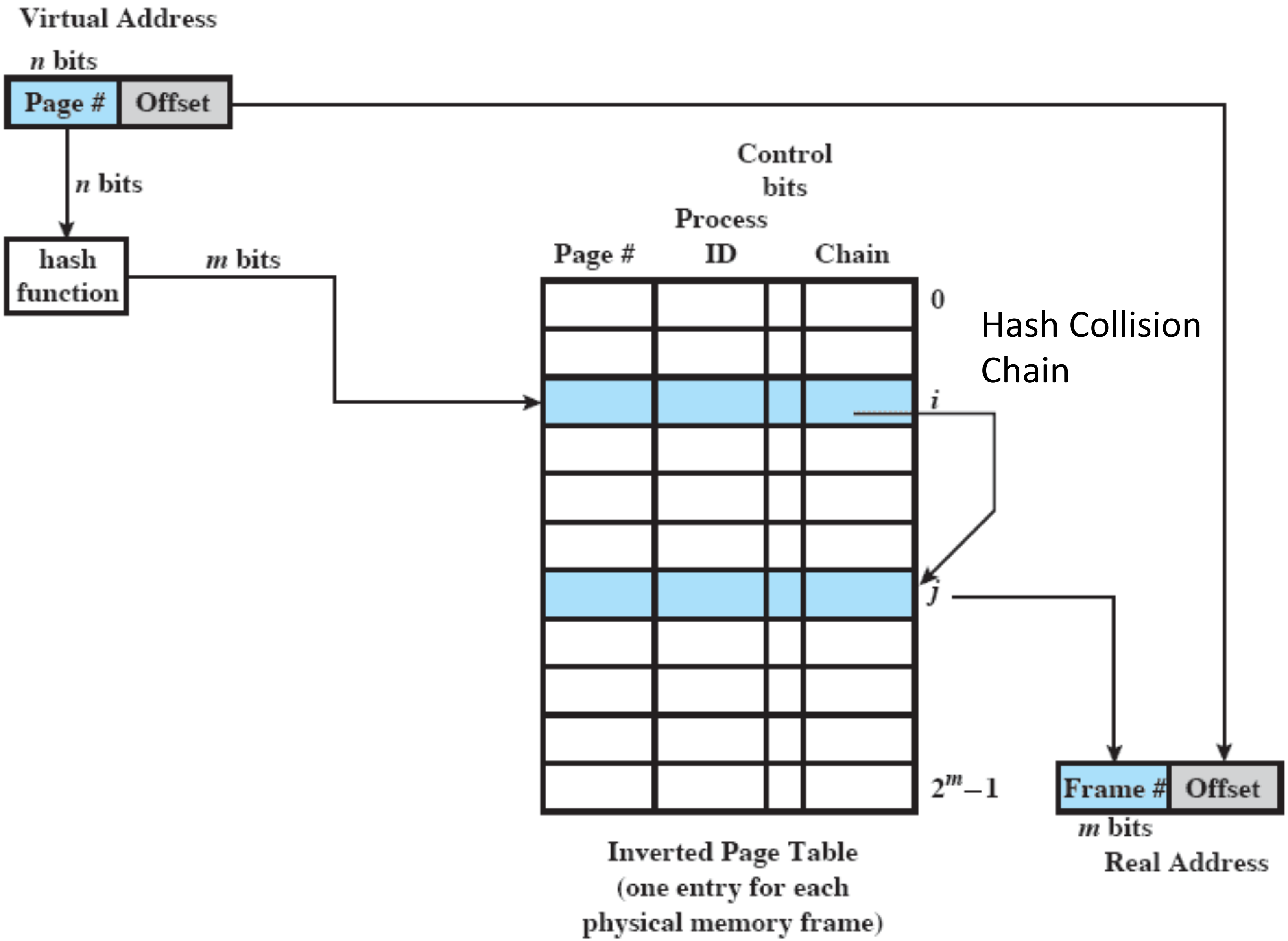
- Handling address spaces larger than 32-bit
- Use a hashed page table
  - Calculate hash value from page number
  - Use value to find a page entry (can be one of the elements in the hash collision list)
  - Extract physical frame address from this descriptor and calculate physical address



Inverted Page Table

- An inverted page table has one entry per memory **frame**
  - Depends only on physical memory size
  - For physical memory with 2^m frames , table is of size 2^m, i^{th} entry refers to frame i
- Only one page (frame) table for all processes
  - Has a fixed size, size of physical memory is known at startup of system
- Table entry records
  - Process ID, Page number, Offset
- Table is a hash table
  - Calculate hash value from page number
  - Hash value is index into page table, compare process id and page number, follow collision chain until match
  - Index of found entry is the required frame number

Inverted Page Table



TLB Translation Lookaside Buffer

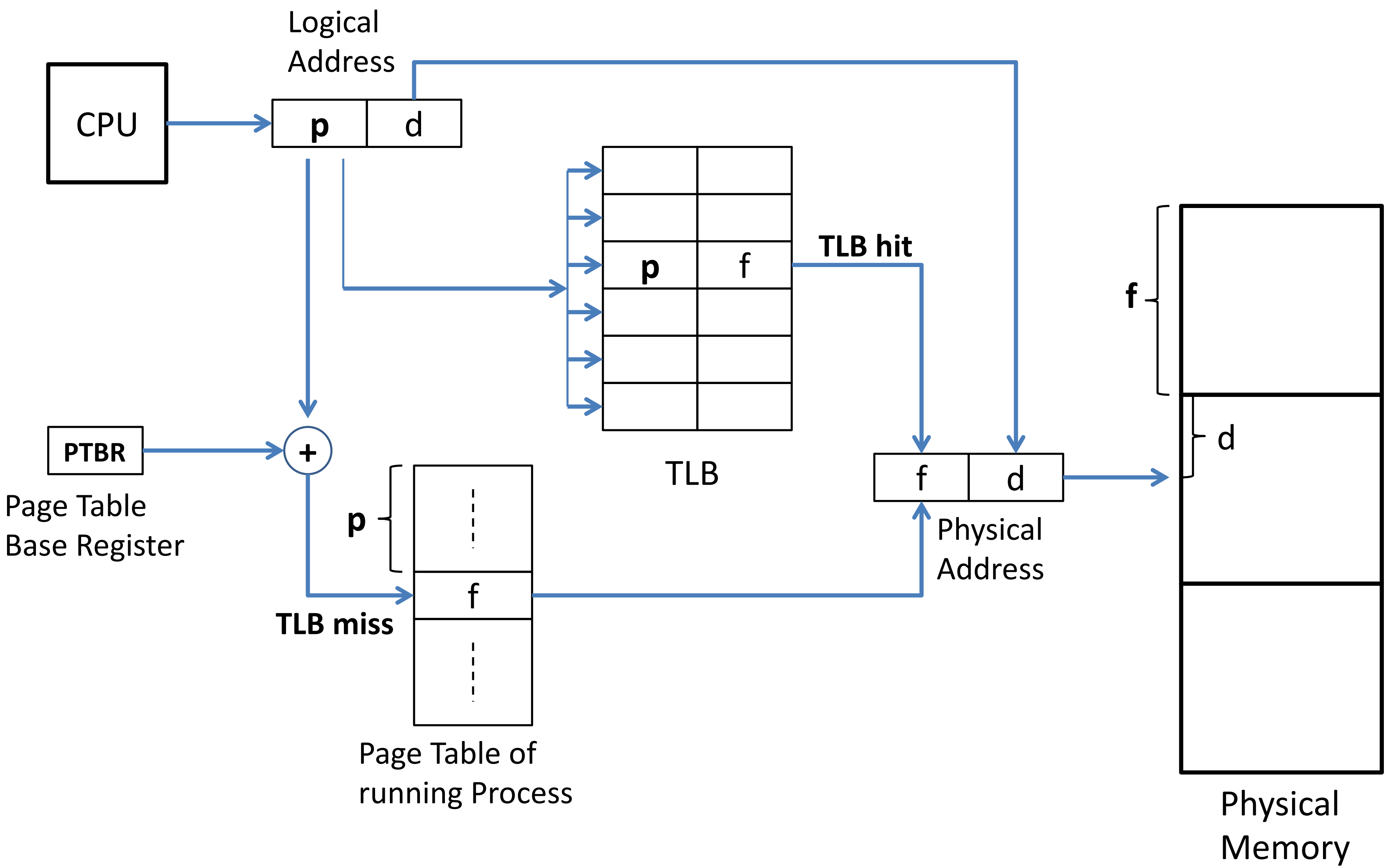
Paging Hardware Support

- Problem with page table in main memory
  - In order to access a particular physical memory location, processor has to access memory twice
    - First, access page table via the page table base register to get the frame address
    - Second, calculate the actual physical address and access physical memory location (load or store data)
  - Memory access slows down 50% !
- Standard solution
  - Translation Look-aside Buffer (TLB): special small and fast-lookup hardware cache

TLB Translation Lookaside Buffer

- Translation Look-aside Buffer (TLB):
  - special small and fast-lookup hardware cache
  - Is special associative cache memory
  - Is associative memory, holds part of the page table (those pages the processor will access “most likely”)
  - Has to be reset and reloaded at each context switch

Translation Lookaside Buffer (TLB)



Translation Lookaside Buffer (TLB)

- Using the TLB
  - CPU generates logical address
  - "**TLB hit**": If lookup of page number in TLB is successful:
    - very fast lookup of corresponding frame number
  - "**TLB miss**": If lookup of page number in TLB is not successful:
    - find frame number in page table, slow lookup of frame numbers
    - Page number and found frame are added to the TLB (potentially replacing other entries according to a replacement policy)
- TLB must be loaded with process-specific page table entries at each context switch

Virtual Memory – Design Concepts

Basic Design Concepts

- Fetch Policy
  - Demand paging
  - Prepaging
- Placement Policy
- Replacement Policy
  - Least recently used (LRU)
  - First-In-First-Out (FIFO)
  - Clock algorithm
  - Page buffering
- Resident Set Management
  - Size manipulation
  - Replacement Scope
- Cleaning Policy
  - Demand-based
  - Precleaning
- Load control
  - Degree of multiprogramming

Virtual Memory Concepts

- Page fault
  - Exception due to memory access: occurs if program references address in virtual memory on a page that is not currently loaded
- Demand paging
  - When a page fault occurs, pages containing the required memory location will be loaded on demand
- Thrashing
  - Frequent occurrence of page faults
  - May occur if Resident Set is too small
  - OS occupied mainly with I/O operations to load pages

Fetch Policy

- Demand Paging
  - Page is loaded on demand, if an address contained in the page is referenced
  - We can start a process without any pages loaded
    - Many page faults at process start, resident set becomes more stable over time
- Prepaging
  - Load more pages in advance in anticipation of future references
  - Principle of Locality
  - Disadvantage: we may load pages we may never need

Principle of Locality

- Despite the elaborate and time-consuming procedure to handle page faults, virtual memory management is efficient
- Principle of Locality

  "**References to memory locations within a program tend to cluster**"

  – if there is an access to a memory location, the next access is, in all likelihood, very near to the previous one
  – Loading one page may satisfy subsequent memory access, as all required memory locations may be contained within this page, no loading of page necessary

Demand Paging

- Demand Paging
  - Only brings a page into main memory when a reference is made to it
  - We can start process without any pages loaded (only the PCB is in memory)
    - the program counter referring to the first instruction already leads to page fault and the loading of a page
    - With this policy, a process will be slow at startup due to many page faults
- Principle of locality
  - as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults are expected to drop to a very low level

Prepaging

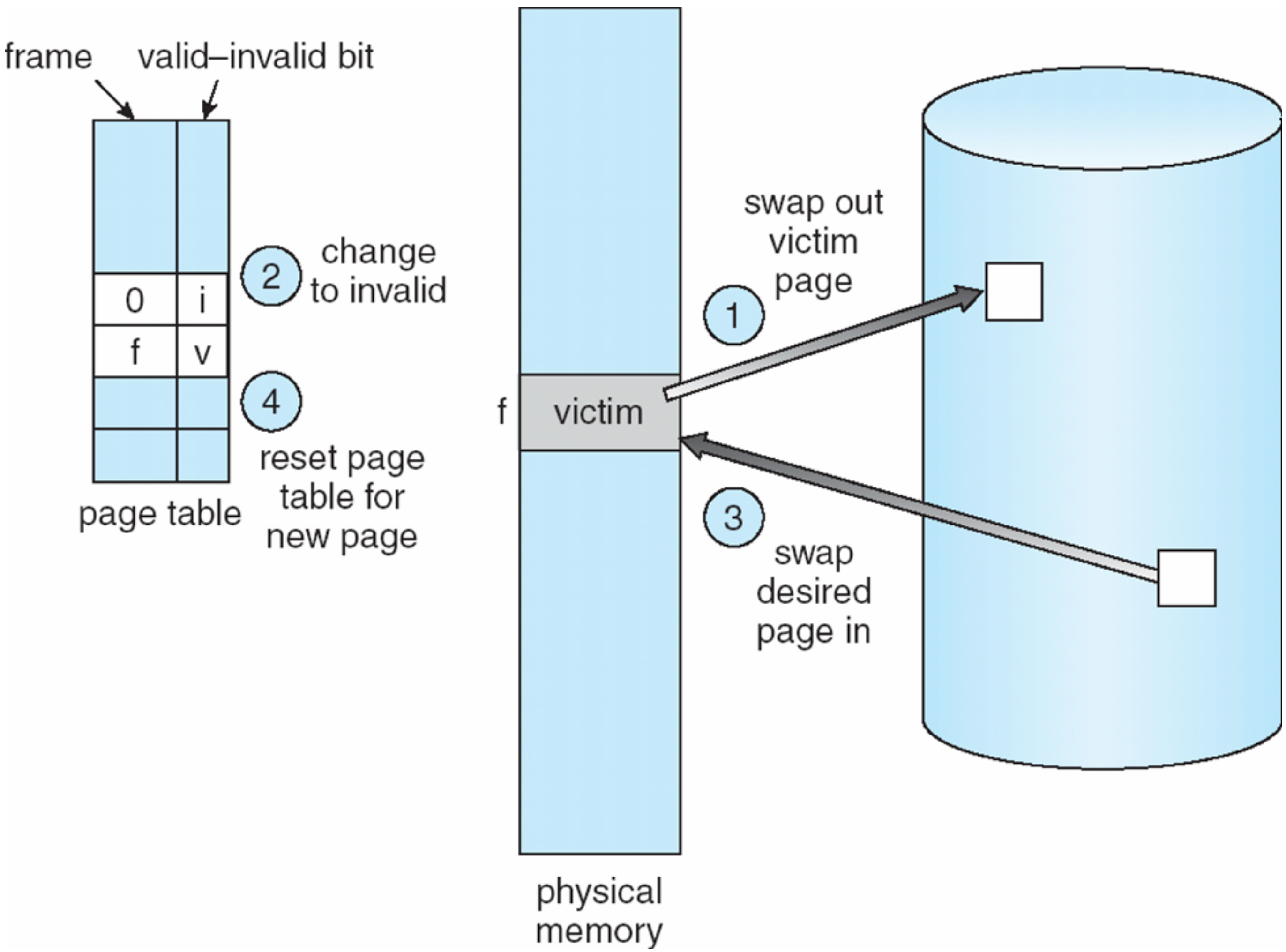
- Exploits characteristics of secondary memory (hard disk)
- It is more efficient to load the demanded page together with neighbouring pages that reside in a contiguous space on the disk
  - According to the principle of locality, it can be expected that these other pages may be referenced in the near future
  - Load more pages than needed – reduce seek times and rotational latency on disk for subsequent page references
- May be ineffective if the extra pages loaded are actually not needed

Page Replacement Policies

Page Replacement

- Page replacement becomes necessary when no physical frames are free for the demanded page
  - A so-called "victim" page has to be selected from the resident set and replaced with the demanded page
- Two nominal I/O operations
  - Write victim to disk
  - Read demanded page from disk
- Introduce a "dirty" bit (modify bit) for each page
  - Indicates whether page has been modified since last load
  - Write operation only necessary if page was modified

Page Replacement



Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in
- A process has a particular allowance for loaded pages
  - Resident set
- If maximal size of resident set is reached:
  - Which page to replace?
- Replacement strategies
  - Local: replace pages within process
  - Global: replace pages across processes

Replacement Policy

- Which page should be replaced?
  - Page removed should be the page least likely to be referenced in the near future
- How is that determined?
  - Most policies predict the future behaviour on the basis of past behaviour
- Metrics
  - Minimize page-fault rate
    - Avoid replacement of pages that are needed for the next memory references

Page Replacement Algorithms

- The optimal policy (OPT) as a benchmark

- Algorithms used
  - Least recently used (LRU)
  - First-In-First-Out (FIFO)
  - Clock Algorithm

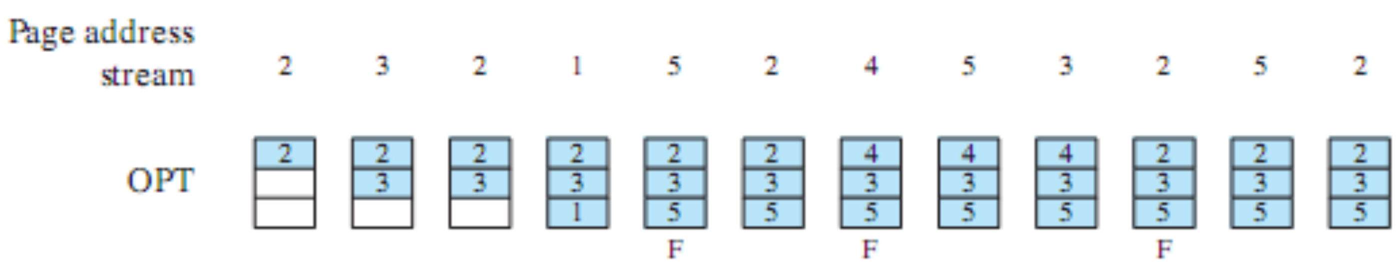
Optimal Policy OPT

- The theoretically optimal replacement policy
  – Look into the future: select the page for which the time until it is referenced again is the longest among all other pages
- Is not a realistic strategy
  – This policy is impossible to implement, as it would require an operating system to have perfect knowledge about all future events
- But we can use it as a **benchmark**
  – Gives us the minimum number of page faults possible
- How to do that
  – Record a finite sequence of page references
  – Replay this sequence: in hindsight, we know now which page can be replaced

Optimal Policy OPT

- Assumption
  - 3 memory frames available
  - Process has 5 pages
  - When process is executed, the following sequence of page references occurs (called a page reference string):

Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2



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

OPT policy results in 3 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 6)

Optimal Policy OPT

Reference to page 2:
- free frame used

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2											
Frame2												
Frame3												
Page fault												

Reference to page 3:
- free frame used

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2										
Frame2		3										
Frame3												
Page fault												

Reference to page 2:
- Page already loaded

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2									
Frame2		3	3									
Frame3												
Page fault												

Optimal Policy OPT

Reference to page 1:
- free frame used

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2								
Frame2		3	3	3								
Frame3				1								
Page fault												

Page 1 not referenced any more

Reference to page 5:
- **We replace page 1 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2							
Frame2		3	3	3	3							
Frame3				1	5							
Page fault					F							

- Why is page 1 replaced?
  - This is a demonstration of the only theoretically possible optimal strategy (OPT)
  - If we have a perfect oracle that tells us the future, we would know that page 1 is not referenced any more in this particular scenario, therefore it is the least important page and can be replaced

Optimal Policy OPT

Reference to page 2:
- Page already loaded

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2						
Frame2		3	3	3	3	3						
Frame3				1	5	5						
Page fault					F							

Reference to page 4:
- **We replace page 2 in Frame 1**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	4					
Frame2		3	3	3	3	3	3					
Frame3				1	5	5	5					
Page fault					F		F					

- Why replace page 2 with page 4?
  - As we assume a perfect oracle of the future, we know that pages 5 and 3 are referenced immediately after this step and that choosing page 2 will result in a minimum number of page faults

Optimal Policy OPT

Reference to page 2:
- **We replace page 4 in Frame 1**

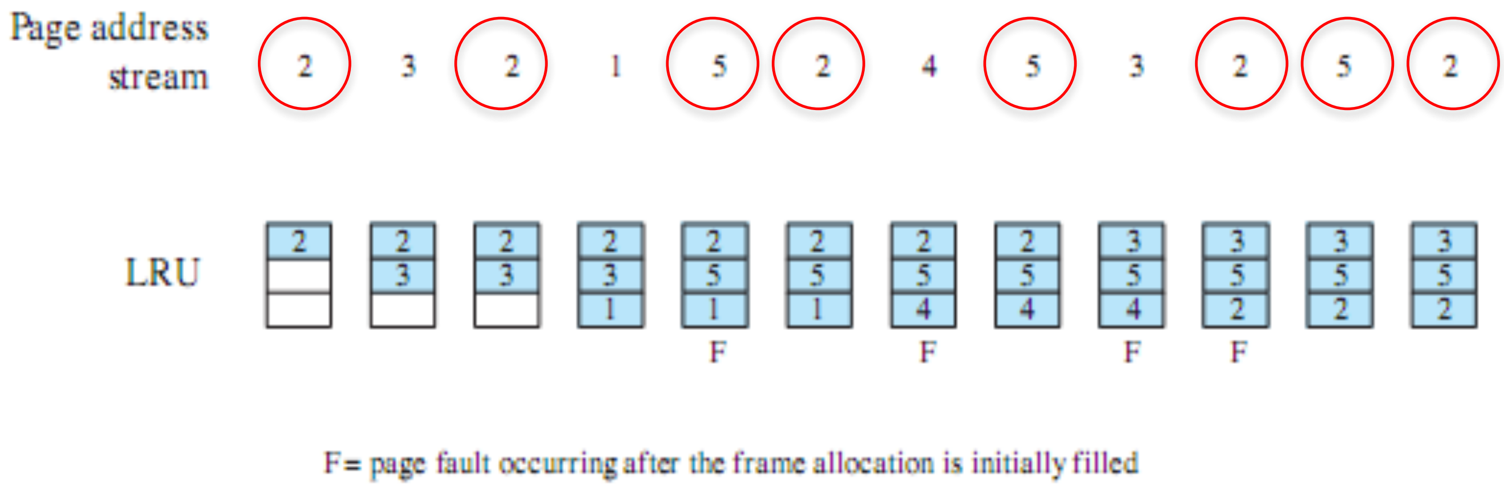
Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	4	4	4	2	2	2
Frame2		3	3	3	3	3	3	3	3	3	3	3
Frame3				1	5	5	5	5	5	5	5	5
Page fault					F		F			F		

- In this particular example, the "optimal" policy would create 3 replacement page faults
- In reality, we cannot know the future, therefore we may have more page faults
- OPT is our "benchmark"

Least Recently Used (LRU)

- Idea: past experience gives us a guess of future behaviour
- Replaces “least recently used” page: replace the page that has not been referenced for the longest time
  - “Least recently used” page should be the page least likely to be referenced in the near future
- Pages that are more frequently used remain in memory
- Good behaviour in terms of page faults
  - Almost as good as OPT
- Difficult to implement
  - Search for oldest page may be costly
  - One approach would be to tag each page with the time of the last reference
  - Requires great deal of overhead to manage

LRU Policy



- The LRU policy does almost as well as the theoretical optimal policy OPT
  - LRU recognises that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not
- We have to look into the past to guess what trend future page references may follow

LRU Policy

References to pages 1, 2 and 3:
- free frames used

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2								
Frame2		3	3	3								
Frame3				1								
Page fault												

Page 3 least recently used

Reference to page 5:
- **We replace page 3 in Frame 2**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2							
Frame2		3	3	3	5							
Frame3				1	1							
Page fault					F							

- Why is page 3 replaced?
  - With the LRU (least recently used) policy, we look into the past
    - The least recently used page was page 3, therefore it is replaced
    - This is different to before, where (with a perfect oracle) we replaced page 1

LRU Policy

Reference to pages 4:
- **We replace page 1 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2					
Frame2		3	3	3	5	5	5					
Frame3				1	1	1	4					
Page fault					F		F					

Reference to page 3:
- **We replace page 2 in Frame 1**

Reference to page 2:
- **We replace page 4 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2	2	3	3	3	3
Frame2		3	3	3	5	5	5	5	5	5	5	5
Frame3				1	1	1	4	4	4	2	2	2
Page fault					F		F		F	F		

- With LRU, we replace page 2 with page 3, and immediately afterwards, we need page 2 again
- LRU is not able to detect this situation
- However, it is a strategy that comes close to OPT

LRU Implementation

- How to identify the "least recently used" page?
  - Time stamp ("time-of-use") associated with each entry in page table
    - Whenever page is accessed, update timestamp
    - Too costly
      - For each page fault we have to search through the table to find the LRU page
  - Stack of page numbers
    - Whenever a page is referenced, it is moved to the top of the stack, LRU always at the bottom of stack
- Problem
  - Each memory reference results in an update of page information
  - Slowdown of operations unacceptable

LRU Implementation

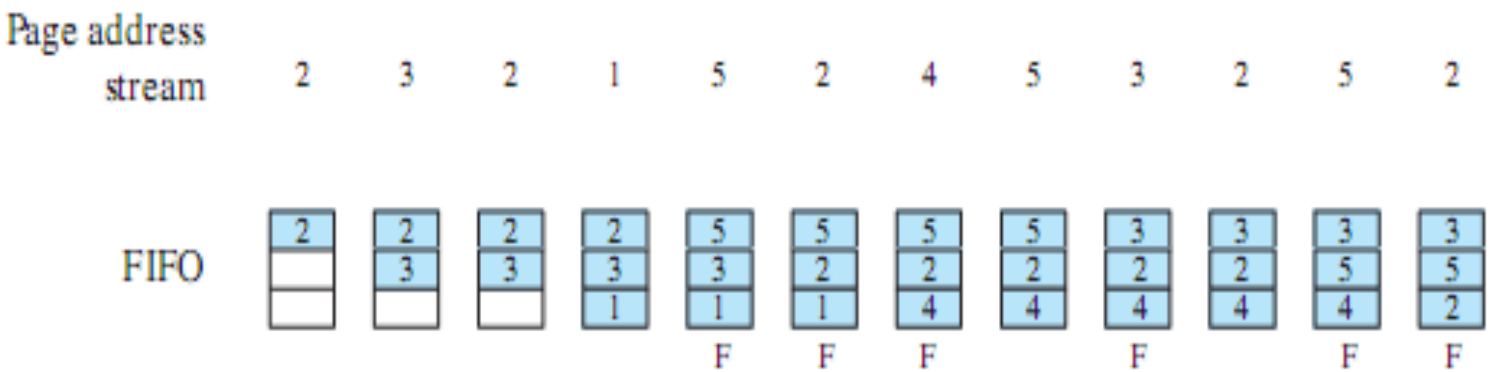
- Many systems use a “reference bit”
  - Initially set to 0
  - set to 1 as soon as a page is referenced
  - Periodically, operating system may clear reference bit to distinguish recently referenced pages
- Does not record in which order pages were referenced (which is the “oldest”?)
- Can be used to implement an approximation of a LRU strategy

FIFO and Second Chance Algorithm

First-In-First-Out (FIFO)

- First in First out
  - The first page that was loaded is also the first to be discarded – FIFO ordering of pages
  - Oldest page is discarded
- Problem
  - Oldest page may be the most frequently used
- Is the simplest replacement policy to implement
  - Treats memory frames allocated to a process as a circular buffer
  - Pages are replaced in a round-robin style
- Does not indicate how heavily a page is actually used

FIFO Example

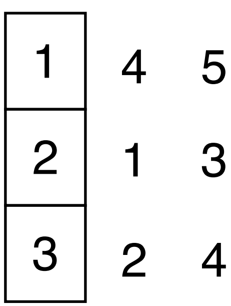


F= page fault occurring after the frame allocation is initially filled

- FIFO policy results in 6 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 9)

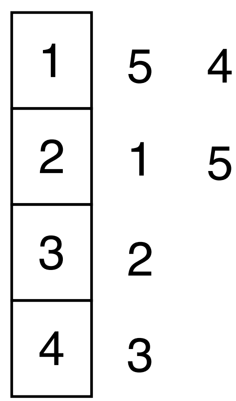
FIFO: Belady's Anomaly

- Problem of FIFO
  - Does not indicate how heavily a page is actually used
- Possible solution: increase number of physical frames
  - FIFO shows Belady's anomaly: the page-fault rate increases, when the number of frames is increased
- Example
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 frames vs 4 frames:



9 total page faults

3 Frames, 5 pages



10 total page faults

4 Frames, 5 pages

FIFO Page Replacement

- Methods to determine the “age” of a page
  - Associates a time stamp with each page
    - Records the time the page was loaded into memory
  - Create a FIFO queue that holds all pages loaded in memory
    - Remove the page which is first in this queue
    - Insert the new page at the tail of this queue

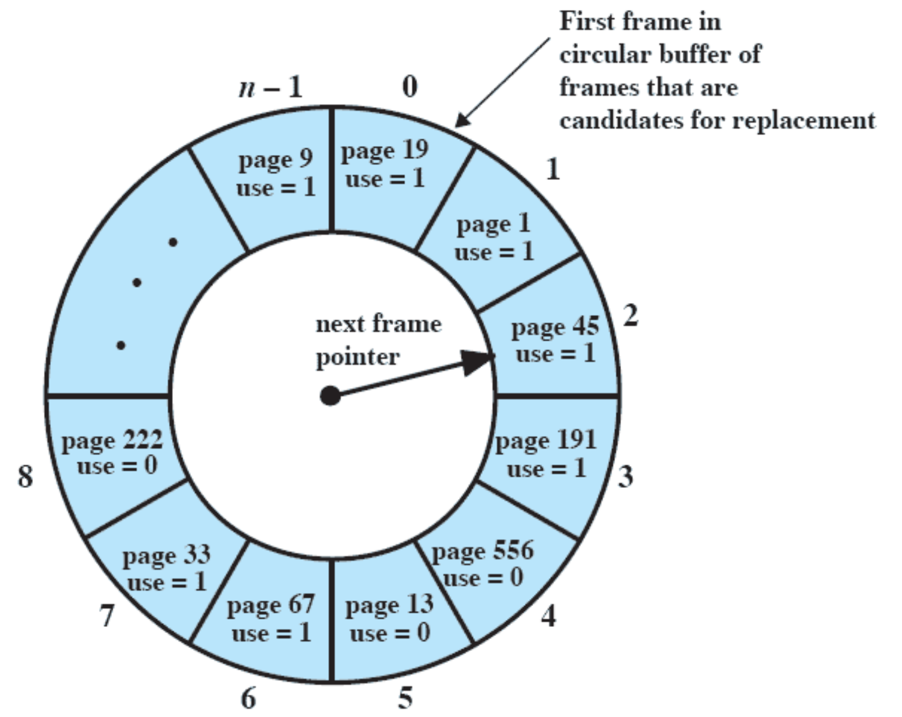
Second Chance Algorithm

- Is a modification of the FIFO replacement algorithm that tries to avoid replacing recently used pages
- Loaded pages are ordered in a FIFO list: “oldest” page first in list
- Each page has reference bit: record recent access by process
- When oldest page selected for replacement, the reference bit is inspected
  - If value is 0: replace page
  - If value is 1: page gets a “second chance”:
    - Clear reference bit, set to 0
    - Move page to tail of FIFO list, becomes youngest page
    - Check next-oldest page
- A page is given a second chance:
  - will move to the tail of the FIFO queue and becomes the youngest page
- Worst case:
  - All reference bits == 1, degenerates into pure FIFO

Clock Policy

Implementation of Second-Chance Algorithm

- The set of frames is considered as laid out like a circular buffer
  - contains a FIFO list of pages
  - Can be circulated in a round-robin fashion
- Each page has a reference bit
  - When a page is first referenced (a page fault that leads to its load), the use bit of the frame is set to 1
- Each subsequent reference to this page again sets/overwrites this bit to 1



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

Clock Algorithm

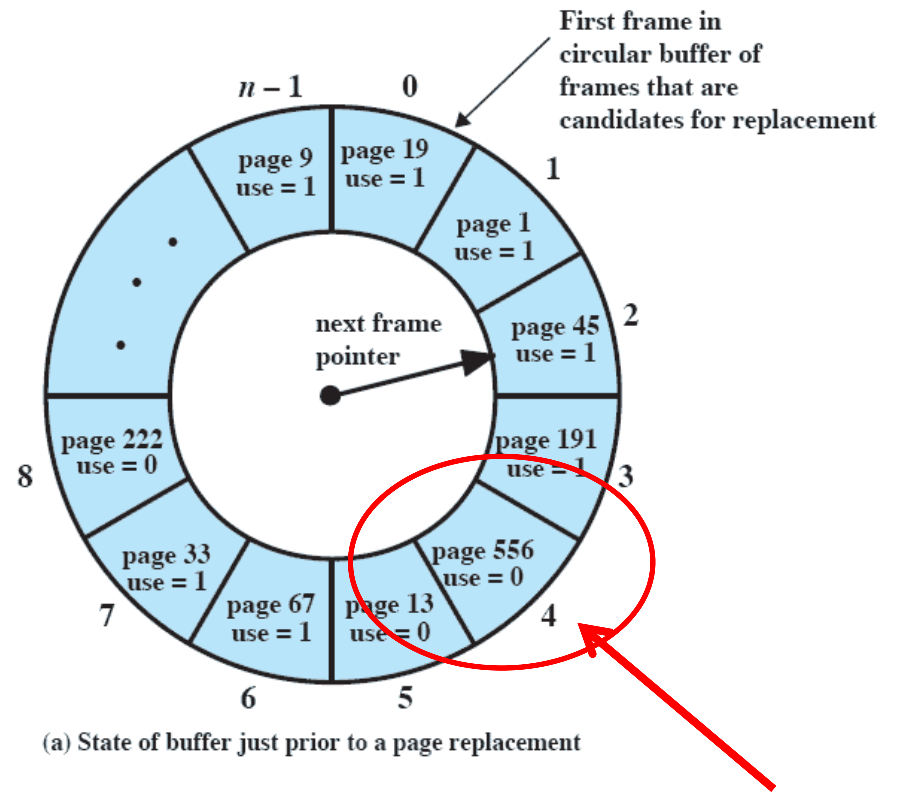
Free Frames Available

- Initially, all frames in circular buffer are empty
  - Each frame entry has a reference bit (also called a “use” bit)
- Frame pointer: is regarded the “clock hand”, is called the “next frame” pointer
- Procedure
  - Occurrence of page fault:
    - Progress position pointer to first unused frame (use bit == 0)
    - During this progression: set use bit of visited frame entries to 0
  - Load page into free frame, set use bit to 1
  - Move position pointer to next frame
- When the circular buffer is filled with pages, the “next frame” pointer has made one full circle
  - it will be back at the first page loaded
    - points to the “oldest” page

Clock Policy

Find the right Page to be replaced

- Finding a page to replace
  - The "next frame pointer" advances from frame to frame in this circular "frame buffer"
  - As long as the "next frame pointer" encounters a page in a the frame with the reference bit set to 1, it is set it to 0 and moves on – it gives the page a ***second chance***
  - The first page encountered with reference bit == 0 is replaced
  - Page is replaced, pointer is advanced to *next* page



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

Clock Policy
Before and after Page Replacement

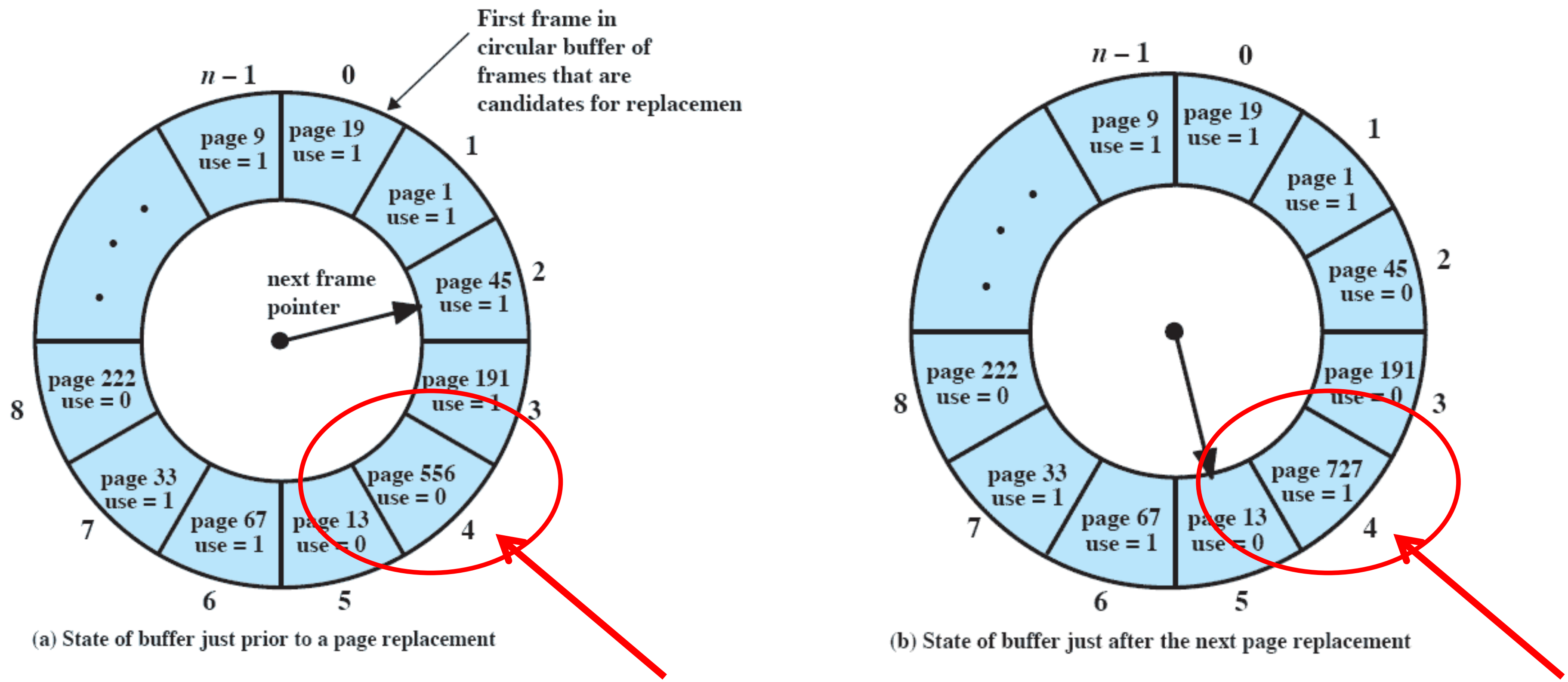


Figure 8.16 Example of Clock Policy Operation

Page-Buffering

- Avoid unnecessary read and write operations
- A replaced page is not lost, but assigned to one of two lists
  - Free page list
    - Move unmodified replaced page into this memory area
    - When page is needed again, its content is still held in memory, no I/O needed, can be reused immediately
  - Modified page list
    - Move replaced page that was modified, into this memory area
    - Pages are written to disk in clusters, saves I/O