Memory Management

Virtual Memory Management

Lecture 9

Overview

- **Virtual Memory:**
  - Motivation
  - Basic Idea
  - Page Fault
- **Common Applications of Virtual Memory:**
  - Demand Paging
- **Aspects of Virtual Memory Management:**
  - Page Table Structure
  - Page Replacement Algorithms
  - Frame Allocation

Virtual Memory: Motivation

- Our last assumption of memory usage:
  - Physical memory is large enough to hold one or more process logical memory space completely

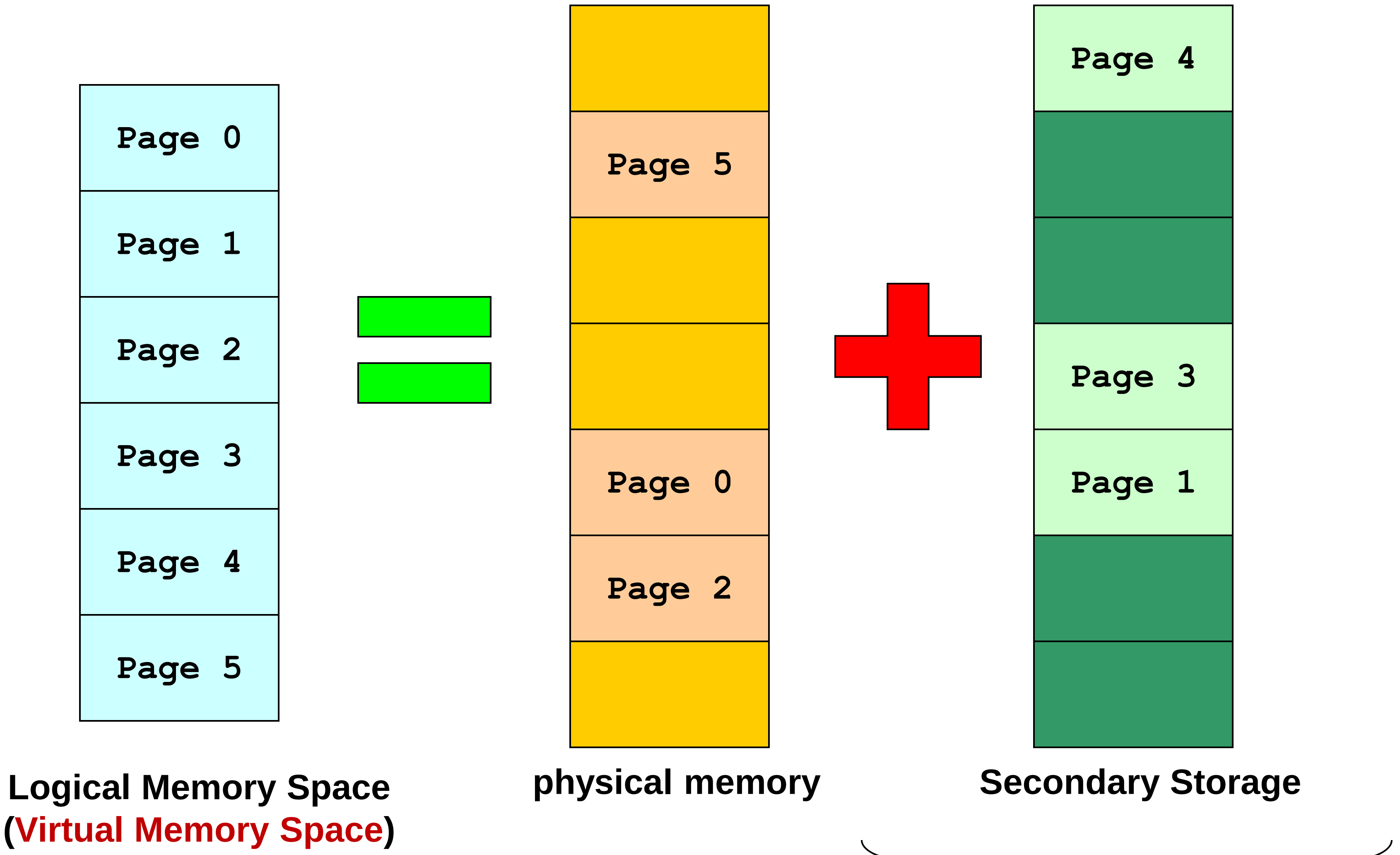
- This assumption is too restrictive:
  - What if the logical memory space of process is >> than physical memory?

  - What if the same program is executed on a computer with lesser physical memory?

Virtual Memory: Basic Idea

- Observation:
  - The logical memory space of a process >> physical memory
  - Secondary storage has much larger capacity compared to physical memory
- Basic Idea:
  - Split the logical address space into pages:
    - Some pages reside in physical memory
    - **Other are stored on secondary storage**

Virtual Memory: Paging Illustration

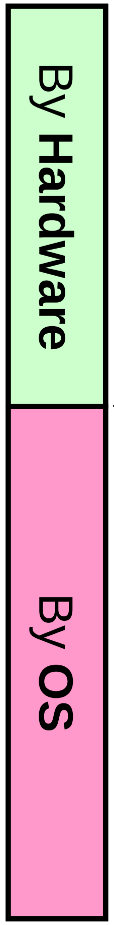


Extension of the basic paging scheme, i.e., the "virtual" part

Extended Paging Scheme

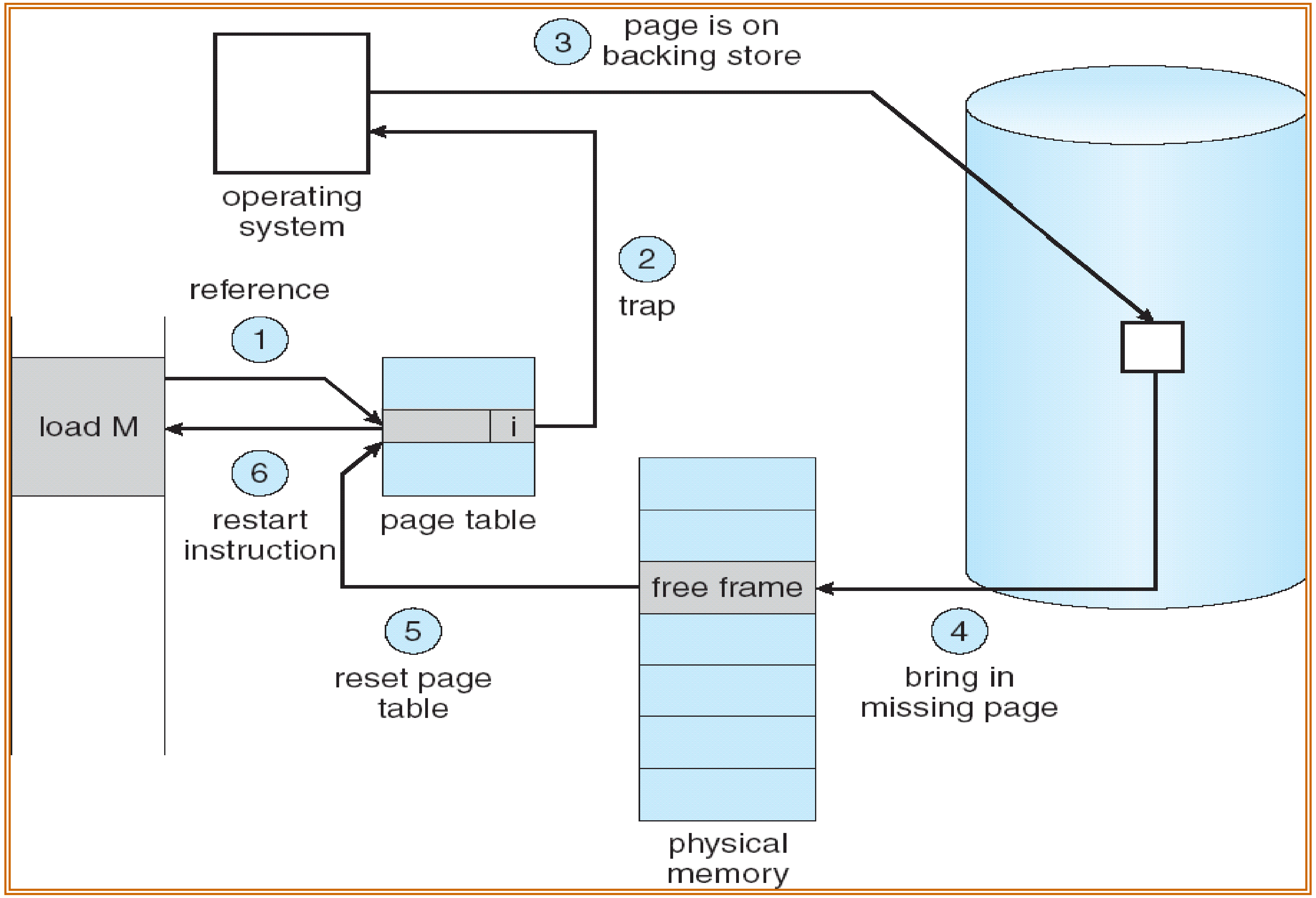
- Basic idea remains unchanged:
  - Use page table to translate **virtual** address to physical address
- New addition:
  - To distinguish between two pages types
    - **memory resident** (pages in physical memory)
    - non-memory resident (pages in secondary storage)
    - ➔ Use a (***is memory resident?***) bit in page table entry
  - CPU can only access memory resident pages:
    - **Page Fault**: When CPU tries to access non-memory resident page
    - OS need to bring a non-memory resident page into physical memory

Accessing Page X: General Steps



1. Check **page table**:
   - Is page **X *memory resident***?
     - Yes: Access physical memory location. Done.
     - No: Continue to the next step
2. **Page Fault: Trap** to OS
3. Locate page X in secondary storage
4. Load page X into a physical memory frame
5. Update page table
6. Go to step 1 and retry

Virtual Memory Accessing: Illustration



Virtual Memory: Justification

- Observation:
  - Secondary Storage access time >> Physical memory access time
- If memory access results in page fault most of the time:
  - Non-memory resident pages need to be loaded
  - Known as **thrashing**
- How do we know that **thrashing** is **unlikely** to happen?
  - Related: How do we know that after a page is loaded, it is likely to be useful for future accesses?

Recap: **Locality Principles**

- *Most programs* exhibit these behaviors:
  - Most time are spent on a relatively small region of the code
  - Within a time period, accesses are made to a relatively small part of the data only
- Formalized as **locality principles**:
  - **Temporal Locality**:
    - Memory address which is referenced ***is likely to be referenced again***
  - **Spatial Locality**:
    - Memory addresses close to a referenced address is likely to be referenced

Virtual Memory and Locality Principle

- Exploiting **Temporal Locality:**
  - After a page is loaded to physical memory, it is likely to be accessed in near future
    - Cost of loading the page is **amortized**
- Exploiting **Spatial Locality:**
  - A page contains contiguous addresses that are likely to be accessed in near future
    - Later access to nearby addresses will not cause page fault
- However, there are always exceptions ☺
  - Programs that behave badly due to poor design or with malicious intention

Virtual Memory: Summary

- Completely separate logical memory addresses from physical memory
  - Amount of physical memory no longer restrict the size of logical memory space
- More efficient use of physical memory
  - Page currently not needed can be on secondary storage
- Allow more processes to reside in memory
  - Improve CPU utilization as there are more processes to choose to run

More on Virtual Memory Management

- More in-depth looks on several aspects:
  - Large page table with big logical memory space ➔ How to structure the page table for efficiency?
    - **Page Table Structures**
  - Each process has limited number of resident memory pages ➔ Which page should be replaced when needed?
    - **Page Replacement Algorithms**
  - Limited physical memory frames ➔ How to distribute among the processes?
    - **Frame Allocation Policies**

Waste not, want not

PAGE TABLE STRUCTURE

Page Table Structure

- Page table information is kept together with the process information and takes up physical memory space

- Modern computer systems provide huge logical memory space
  - 4GiB(32bit) was common, 8TiB or (much) more is possible now
  - Huge logical memory space ➔ Large number of pages
  - ➔ Each page has a page table entry ➔ Large page tables

- Problems with large page table
  - High overhead
  - Fragmented page table:
    - Page table occupies several memory pages

Page Table Structure: **Direct Paging**

- Direct Paging: keep all entries in a single table

- With 2^p pages in logical memory space
  - **p** bits to specify one unique page
  - 2^p page table entries (PTE), each contains:
    - physical frame number
    - additional information bits (valid/invalid, access right, etc.)

- Example:
  - Virtual Address: 32 bits, Page Size = 4KiB
  - P = 32 – 12 = 20
  - Size of PTE = 2 bytes
  - Page Table Size = 2^{20} * 2 bytes = 2MiB (!)

2-Level Paging: Basic Idea

- Observation:
  - Process may not use the entire virtual memory space ➔ Full page table is a waste!

- Basic Idea:
  - Split the full page table into regions
  - Only a few regions are used
    - As memory usage grows, new regions can be allocated
  - This idea is similar to split logical memory space into pages ☺
  - Need a directory to keep track of the regions
    - Analogues of page table ←➔ pages

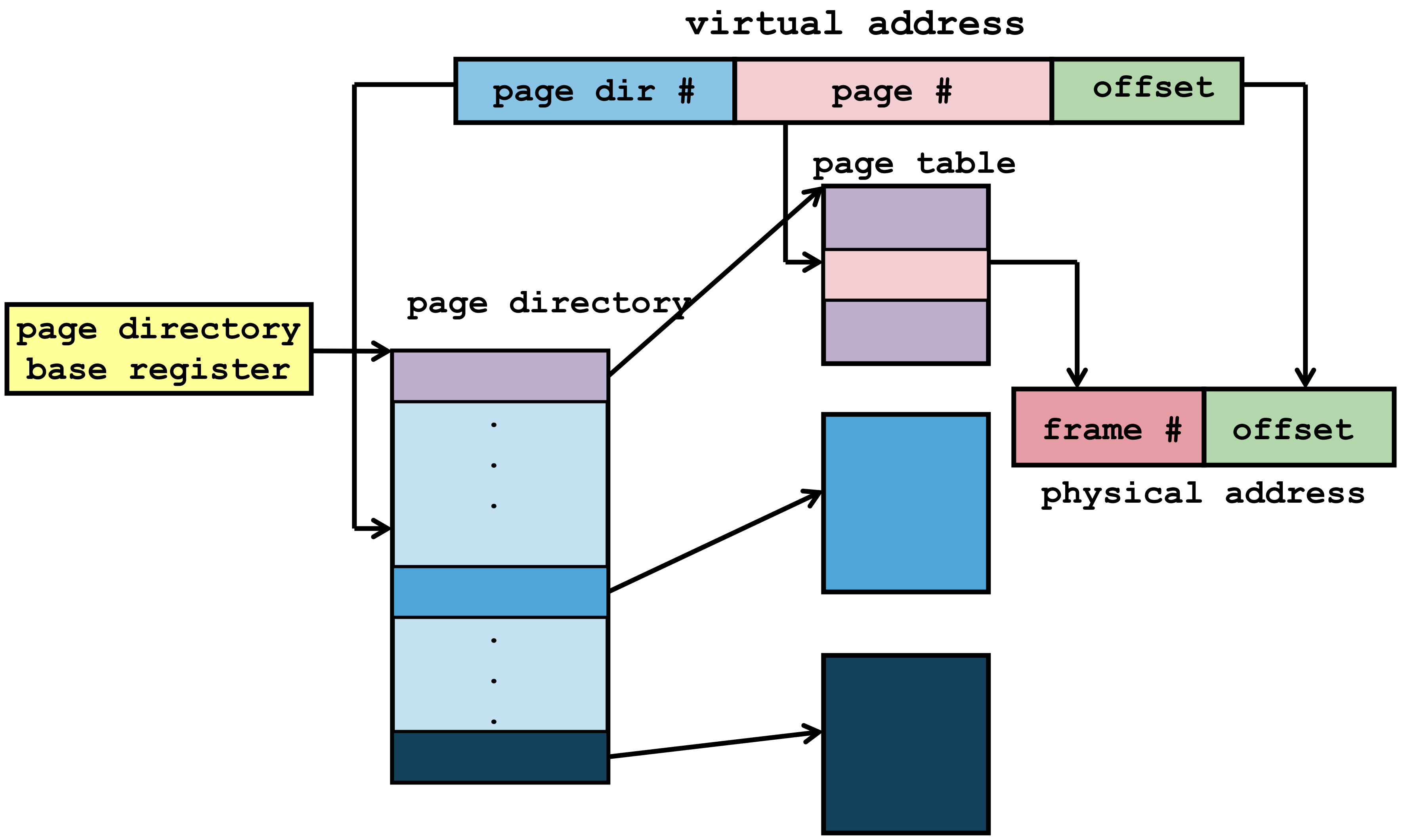
2-Level Paging: Description

- Split page table into *smaller page tables*
  - Each with a **page table number**
  - Each **page table size** is equal to the **page size**

- If the original page table has 2^P entries:
  - With 2^M smaller page tables, M bits are needed to uniquely identify one page table
  - Each smaller page table contains $2^{(P-M)}$ entries

- To keep track of the smaller page tables
  - A single **page directory** is needed
  - Page directory contains 2^M indices to locate each of the smaller page table

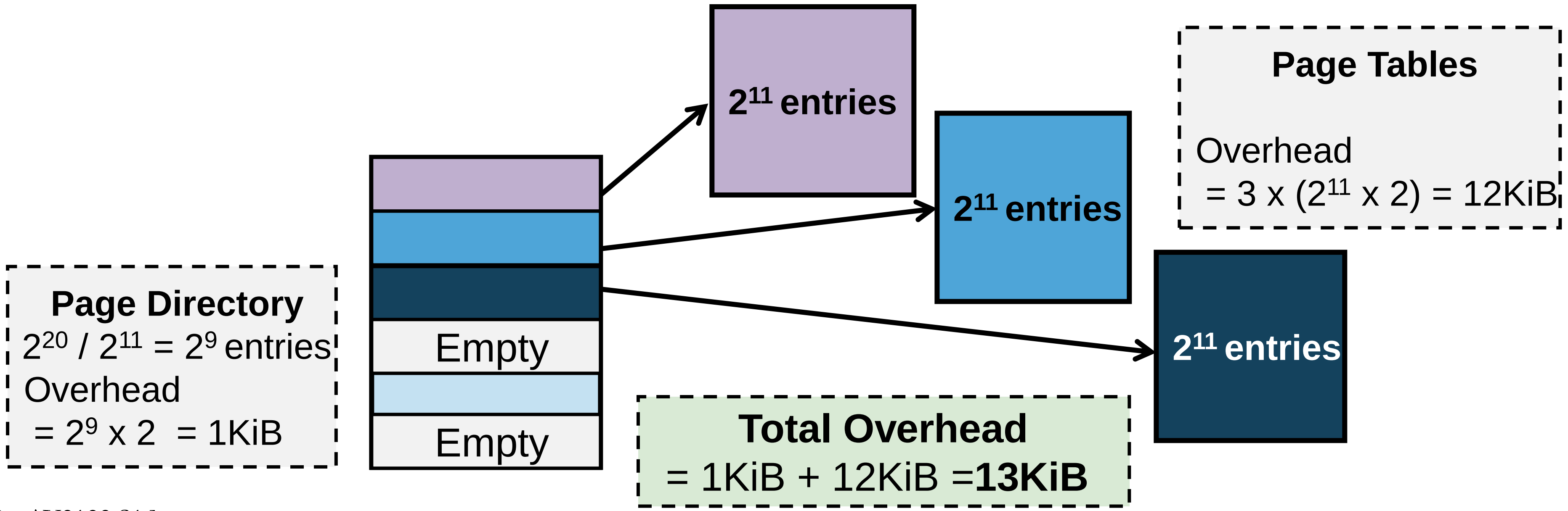
2-level Paging: Illustration



2-Level Paging: **Advantages**

- We can have empty entries in the page directory
  - ➔ The corresponding page tables need not be allocated!

- Using the same setting as the previous example:
  - Assume only **3** page tables are in use
  - Overhead = 1 page directory + 3 smaller page tables



Inverted Page Table: Basic Idea

- Page table is a per-process information
  - With M processes in memory, there are M independent page tables
- Observation:
  - Difficult to find out which frames are occupied, and by which processes
  - Only N physical memory frames can be occupied
  - ➔ Out of the M page tables, only N entries are valid
  - ➔ Huge waste: N << Overhead of M page tables

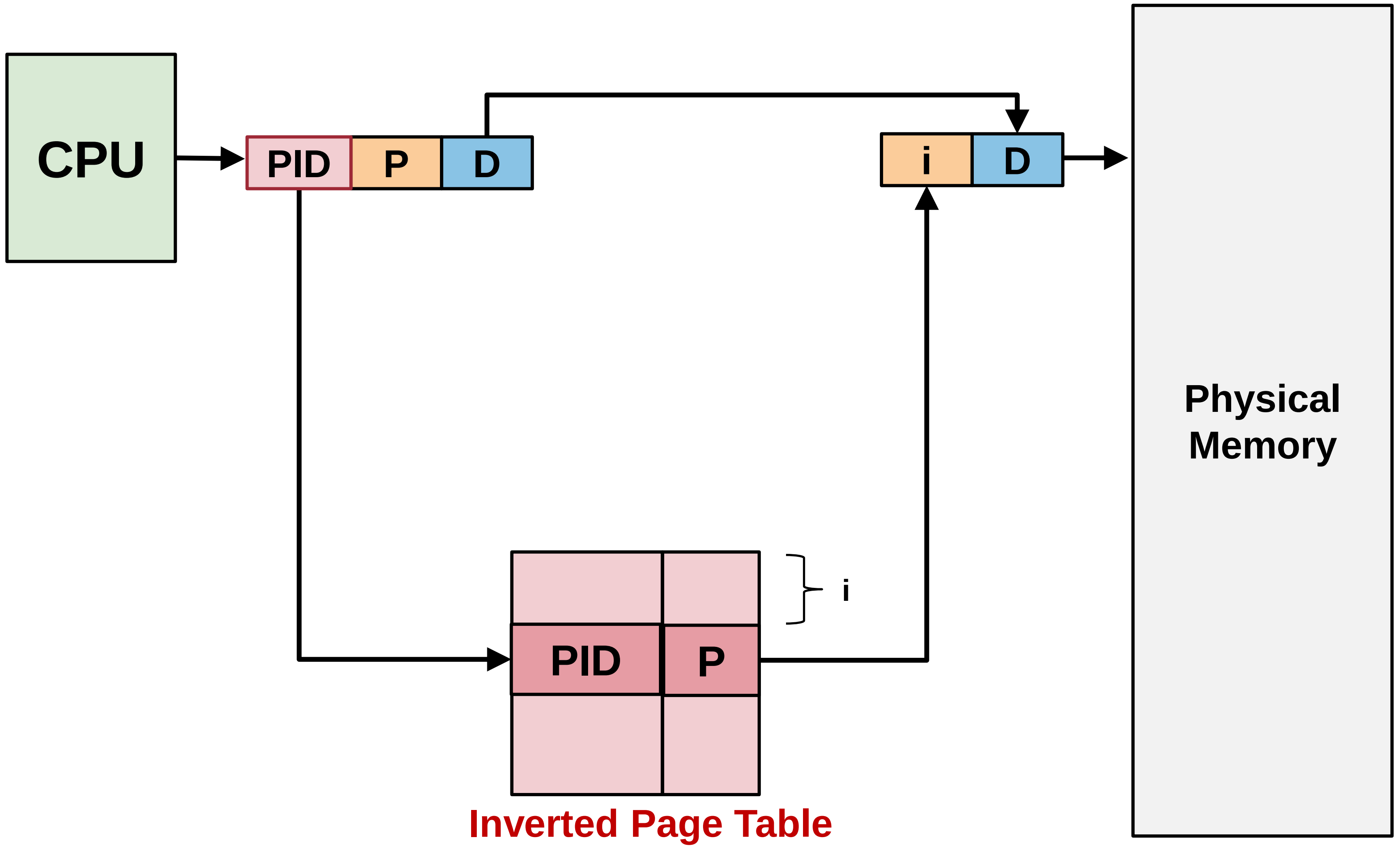
- Idea:
  - Keep a **single** mapping of physical frame to <pid, page#>
    - pid = process id, page# = page number
    - page# is not unique among processes
    - pid + page# can uniquely identify a memory page

Inverted Page Table: Basic Idea (cont)

- In a normal page table, entries are ordered by page number
  - To lookup page X, simply access the Xth entry
- In an inverted page table, entries are ordered by frame number
  - To lookup page X, need to search the whole table

- **Advantage:**
  - Huge saving: One table for all processes
  - Frame management is easier and faster
- **Disadvantage**:
  - Slow translation

Inverted Table: Illustration



Who should I kick out next?

PAGE REPLACEMENT ALGORITHMS

Page Replacement Algorithms

- Suppose there is no free physical memory frame during a page fault:
  - Need to evict (free) a memory page
- When a page is evicted:
  - Clean page: not modified ➔ no need to write back to storage
  - Dirty page: modified ➔ need to write back to storage

- Algorithms to find a suitable replacement page
  - **Optimum (OPT)**
  - **FIFO**
  - **Least Recently Used**
  - **Second-Chance (Clock)**
  - etc.

Modeling Memory References

- In actual memory references:
  - Logical Address = Page Number + Offset

- However, to study page replacement algorithms
  - Only **page number** is important

➔ To simplify discussion, memory references are often modeled as **memory reference strings**, i.e., a sequence of page numbers

Page Replacement Algorithms: Evaluation

Memory access time:

$$T_{access} = (1 - p) * T_{mem} + p * T_{page\_fault}$$

- p = probability of page fault
- T_{mem} = access time for memory resident page
- T_{page_fault} = access time if page fault occurs

- Since $T_{page\_fault} >> T_{mem}$
  - Need to reduce p to keep T_{access} reasonable
- See for yourself, try to find p if:
  - T_{mem} = 100ns, T_{page_fault} = 10ms, T_{access} = 120ns

Good algorithm should **reduce the total number of page faults**

Optimal Page Replacement (OPT)

- General Idea:
  - Replace the page that **will not** be used again for the **longest period of time**
  - **Guarantees** minimum number of page faults
- Unfortunately, not feasible:
  - Need **future knowledge** of memory references
- Still useful:
  - As a base of comparison for other algorithms
  - The closer to OPT == better algorithm

Example: OPT (6 Page Faults)

Time	Memory Reference	Frame			Next Use Time			Fault?
		A	B	C				
1	**2**	**2**			3	⍰	⍰	**Y**
2	**3**	2	**3**		3	9	⍰	**Y**
3	**2**	**2**	3		6	9	⍰	
4	**1**	2	3	**1**	6	9	⍰	**Y**
5	**5**	2	3	**5**	6	9	8	**Y**
6	**2**	**2**	3	5	10	9	8	
7	**4**	**4**	3	5	⍰	9	8	**Y**
8	**5**	4	3	**5**	⍰	9	11	
9	**3**	4	**3**	5	⍰	⍰	11	
10	**2**	**2**	3	5	12	⍰	11	**Y**
11	**5**	2	3	**5**	⍰	⍰	11	
12	**2**	**2**	3	5	⍰	⍰	⍰	

FIFO Page Replacement Algorithm

- General Idea:
  - Memory pages are evicted based on their loading time
  - ➔ Evict the oldest memory page

- Implementation:
  - OS maintains a queue of resident page numbers
    - Remove the first page in queue if replacement is needed
    - Update the queue during page fault trap
  - Simple to implement
    - No hardware support needed

Example: FIFO (9 Page Faults)

Time	Memory Reference	Frame			Loaded at Time			Fault?
		A	B	C				
1	**2**	**2**			1			**Y**
2	**3**	2	**3**		1	2		**Y**
3	**2**	**<u>2</u>**	3		1	2		
4	**1**	2	3	**1**	1	2	4	**Y**
5	**5**	**5**	3	1	5	2	4	**Y**
6	**2**	5	**2**	1	5	6	4	**Y**
7	**4**	5	2	**4**	5	6	7	**Y**
8	**5**	**<u>5</u>**	2	4	5	6	7	
9	**3**	**3**	2	4	9	6	7	**Y**
10	**2**	3	**<u>2</u>**	4	9	6	7	
11	**5**	3	**5**	4	9	11	7	**Y**
12	**2**	3	5	**2**	9	11	12	**Y**

FIFO: Problems

- If number of physical frames increases (e.g., more RAM)
  - Number of page faults should decrease
- FIFO violates this simple intuition!
  - Use 3 / 4 frames to try: **1 2 3 4 1 2 5 1 2 3 4 5**
- Opposite behavior (↑ frames ➔ ↑ page faults)
  - Known as **Belady's Anomaly**
- Reason:
  - FIFO does not exploit **temporal locality**

Least Recently Used Page Replacement (**LRU**)

- General Idea:
  - Make use of temporal locality:
    - Replace the page that has not been accessed in the longest time
  - Expect a page to be reused in a short time window
    - Have not accessed for some time ➔ most likely will not be accessed again
- Notes:
  - Attempts to approximate the OPT algorithm
    - Gives good results generally
  - Does not suffer from Belady's Anomaly

Example: LRU (7 Page Faults)

Time	Memory Reference	Frame			Last Use Time			Fault?
		A	B	C				
1	**2**	**2**			1			**Y**
2	**3**	2	**3**		1	2		**Y**
3	**2**	**2**	3		3	2		
4	**1**	2	3	**1**	3	2	4	**Y**
5	**5**	2	**5**	1	3	5	4	**Y**
6	**2**	**2**	5	1	6	5	4	
7	**4**	2	5	**4**	6	5	7	**Y**
8	**5**	2	**5**	4	6	8	7	
9	**3**	**3**	5	4	9	8	7	**Y**
10	**2**	3	5	**2**	9	8	10	**Y**
11	**5**	3	**5**	2	9	11	10	
12	**2**	3	5	**2**	9	11	12	

LRU: Implementation Details

- Implementing LRU is not easy:
  - Need to keep track of the "last access time" somehow
  - Need substantial hardware support

1. Approach A - **Use a Counter**:
   - A logical "time" counter, which is incremented for every memory reference
   - Page table entry has a "time-of-use" field
     - Store the time counter value whenever reference occurs
     - Replace the page with smallest "time-of-use"
   - Problems:
     - Need to search through all pages
     - "Time-of-use" is forever increasing (overflow possible!)

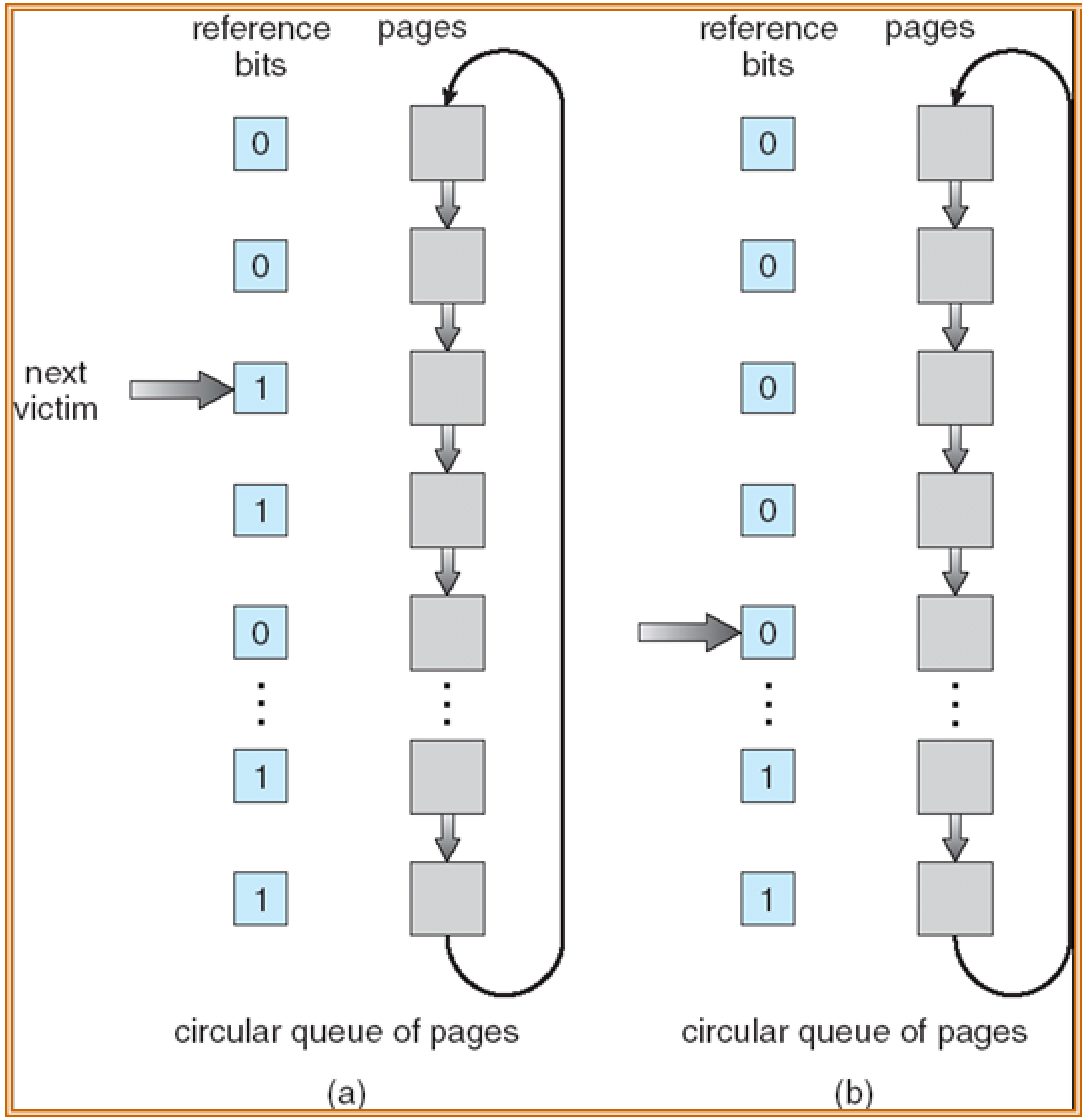
LRU: Implementation Details (cont)

2. Approach B - Use a "Stack":
   - Maintain a stack of page numbers
   - If page **X** is referenced
     - Remove from the stack (if entry exists)
     - Push on top of stack
   - Replace the page at the bottom of stack
     - No need to search through all entries
   - Problems:
     - Not a pure stack: Entries can be removed from anywhere in the stack
     - Hard to implement in hardware

Second-Chance Page Replacement (CLOCK)

- General Idea:
  - Modified FIFO to give a second chance to pages that are accessed
  - Each PTE now maintains a "reference bit":
    - 1 = Accessed, 0 = Not accessed
  - Algorithm:
    1. The oldest FIFO page is selected
    2. If reference bit == 0 ➔ Page is replaced
    3. If reference bit == 1 ➔ Page is given a 2nd chance
       - Reference bit cleared to 0
       - Load time reset ➔ page taken as newly loaded
       - Next FIFO page is selected, go to Step 2
  - Degenerate into FIFO algorithm
    - When all pages have reference bit == 1 (or all == 0)

Second-Chance: Implementation Details



- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance to a page with '0' reference bit
  - Clear the reference bit as pointer passes through

Example: CLOCK(6 Page Faults)

Time	Memory Reference	Frame (with Ref Bit)			Fault?
		A	B	C	
1	2	▶2 (0)			Y
2	3	▶2 (0)	3 (0)		Y
3	2	▶<u>2</u> (1)	3 (0)		
4	1	▶2 (1)	3 (0)	1 (0)	Y
5	5	2 (0)	5 (0)	▶1 (0)	Y
6	2	<u>2</u> (1)	5 (0)	▶1 (0)	
7	4	▶2 (1)	5 (0)	4 (0)	Y
8	5	▶2 (1)	<u>5</u> (1)	4 (0)	
9	3	▶2 (0)	5 (0)	3 (0)	Y
10	2	▶<u>2</u> (1)	5 (0)	3 (0)	
11	5	▶2 (1)	<u>5</u> (1)	3 (0)	
12	2	▶<u>2</u> (1)	5 (1)	3 (0)	

▶ Victim Page

Which process should I favor?

FRAME ALLOCATION

Frame Allocation

- Consider:
  - There are **`N`** physical memory frames
  - There are **`M`** processes competing for frames
  - What is the best way to distribute the **`N`** frames among **`M`** processes?

- Simple Approaches:
  - **Equal Allocation:**
    - Each process gets **`N / M`** frames
  - **Proportional Allocation:**
    - Processes are different in size (memory usage)
    - Let $\mathbf{size}_p$ = size of process **`p`**, $\mathbf{size}_{total}$ = total size of all processes
    - Each process gets $\mathbf{size}_p/\mathbf{size}_{total}*\mathbf{N}$ frames

Frame Allocation and Page Replacement

- The implicit assumption for page replacement algorithms discussed:
  - Victim pages are selected **among pages of the process** that causes page fault
  - Known as **local replacement**

- If victim page can be chosen **among all physical frames**:
  - ➔ Process P can take a frame from Process Q by evicting Q's frame during replacement!
  - Known as **global replacement**

Local vs Global Replacement

- **Local Replacement:**
  - **Pros:**
    - Frames allocated to a process remain constant ➔ Performance is stable between multiple runs
  - **Cons:**
    - If frames allocated to a process are not enough ➔ hinder the performance of the process

- **Global Replacement:**
  - **Pros:**
    - Allow dynamic self-adjustment between processes
      - Process that needs more frames can get from other
  - **Cons:**
    - Badly behaved process can negatively affect others
    - Frames allocated to a process can be different from run to run

Frame Allocation and Thrashing

- Insufficient physical frames ➔ Thrashing in process
  - Heavy I/O to bring non-resident pages into RAM

- Hard to find the right number of frames:
  - If global replacement is used:
    - A thrashing process "steals" page from other processes
    - ➔ causes other processes to thrashing (**Cascading Thrashing**)
  - If local replacement is used:
    - Thrashing can be limited to one process
    - But that single process can hog the I/O and degrades the performance of other processes

Finding the right number of frames…

- Observation:
  - The **set** of pages referenced by a process is relatively constant in a period of time
    - Known as **locality**
  - However, as time passes, the set of pages can change

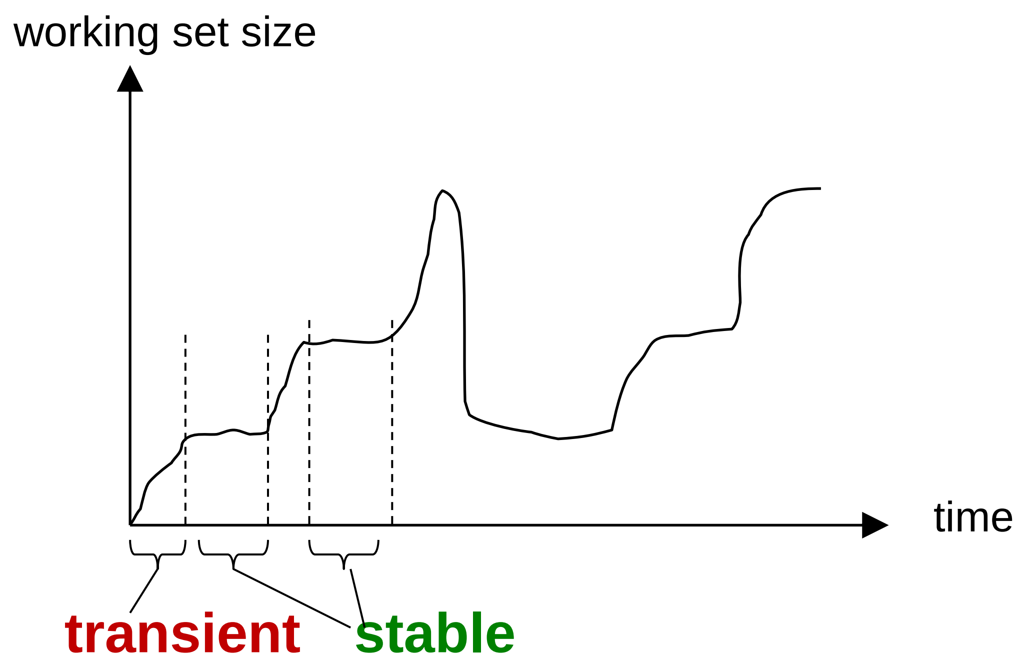
- Example:
  - When a function is executing, memory references are likely on:
    - *local variables, parameters, code in that function*
    - these pages define the locality for the function
  - After the function terminates, references will change to another set of pages

Working Set Model

- Using the observation on locality:
  - In a new “locality”:
    - A process will cause page fault for the set of pages
  - With the set of pages loaded in frames:
    - No/few page faults until process transits to new locality

- Working Set Model:
  - Defines Working Set Window Δ
    - An interval of time
  - **`W(t,Δ)`** = active pages in the interval at time **`t`**
  - Allocate enough frames for pages in **`W(t, Δ)`** to reduce the number of page faults

Working Set Model: Illustration



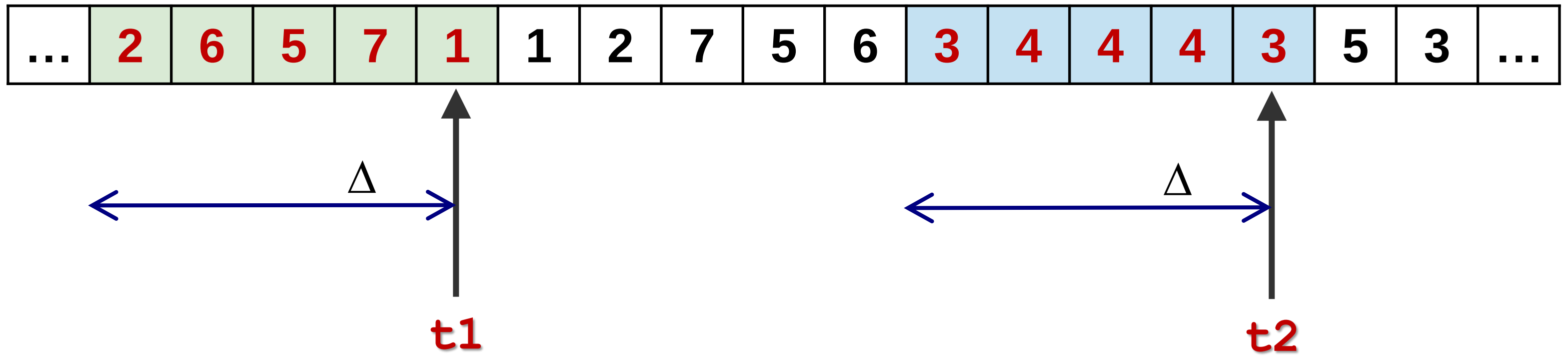
Transient region:
working set changing in size

Stable region:
working set about the same for a long time

- Accuracy of working set model is directly affected by the choice of Δ
  - Too small: May miss pages in the current locality
  - Too big: May contains pages from different locality

Working Set Model: Illustration

- Example memory reference strings



- Assume
  - Δ = an interval of 5 memory references
- **W(*t1*,Δ)={1,2,5,6,7}** (5 frames needed)
- **W(*t2*,Δ)={3,4}** (2 frames needed)
- Try using different Δ values

Summary

- Virtual memory
  - The "why" and "how"

- Discussed different aspects of virtual memory management
  - Use different page table structure to reduce page table overhead
  - Use different page replacement algorithms to reduce page fault
  - How frame allocation affects page fault of a process