Operating Systems (A) (Honor Track)

Lecture 10: Page Replacement

Yao Guo (郭耀)

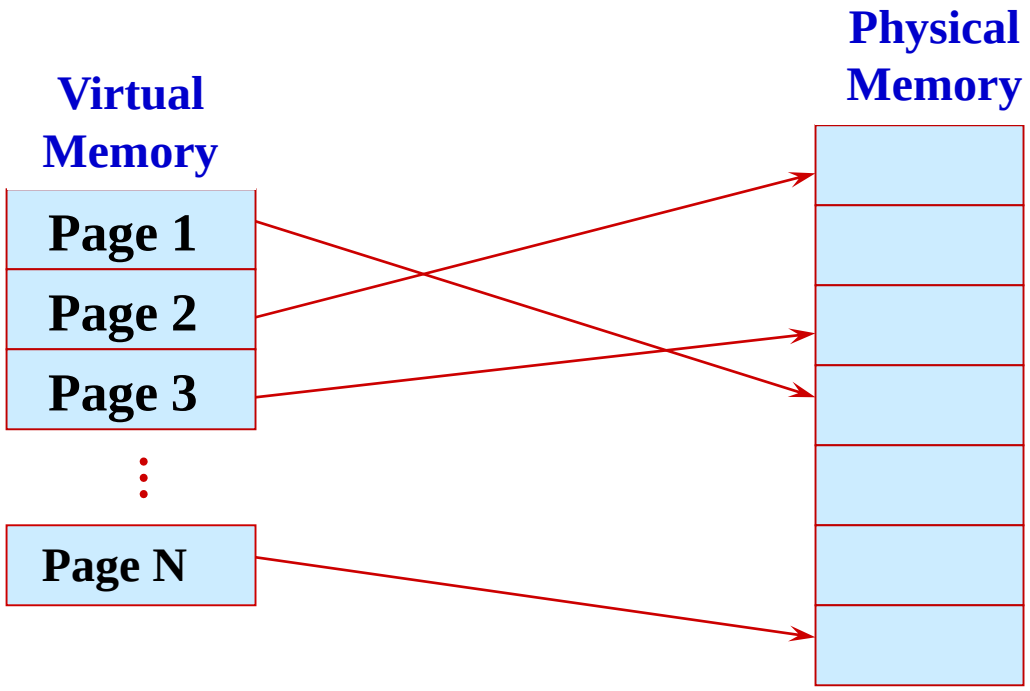
Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

Review: Paging

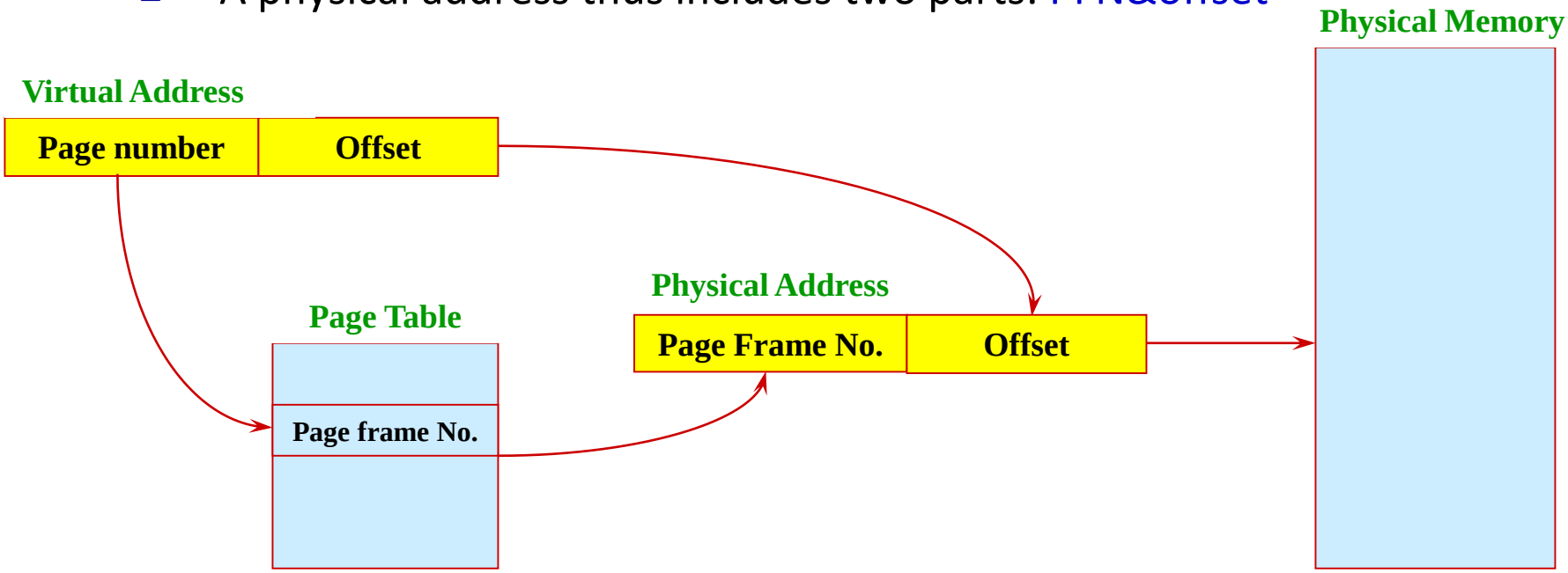
- (Virtual) Pages: fixed-size units in the virtual address space
- Page frames: the corresponding units in the physical memory
- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory



Paging Basics

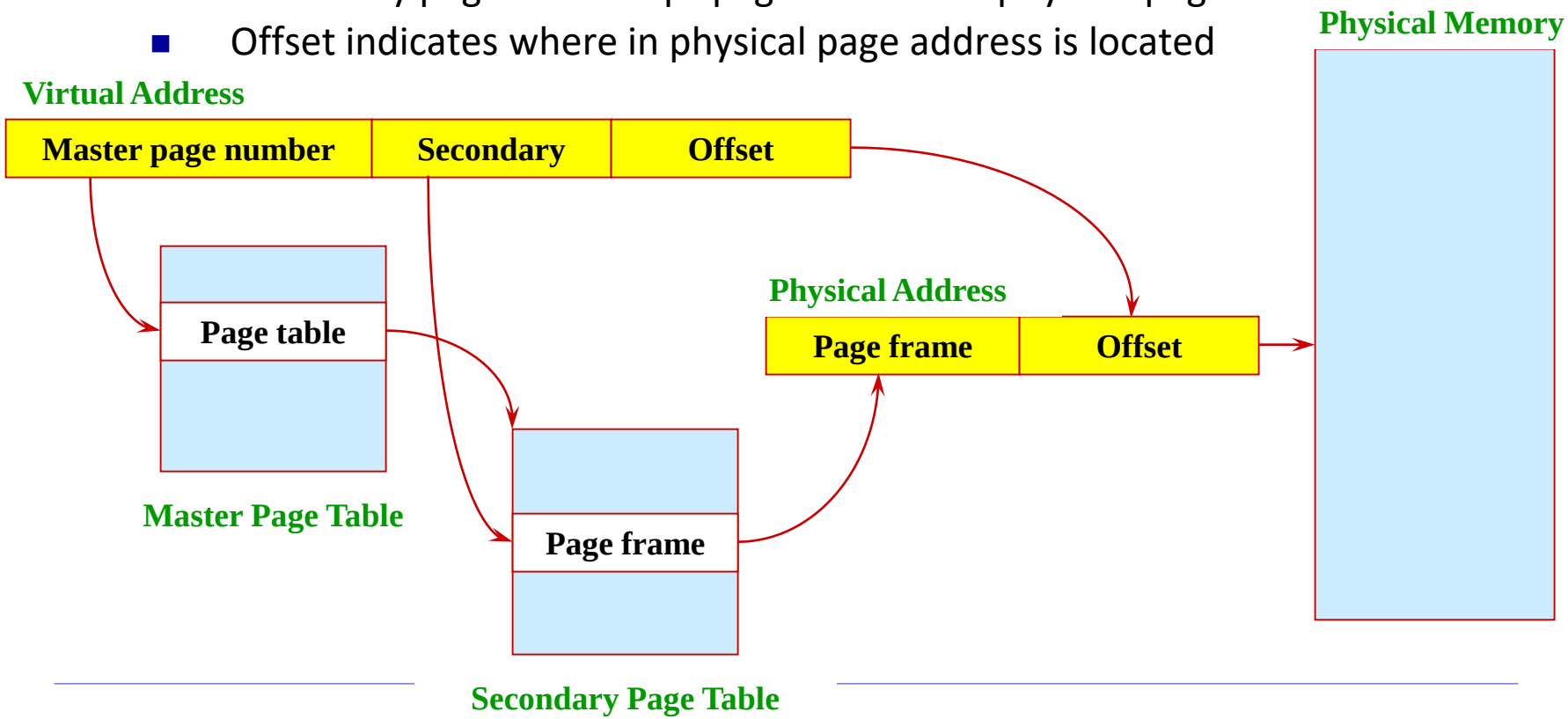
- Translating addresses
  - Virtual address has two parts: virtual page number (VPN) and offset

  - Page tables map VPN to page frame number (PFN)
  - A physical address thus includes two parts: PFN&offset



Two-Level Page Tables

- Two-level page tables
  - Virtual addresses (VAs) have three parts:
    - Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located



This Lecture

Page Replacement

Demand paged virtual memory

Page replacement algorithms

Design issues

Implementation Issues

Buzz Words

- Page replacement
- Demand paging
- Temporal/spatial locality
- Least recently used (LRU)
- Working set
- Copy on write
- Shared memory
- Mapped files

This Lecture

Page Replacement

Demand paged virtual memory

Page replacement algorithms

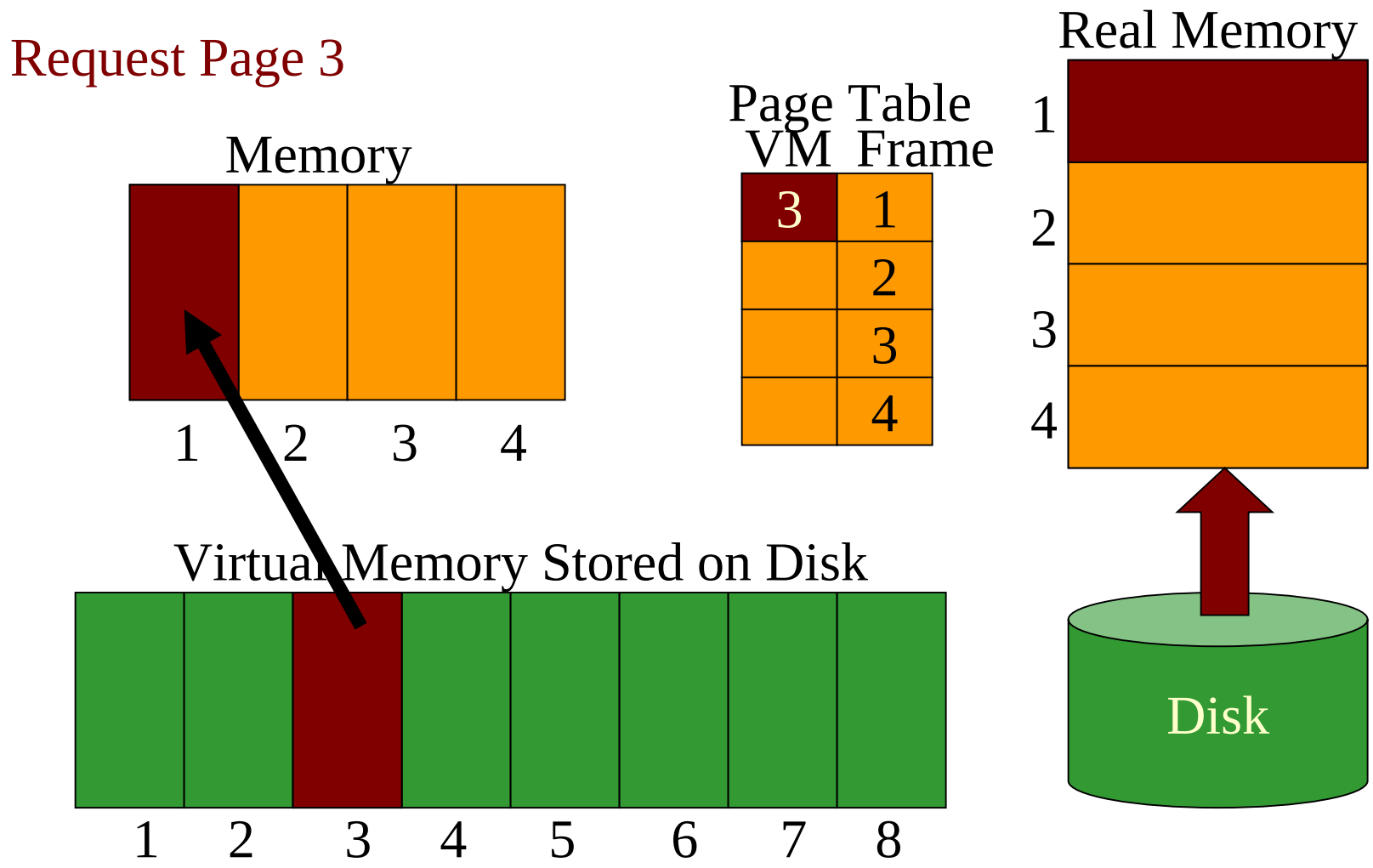
Design issues

Implementation Issues

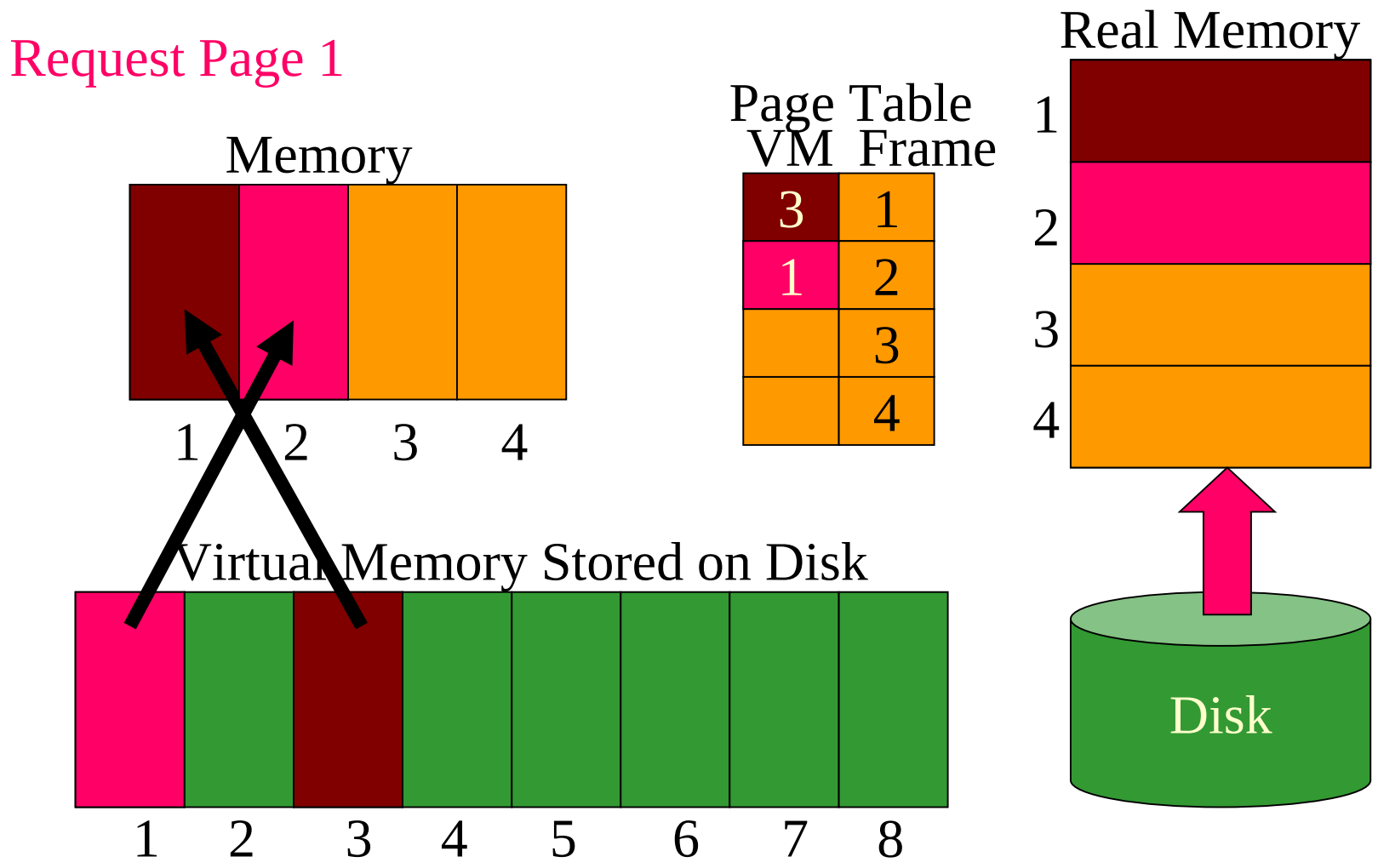
Paged Virtual Memory

- What if not all virtual memory can fit into physical memory
  - The physical memory is small, or too many running processes
- Pages can be moved between memory and disk (swapping)
  - This process is called demand paging
- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
  - Evicted pages go to disk (the swap file/backing store)
  - The movement of pages between memory and disk is done by the OS, and is transparent to the application

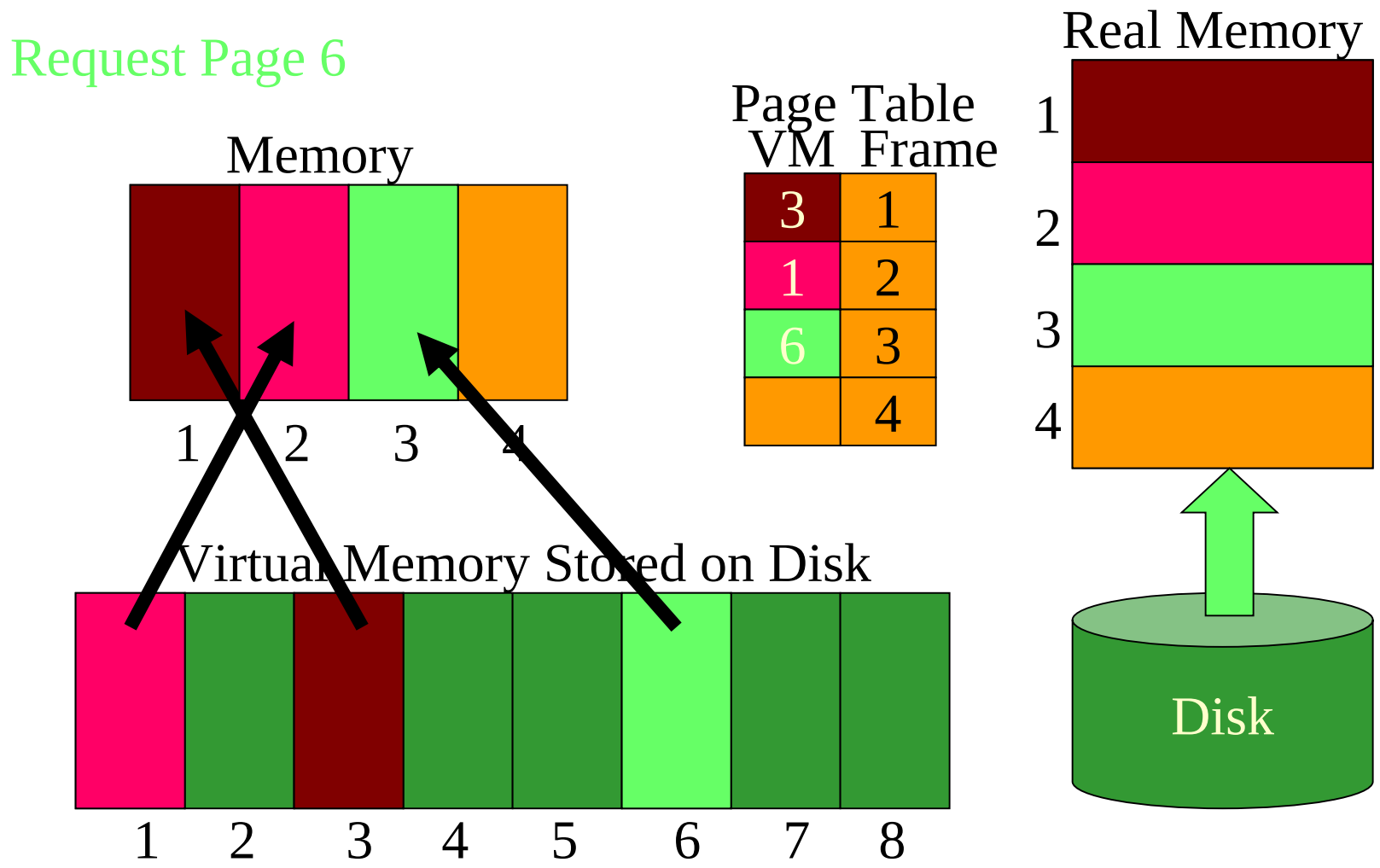
Demand Paging



Demand Paging

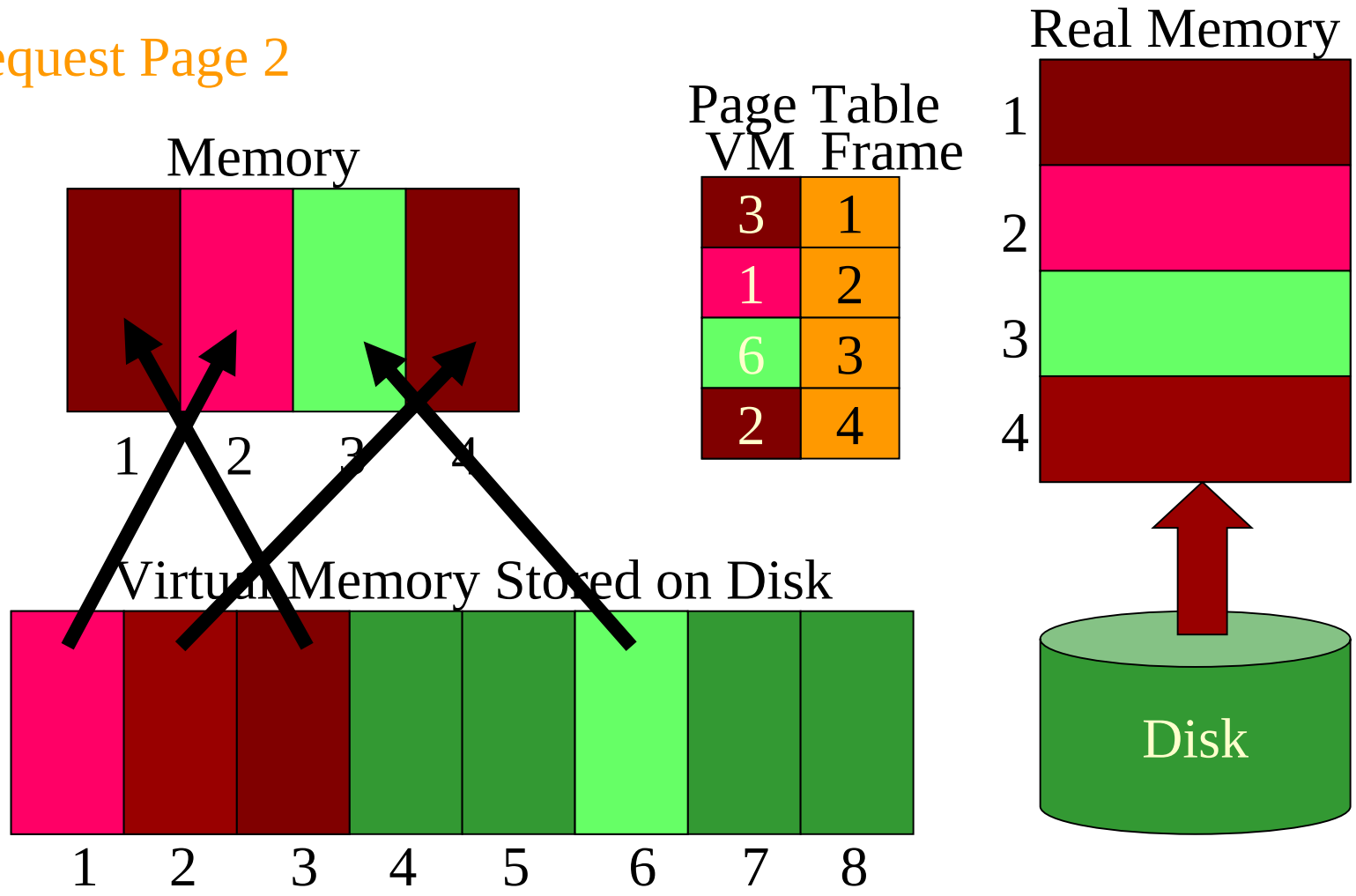


Demand Paging



Demand Paging

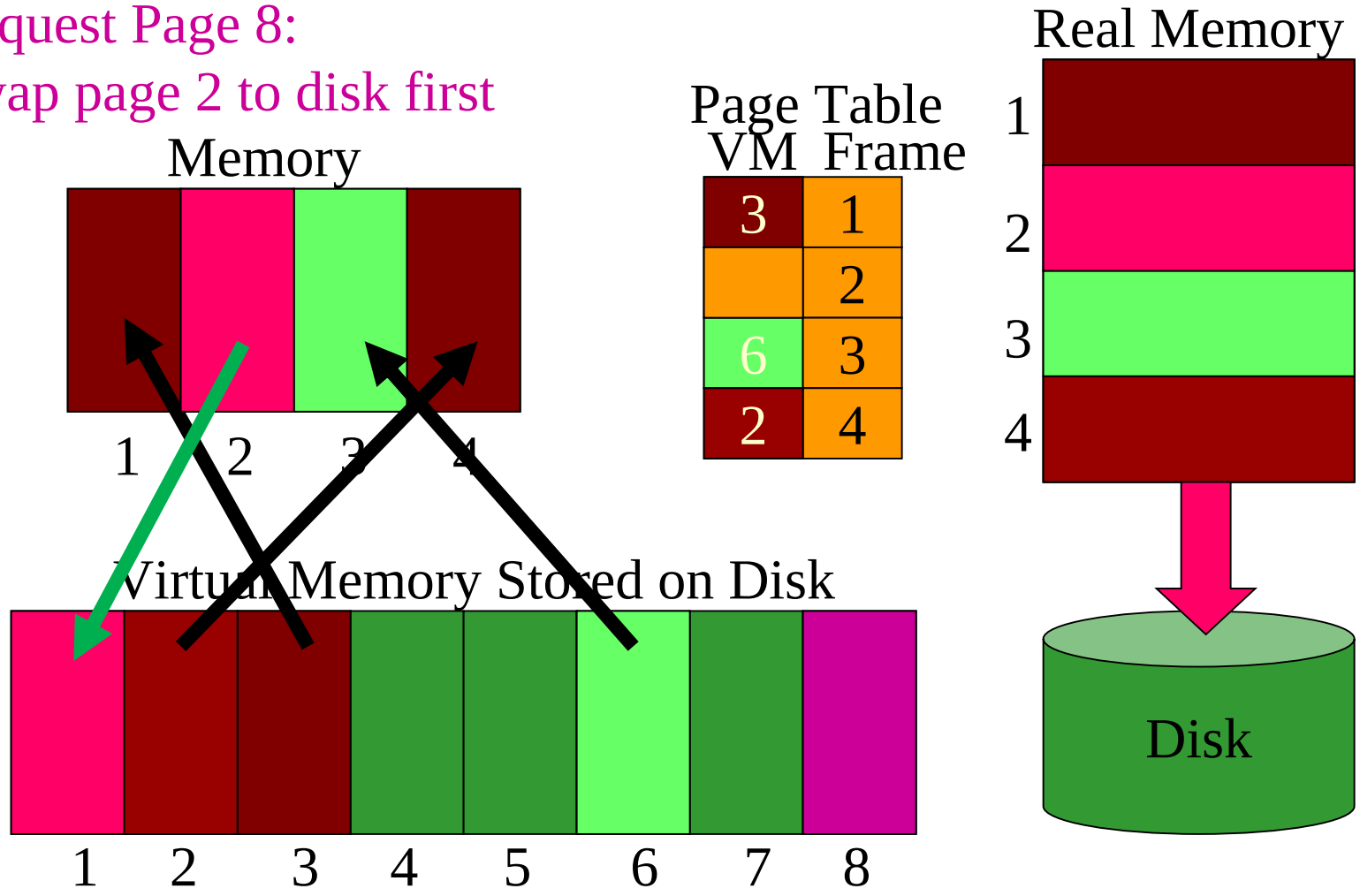
Request Page 2



Demand Paging

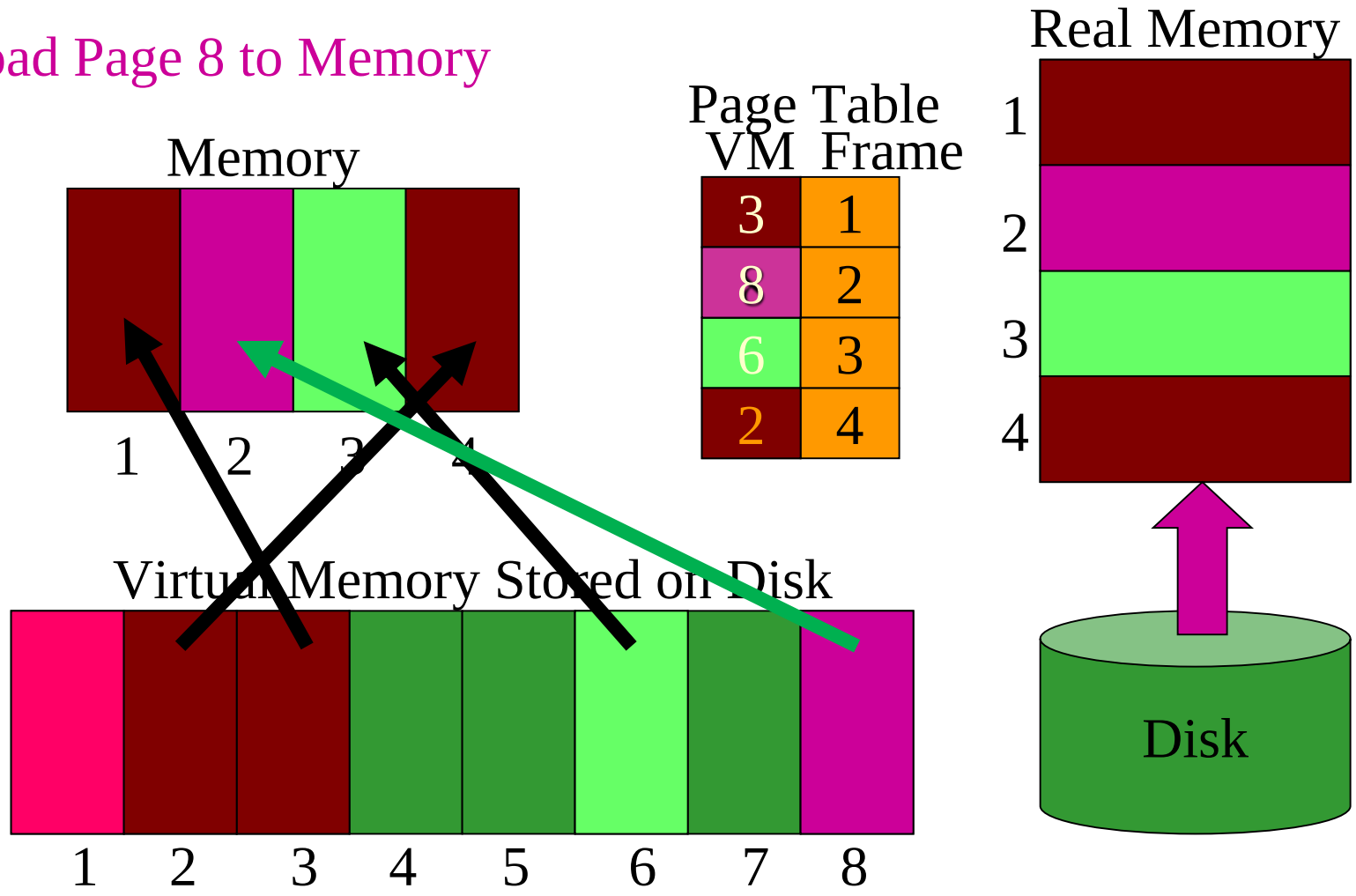
Request Page 8:

Swap page 2 to disk first



Demand Paging

Load Page 8 to Memory



Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
  2. When a process accesses the page, the invalid PTE will cause a trap (page fault)
  3. The trap will run the page fault handler (in OS)
  4. Handler uses the invalid PTE to locate page in the swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process
- But where do we put the page? Have to evict another page
  - OS usually keeps a pool of free pages around so that allocations do not always cause evictions

Put Things Together

- We started the topic of memory management with the high-level problem of translating virtual addresses into physical addresses
- We've covered all of the pieces
  - Virtual and physical addresses
  - Virtual pages and physical page frames
  - Page tables and page table entries (PTEs), protection
  - TLBs
  - Demand paging
- Now let's put it together

The Common Case

- Situation: a process is executing on the CPU, and it issues a read to an address
  - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address
  3. TLB validates that the PTE protection allows reads (in this example)
  4. PTE specifies which physical frame holds the page
  5. MMU combines the physical frame and offset into a physical address
  6. MMU then reads from that physical address, returns value to CPU
- Note: This is all done by the hardware

TLB Misses

- At this point, two other things can happen
  1. TLB does not have a PTE mapping this virtual address
  2. PTE exists, but memory access violates PTE protection bits
- We'll consider each in turn

Reloading the TLB

- If TLB does not have the mapping, two possibilities:
  1. MMU loads PTE from the page table in memory
     - Hardware managed TLB, OS not involved in this step
     - OS has already set up the page tables so that the hardware can access it directly
  2. Trap to the OS
     - Software managed TLB, OS intervenes at this point
     - OS does lookup in page table, loads PTE into TLB
     - OS returns from exception, TLB continues
- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB

TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out

- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
    - Becomes a page fault…

Page Faults

- PTE can indicate a protection/page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to process, or use other tricks (e.g., copy on write, mapped files)
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame, reads from disk, maps PTE to physical frame

The Demand Paging Algorithm

- Never bring a page into primary memory until it's needed
  1. Page fault
  2. Check if a valid virtual memory address. Kill job if not
  3. If valid reference, check if it's cached in memory already (perhaps for some other process.) If so, skip to 7
  4. Find a free page frame. If no free page available, choose one to evict

     4.a If the victim page is dirty, write it out to disk first
  5. Suspend user process, map address into disk block and fetch disk block into page frame
  6. When disk read finished, add VM mapping for page frame
  7. If necessary, restart process

Demand Paging

- Summary
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
  - References to evicted pages cause a TLB miss
    - PTE was invalid, causes fault
  - OS allocates a page frame, reads page from disk
  - When I/O completes, the OS fills in PTE, marks it valid, and restarts faulting process
- Dirty vs. clean pages
  - Only dirty pages (modified) need to be written to disk
  - Clean pages do not – but you need to know where on disk to read them from again

This Lecture

Page Replacement

Demand paged virtual memory

Page replacement algorithms

Design issues

Implementation Issues

Page Replacement

1. First find the location of the selected page on the disk
2. Find a free page frame
    1. If there is free page frame, use it
    2. Otherwise, select a page frame using the page replacement algorithm
    3. Write the selected page to the disk and update any necessary tables
3. Read the requested page from the disk
4. Restart the user process
5. Be careful of synchronization problems
    - For example, page faults may occur for pages being paged out
    - In this case, the page may be in the air – cannot find it in the disk either

Issue: Eviction

- Dirty pages require writing, clean pages don't
  - Hardware has a dirty bit for each page frame indicating this page has been updated or not
  - Where do you write?
    - To "swap space"
- Hopefully, kick out a less-useful page
- Goal: kick out the page that's the least useful
- Problem: how do you determine usefulness?
  - Kick out pages that aren't likely to be used again
  - Heuristic: temporal locality exists

Locality

- Most paging schemes depend on locality
  - Processes reference pages in localized patterns
- Temporal locality
  - Locations referenced recently likely to be referenced again
- Spatial locality
  - Locations near recently referenced locations are likely to be referenced soon – pre-paging
- Although the cost of paging is high, if it is infrequent enough such that it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging practical

Page Replacement Strategies

- The principle of optimality
  - Replace the page that will not be used again for the longest time in the future
- Random page replacement
  - Choose a page randomly
- NRU - Not Recently Used
- FIFO - First in First Out
- SCR - Second Chance Replacement
- Clock - The Clock page replacement
- LRU - Least Recently Used
- LFU - Least Frequently Used
- Working Set and WSClock

Belady's Algorithm

- Known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream/sequence
  - Idea: replace the page that will not be used for the longest time in the future
- Problem: have to predict the future
- Why is Belady's useful then? Use it as a benchmark
  - Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could be improved
    - Random replacement is often the lower bound

Optimal Example

- Five kinds of references: A, B, C, D, E
  - 12 references, 7 faults: hit ratio? miss ratio?

Reference sequence				
	Fault?			
A	Yes	A		
B	Yes	A	B	
C	Yes	A	B	C
D	Yes	A	B	**D**
A	No	A	B	D
B	No	A	B	D
E	Yes	A	B	E
A	No	A	B	E
B	No	A	B	E
C	Yes	C	B	E
D	Yes	C	D	E
E	No	C	D	E

And what do we expect when we have more memory?

Intuitive Paging Behavior with Increasing Number of Page Frames

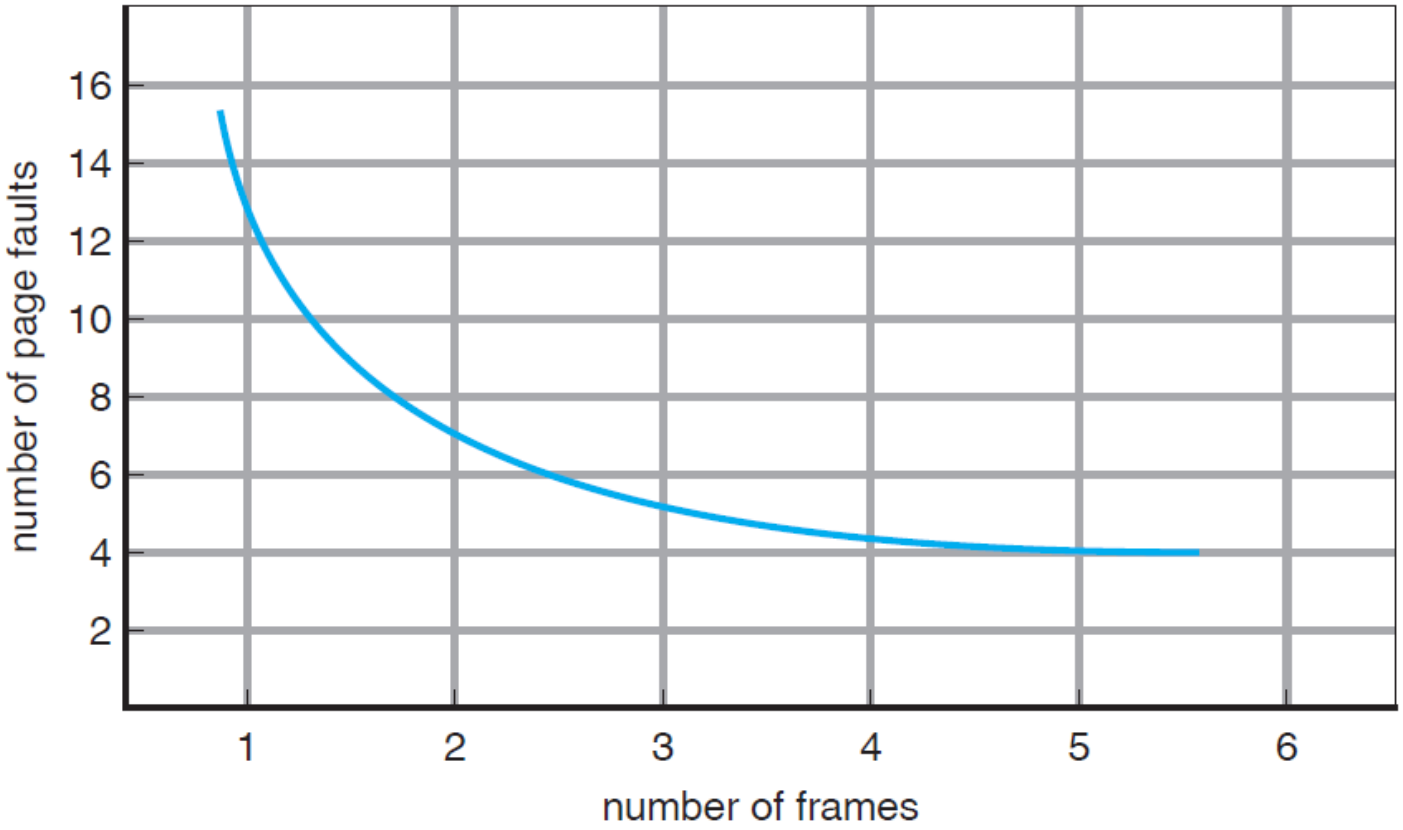


Figure 8.11 Graph of page faults versus number of frames.

Fig. from Operating System Concepts Essentials, Silberschatz et al., 2011

NRU: Not Recently Used

- NRU: Evict a page that is NOT recently used
- OS: collect useful page usage statistics
  - Use two bits:
  - R – the page is referenced (read/write)
  - M – the page is recently modified
- Implementation
  - Hardware bits
  - OS simulation: can be simulated using the operating system's page fault and clock interrupt mechanisms
  - R bit will be cleared periodically based on clock

NRU: Not Recently Used (2)

- Pages can be classified into 4 classes:
  - Class 0: not referenced, not modified.
  - Class 1: not referenced, modified.
  - Class 2: referenced, not modified.
  - Class 3: referenced, modified.

- Why is Class 1 possible?

- Which one should we replace during a page fault?

First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in the longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago will not be used
- Why might this be bad?
  - Then again, maybe it's not
  - We don't have any info to say one way or the other
- FIFO suffers from "Belady's Anomaly"
  - The fault rate might actually increase when the algorithm is given more memory (very bad)

FIFO Example

- Five kinds of references: A, B, C, D, E
  - 12 references, 9 faults

Reference sequence				
	Fault?			
A	Yes	A		
B	Yes	A	B	
C	Yes	A	B	C
D	Yes	**D**	B	C
A	Yes	D	A	C
B	Yes	D	A	B
E	Yes	E	A	B
A	No	E	A	B
B	No	E	A	B
C	Yes	E	C	B
D	Yes	E	C	D
E	No	E	C	D

Belady's Anomaly (for FIFO)

- Five kinds of references: A, B, C, D, E
  - 12 references, 10 faults (we have 9 faults with 3 slots!)

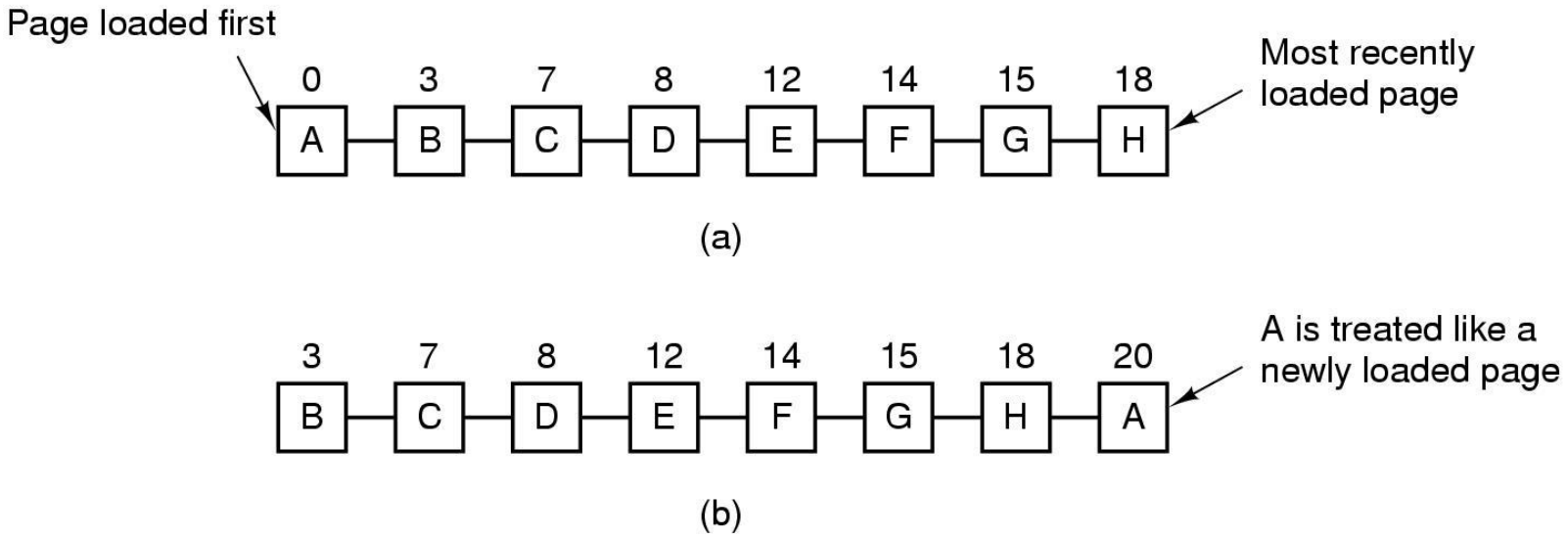
Reference sequence					
	Fault?				
A	Yes	A			
B	Yes	A	B		
C	Yes	A	B	C	
D	Yes	A	B	C	D
A	No	A	B	C	D
B	No	A	B	C	D
E	Yes	E	B	C	D
A	Yes	E	A	C	D
B	Yes	E	A	B	D
C	Yes	E	A	B	C
D	Yes	D	A	B	C
E	Yes	D	E	B	C

Belady's Anomaly

- It was believed that an increase in the number of page frames would always result in the same number of or fewer page faults.
- Until Bélády's anomaly was demonstrated in 1969
  - L. A. Belady, R. A. Nelson, and G. S. Shedler. 1969. An anomaly in space-time characteristics of certain programs running in a paging machine. Commun. ACM 12, 6 (June 1969), 349-353.
- In 2010, a paper showed that the anomaly is in fact unbounded
  - Can construct a reference string to any arbitrary page fault ratio
  - FIFO anomaly is unbounded, arXiv:1003.1336

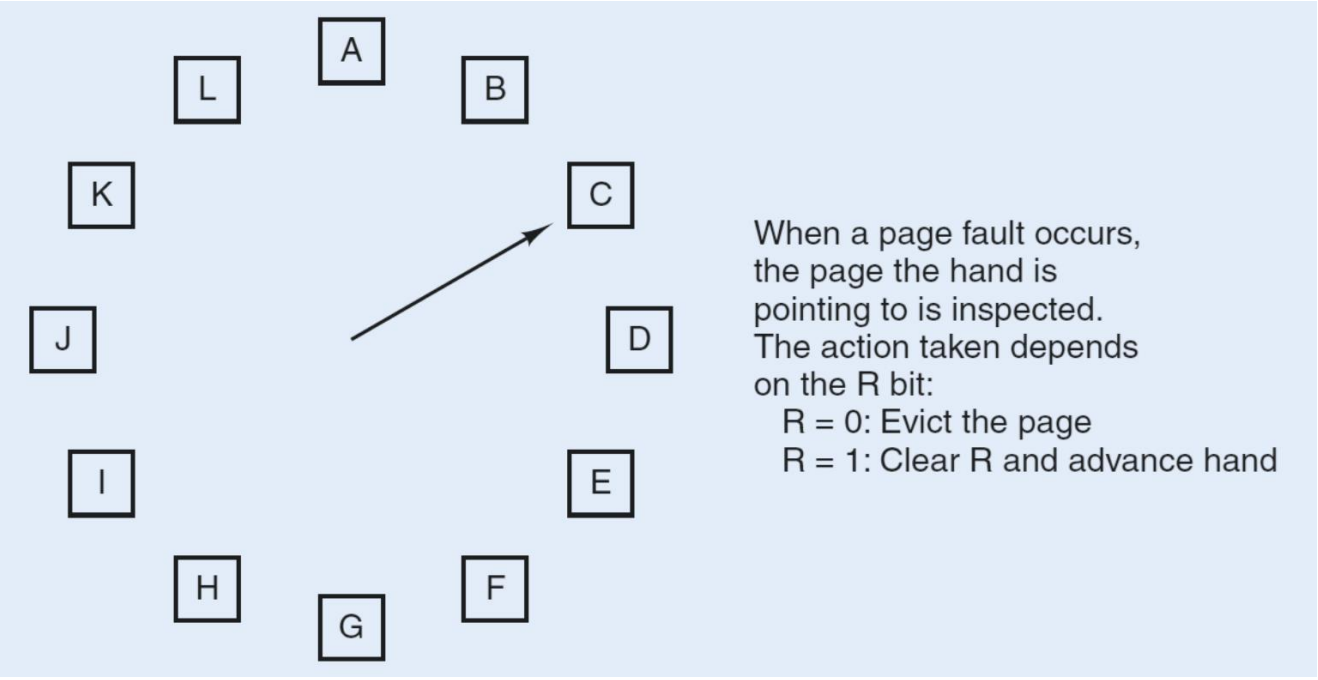
Second-Chance (SCR)

- SCR: A simple modification to FIFO that avoids the problem of throwing out a heavily used page
  - Use an R bit for each page
  - Check R bit of the oldest page
  - Don't evict it if it has been used recently (R = 1)

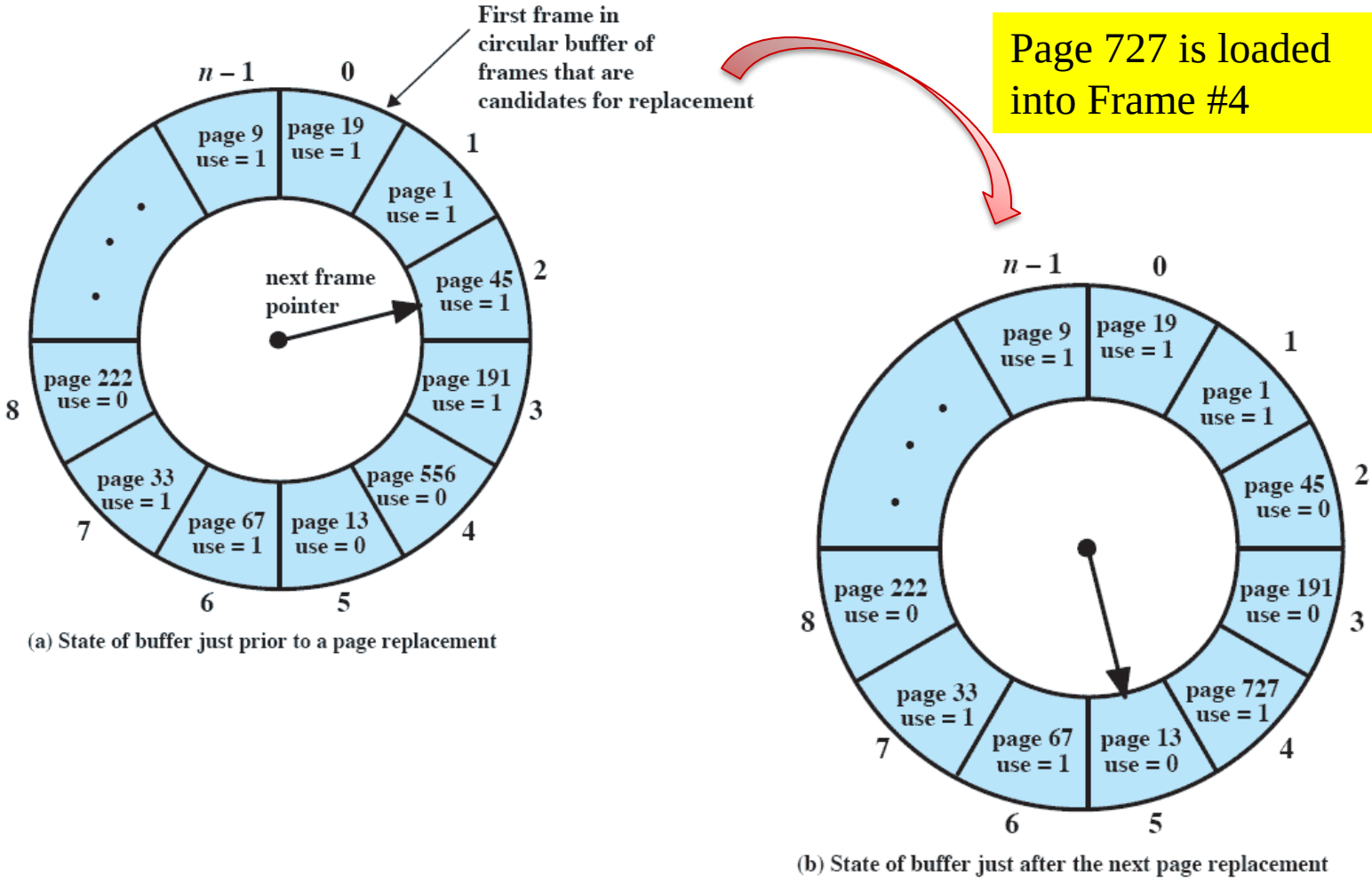


Clock Page Replacement

- Problem with SCR: it is unnecessarily inefficient because it is constantly moving pages around on its list.
- Clock: keep all the page frames on a circular list in the form of a clock, the hand points to the oldest page



Clock Algorithm: example



Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - Idea: we can't predict the future, but we can make a guess based upon past experience
  - On replacement, evict the page that has not been used for the longest time in the past
    - (comparing to Belady's: future)
  - When does LRU do well? When does LRU do poorly?

LRU Example

- Five kinds of references: A, B, C, D, E
  - 12 references, 10 faults

Reference sequence				
	Fault?			
A	Yes	A		
B	Yes	A	B	
C	Yes	A	B	C
D	Yes	**D**	B	C
A	Yes	D	A	C
B	Yes	D	A	B
E	Yes	E	A	B
A	No	E	A	B
B	No	E	A	B
C	Yes	C	A	B
D	Yes	C	D	B
E	Yes	C	D	E

LRU Implementation

- Counter implementation
  - Every page entry has a counter.
  - Every time a page is referenced, increment a global counter and store it in the page counter.
  - For page replacement, select the page with the lowest counter (search for it).
- Stack implementation
  - Maintain a stack of page number in a double link list.
  - Move a referenced page to the top (locate and change pointers).
  - For page replacement, take the bottom page.
- Both approaches require large processing overhead, more space, and hardware support

LRU and Anomalies

- Five kinds of references: A, B, C, D, E
  - 12 references, 8 faults (v.s. 10) – Anomalies do not occur

Reference sequence					
	Fault?				
A	Yes	A			
B	Yes	A	B		
C	Yes	A	B	C	
D	Yes	A	B	C	D
A	No	A	B	C	D
B	No	A	B	C	D
E	Yes	A	B	E	D
A	No	A	B	E	D
B	No	A	B	E	D
C	Yes	A	B	E	C
D	Yes	A	B	D	C
E	Yes	E	B	D	C

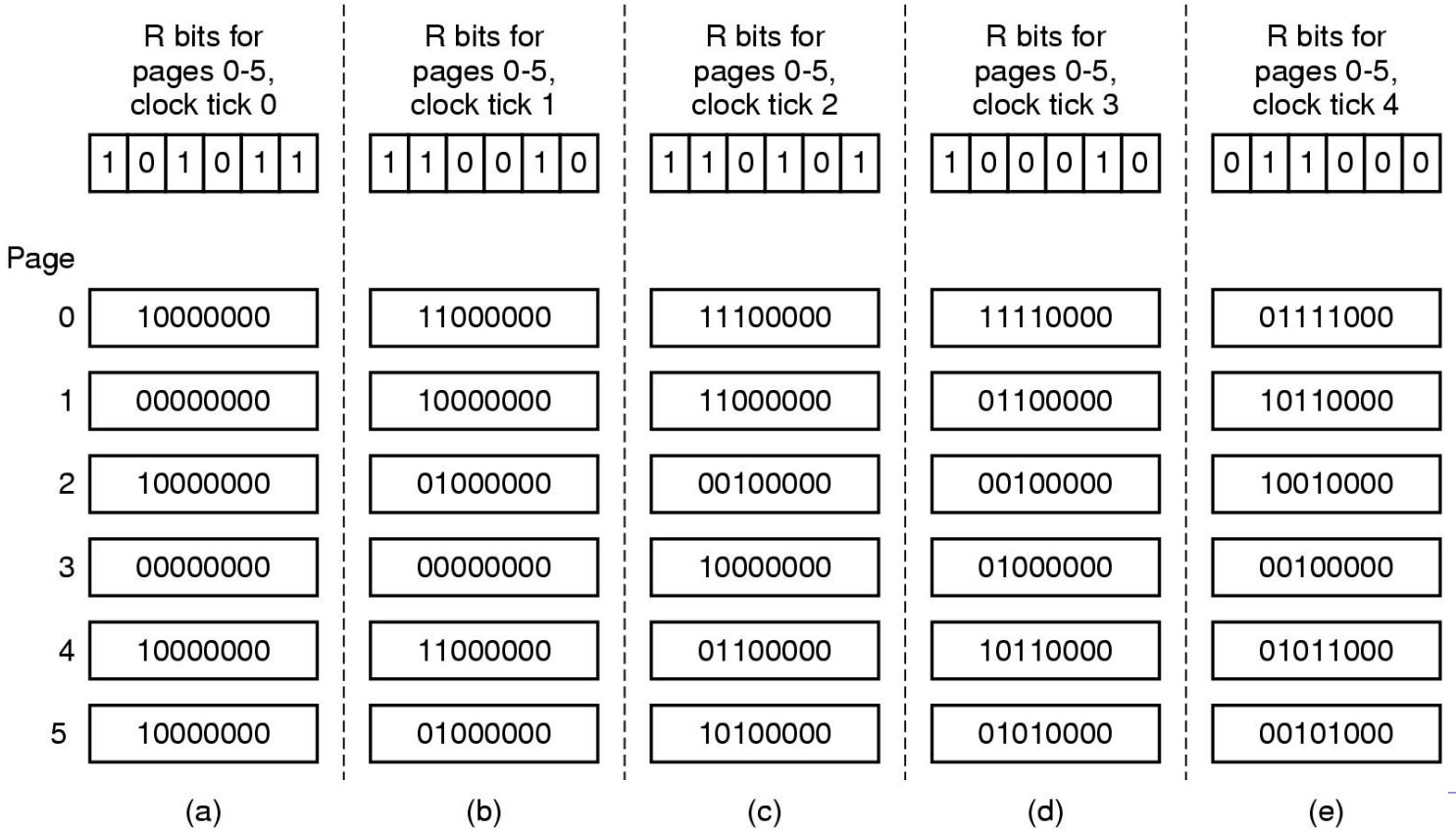
Simulating LRU in Software (1)

- NFU (Not Frequently Used)
  - Use a counter to (roughly) keep track of how often each page has been referenced
  - For each page, the R bit, which is 0 or 1, is added to the counter for each clock interrupt

- Problem: it never forgets anything (like an elephant)
  - Old pages cannot be removed even they are not used recently

Simulating LRU in Software (2)

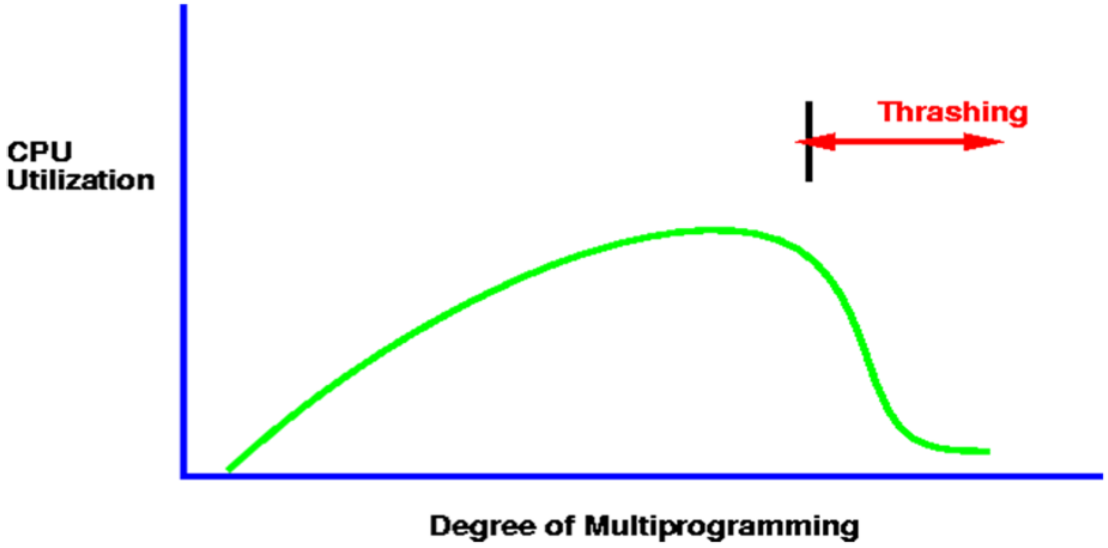
- Aging: A slight modification to NRU
  - The counters are each shifted right 1 bit before the R bit is added in
  - The R bit is added to the leftmost rather than the rightmost bit



How does aging compare to LRU?

Thrashing and CPU Utilization

- As the page fault rate goes up, processes get suspended on page out queues for the disk
- System may try to optimize performance by starting new jobs
  - But is it always good?
- Starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests
- System throughput plunges

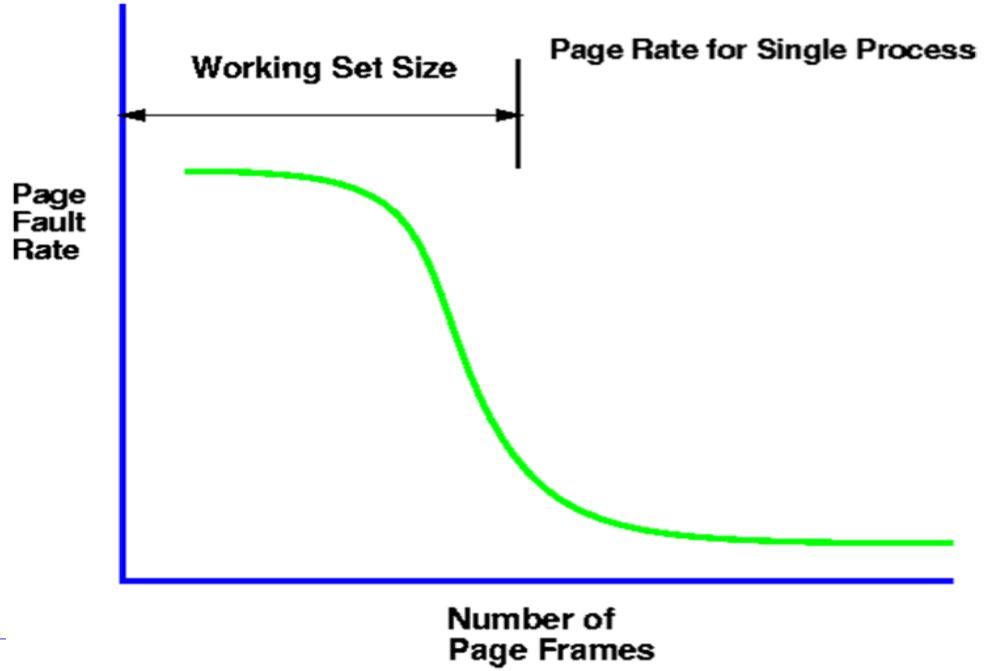


Fixed vs. Variable Space

- In a multiprogramming system, we need a way to allocate memory to competing processes
- Problem: How to determine how much memory to give to each process?
  - Fixed space algorithms
    - Each process is given a limit of pages it can use
    - When it reaches the limit, it replaces from its own pages
    - Local replacement
      - Some processes may do well while others suffer
  - Variable space algorithms
    - The page set for each process grows and shrinks dynamically
    - Global replacement
      - One process can ruin it for the rest

Working Set

- The working set model assumes locality
- The principle of locality states that a program clusters its access to data and text temporally
- *As the number of page frames increases above some threshold, the page fault rate will drop dramatically*



Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 1960s
- Definition
  - WS(t)={pages P such that P was referenced in the time interval (t_k, t)}
  - t – time, k – working set window (measured in page refs)
  - T_k – some time that between t_k and t, there is a k
  - Note: k may change over time
- A page is in the working set (WS) only if it was referenced in the last k references

Working Set Size

- The working set size is the number of pages in the working set
  - Note: the working set size is not the working set window k
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages that a process needs in memory to prevent heavy faulting
  - Each process has a parameter w (working set window) that determines a working set with few faults – thus determines the interval (t_k, t)
  - Denning: Don't run a process unless working set is in memory

Working Set Size

- The working set size increases with the increase of working size window (k)
  - Explain the trend: why is the increase non-linear?

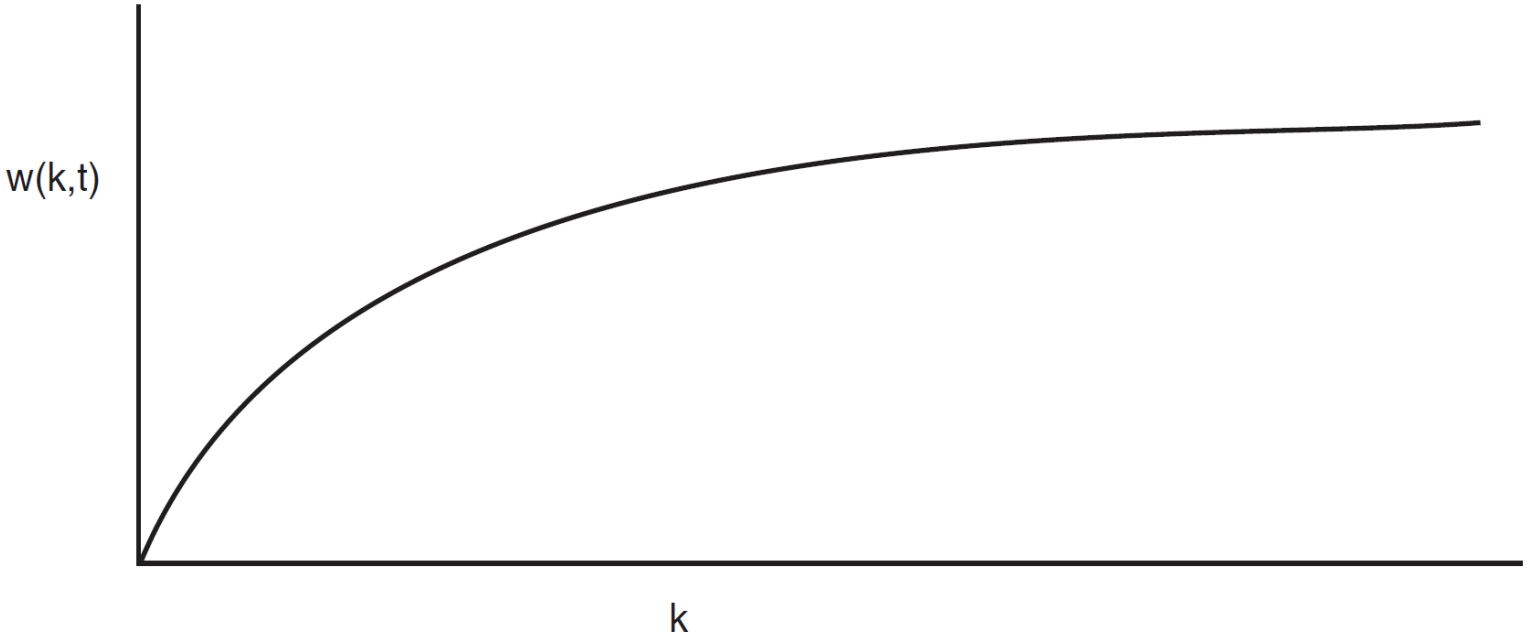
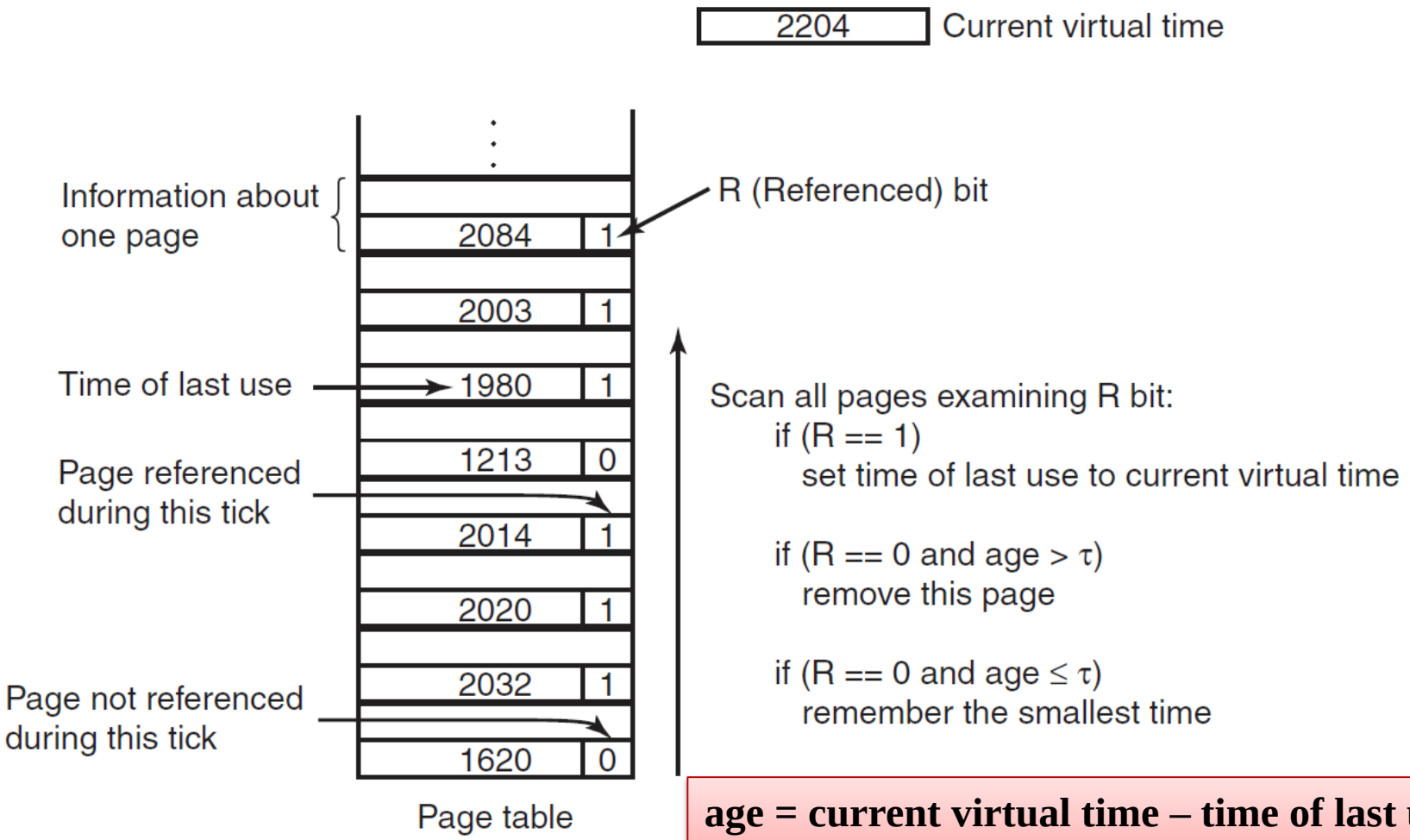


Figure 3-18. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

Working Set in Action

- Algorithm
  - If # free page frames > working set of some suspended process, then activate process and map in all its working set (pre-paging)
  - If working set size of some process increases and no page frame is free, suspend process and release all its pages
- Moving window over reference string used for determination
- Keeping track of the working set
  - Approximate working set model using timer and reference bit
  - The working set of a process becomes the set of pages it has referenced during the past τ seconds of virtual time.
  - Remember the last time of use and use a Referenced bit
  - A periodic clock interrupt is assumed to cause software to run that clears the Referenced bit on every clock tick.

Working Set example

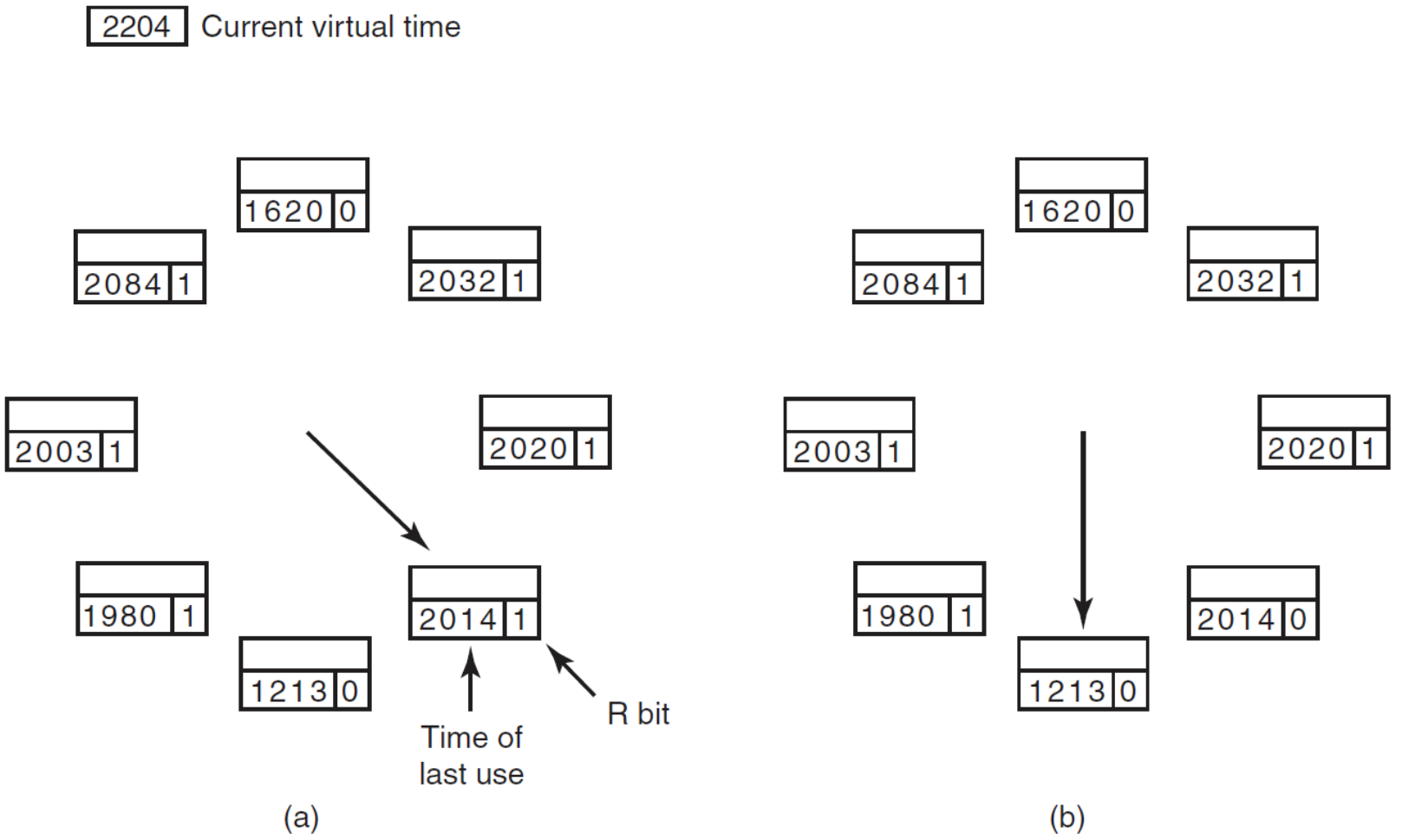


age = current virtual time – time of last use
τ : the size of the working set window (time)

WSClock

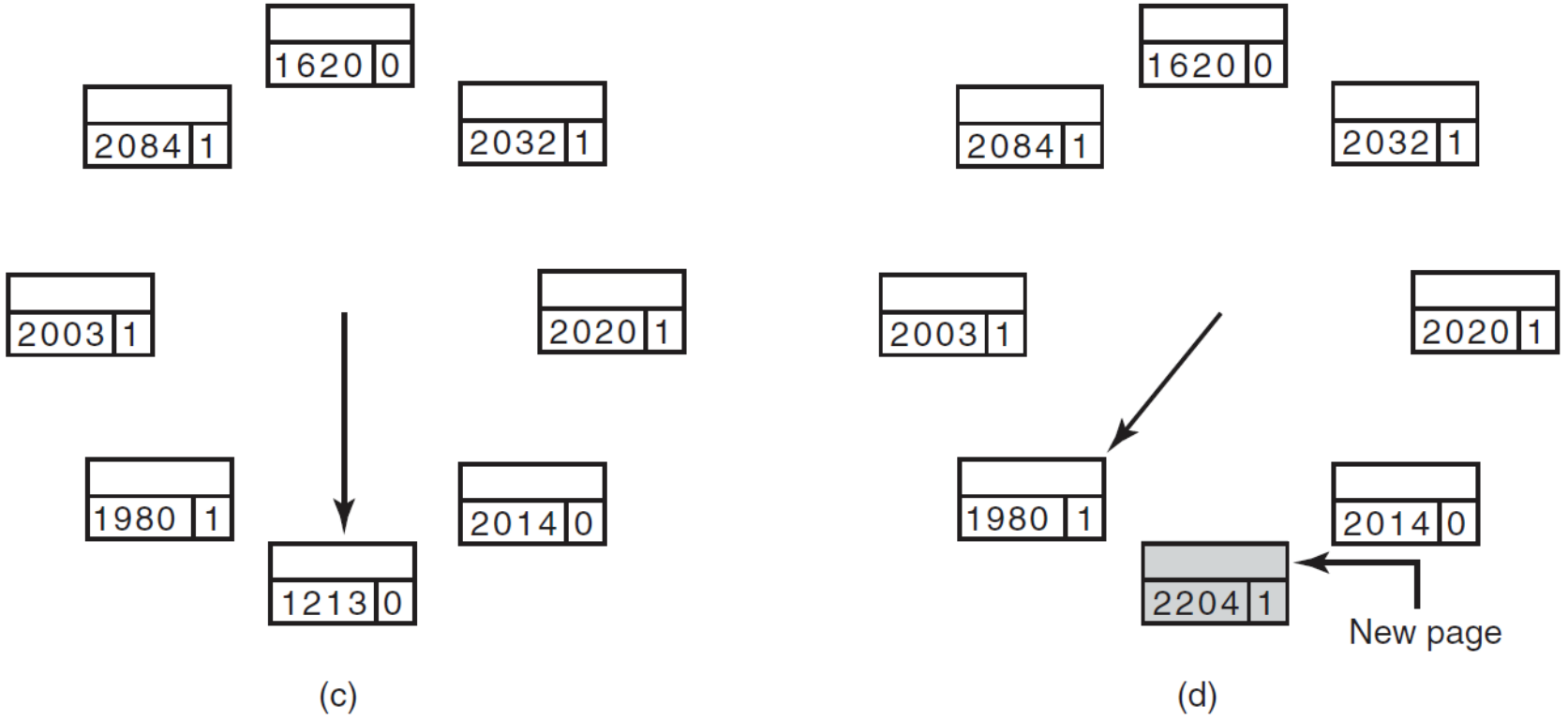
- Combining Working Set and Clock
  - (Carr and Hennessey, 1981)
  - Simplicity of implementation and good performance
  - widely used in practice
- Each entry contains
  - the Time of last use field from the basic working set algorithm
  - the R bit and the M bit
- Why is WSClock better than the original working set algorithm?

WSClock Example (1)



R=1: clear it and go to the next page

WSClock Example (2)



R=0 : replace it

WSClock in Action

- If the age of a page is greater than τ and the page is clean, it is claimed.
- What happens if it is dirty (modified)?
  - Schedule the write to disk and continues looking.
- What happens if no pages can be found in the end?
  - If at least one write has been scheduled => keep looking
  - If no writes have been scheduled (all pages are in the working set) => claim any clean page or choose the current page

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

This Lecture

Page Replacement

Demand paged virtual memory

Page replacement algorithms

Design issues

Implementation Issues

Local versus Global Allocation Policies

- Local algorithms: allocating every process a fixed fraction of the memory
  - If the working set grows, thrashing will occur
  - If the working set shrinks, it wastes memory
- Global algorithms: dynamically allocate page frames among the runnable processes
  - the number of page frames assigned to each process varies in time
  - must continually decide how many page frames to assign to each process
    - Can use the PFF (Page Fault Frequency) as an indicator

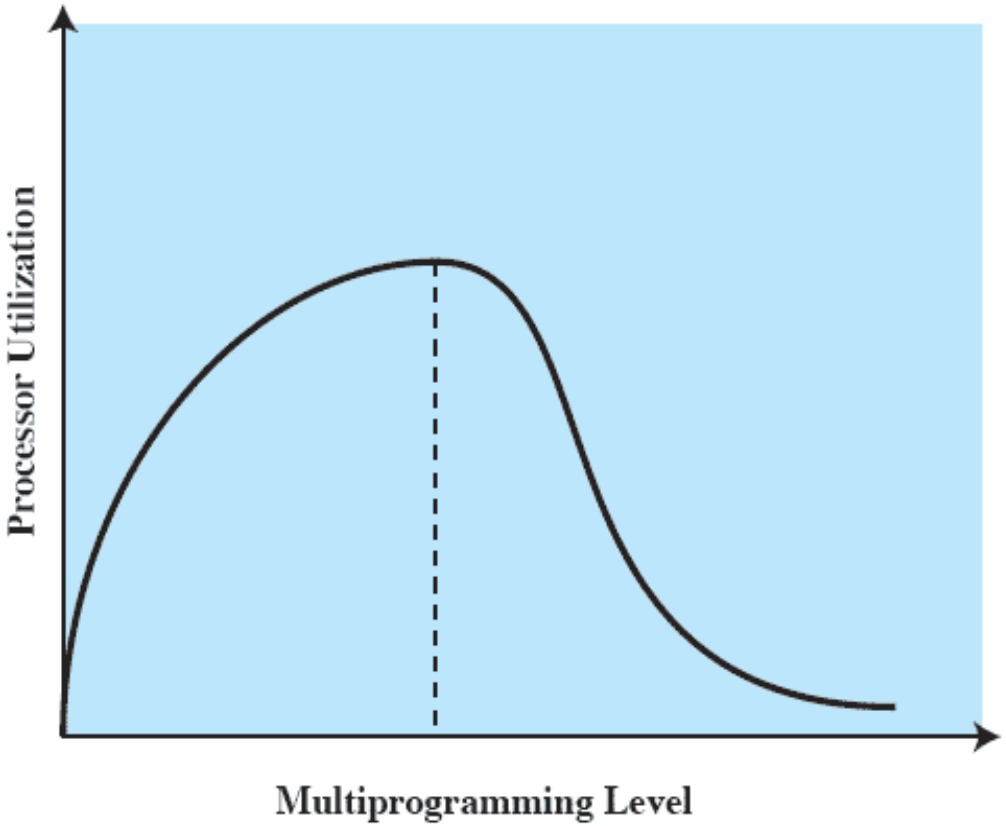
Local versus Global Allocation Policies

- Some algorithms can be used for both local and global policies

- Some algorithms can only be used for local policies

Load Control

- Load control: reduce the number of processes competing for memory when the working set is too big
- Swapping out the processes to disk
- Consider the degree of multiprogramming



Separate Instruction and Data Spaces

- Separate address spaces for instructions (program text) and data, called I-space and D-space, respectively

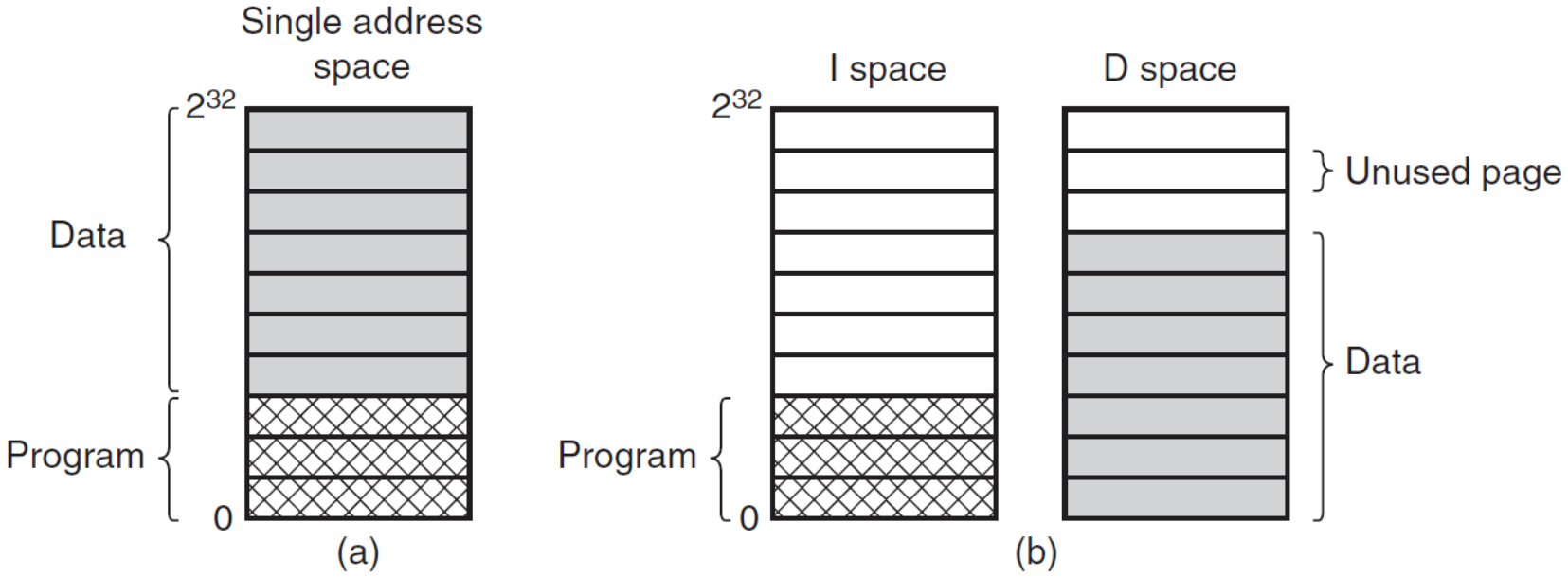


Figure 3-24. (a) One address space. (b) Separate I and D spaces.

Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want
- But this makes it difficult to share data
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
- We can use shared memory to allow processes to share data using direct memory references
  - Both processes see updates to shared memory segment
    - Process B can immediately read an update by process A
  - How are we going to coordinate accesses to shared data?

Sharing (2)

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid
- Can map shared memory at the same or different virtual addresses in each process' address space
  - Different: flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same: less flexible, but shared pointers are valid
- What happens if a pointer inside the shared segment references an address outside the segment?

Copy on Write

- OSs spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in child's page table, restart write instruction
  - How does this help fork()?

Shared Libraries

- Sharing libraries between processes is straightforward
- What if different addresses are used in two processes?

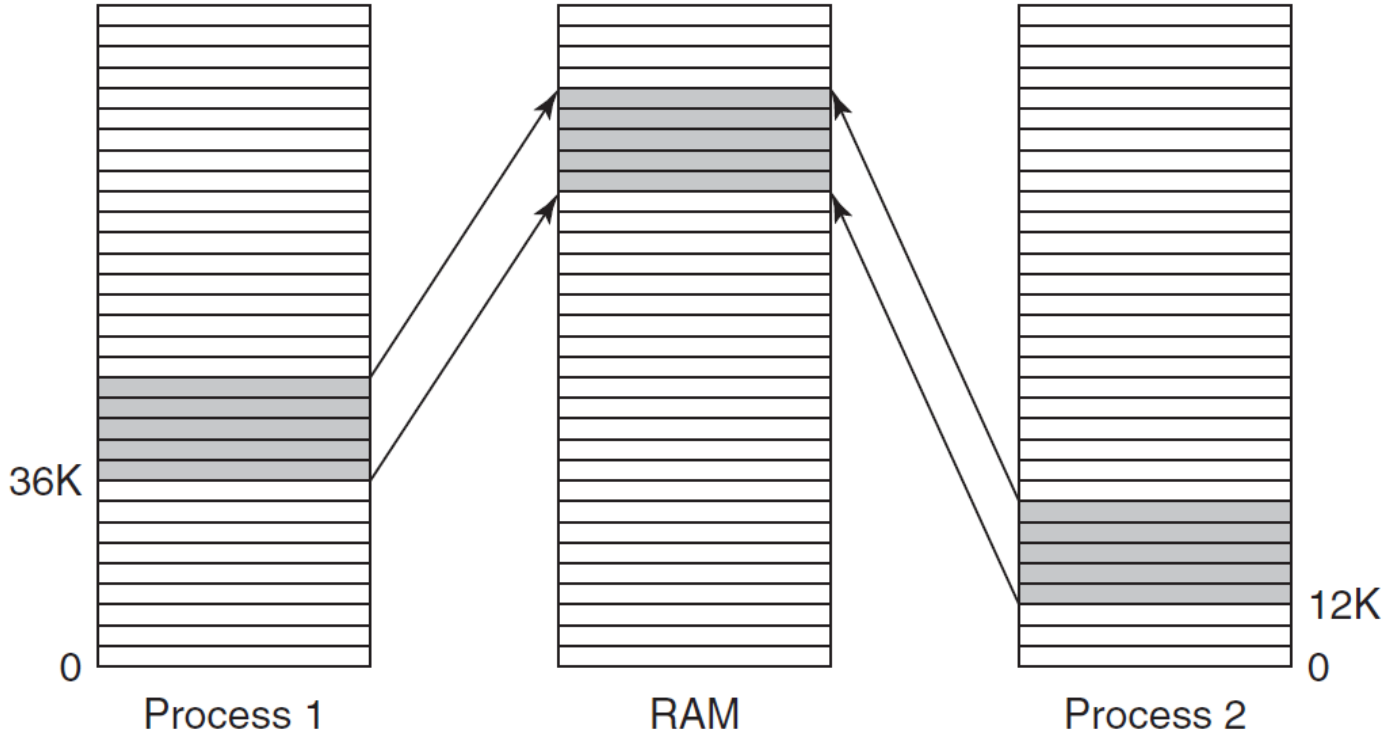


Figure 3-26. A shared library being used by two processes.

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
- Compare: virtual address space not backed by “real” files also called Anonymous VM
  - Initially backed by zero page (return only zeroes on read; write prohibited by flag)

Mapped Files (2)

- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE
- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying between files and MEM (using VM mechanisms)
- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)

Cleaning Policy

- Expectation: When page faults occur, there should be adequate free page frames available
  - Otherwise, the page fault penalty will be very high
- Method: A paging daemon process
  - Checking periodically to make sure there are enough free page frames
  - If not, select pages to evict using replacement algorithm
  - Make sure all free frames are “clean”
  - Apply a two-hand clock algorithm
    - Front hand saves a dirty page to disk
    - Back hand used for page replacement, no write back needed

This Lecture

Page Replacement

Demand paged virtual memory

Page replacement algorithms

Design issues

Implementation Issues

Locking Pages in Memory

- Some pages cannot be removed from memory
  - For example, when a page containing the I/O buffer is chosen to be removed from memory, and the I/O process is doing DMA transfer…
- Solutions
  - One solution is to lock pages engaged in I/O in memory so that they will not be removed.
    - Locking a page is often called pinning it in memory.
  - Another solution is to do all I/O to kernel buffers and then copy the data to user pages later.

Separation of Policy and Mechanism

- Split policy from mechanism is an important tool for managing the complexity of any system

- The memory management system can be divided into three parts:
  - 1. A low-level MMU handler (machine-dependent)
  - 2. A page fault handler that is part of the kernel (mechanisms)
  - 3. An external pager running in user space (policies)

Separation of Policy and Mechanism

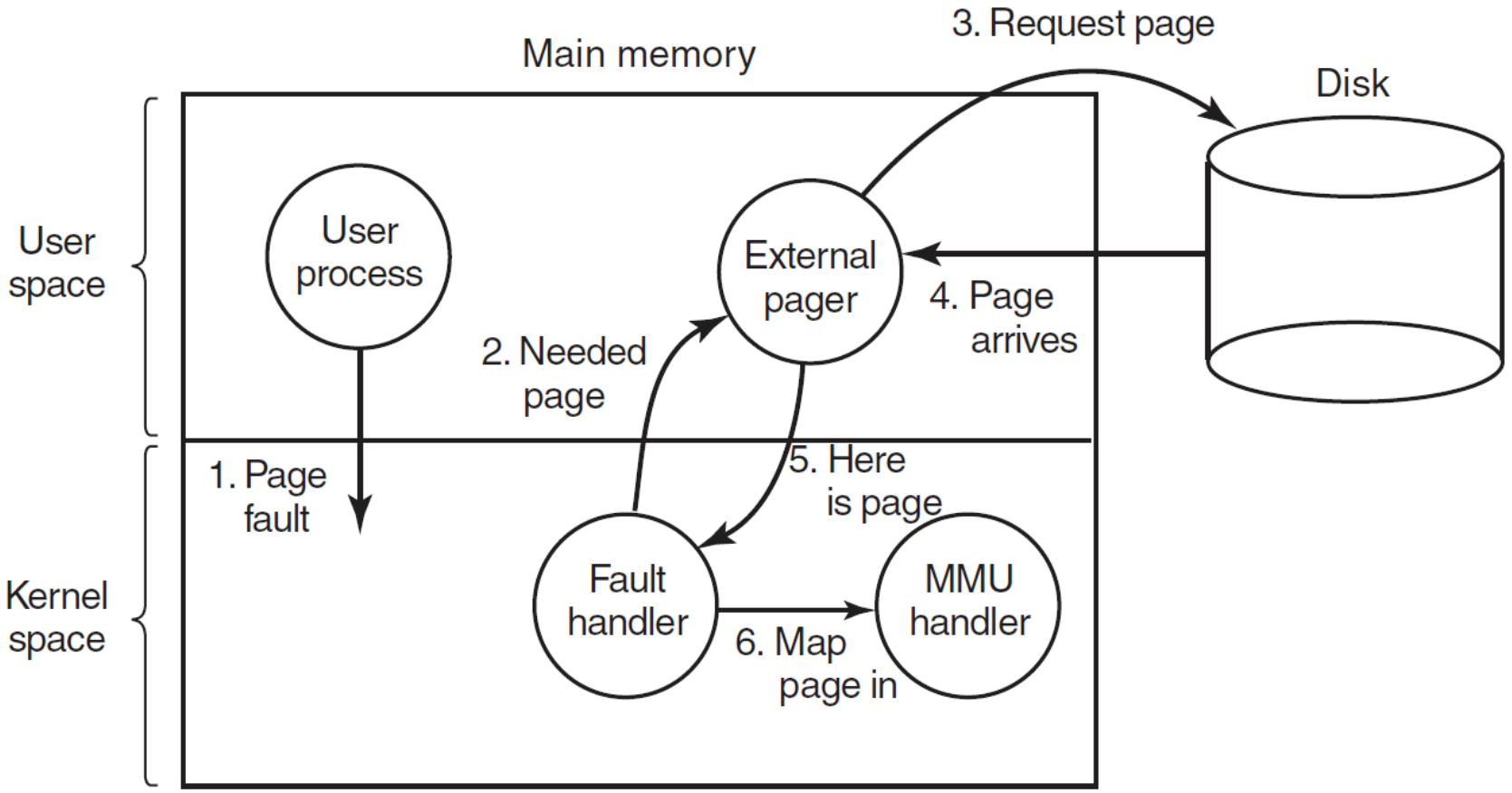


Figure 3-29. Page fault handling with an external pager.

Summary

- Demand paged virtual memory
- Page replacement algorithms
  - Optimal (Belady's)
  - FIFO, SCR, NRU, LRU, Clock, WS, WSClock
- Design & implementation issues

- Next lecture: Windows VM