SUPSI

Page replacement algorithms

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Study common page replacement algorithms
- Get an overview of memory management primitives

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

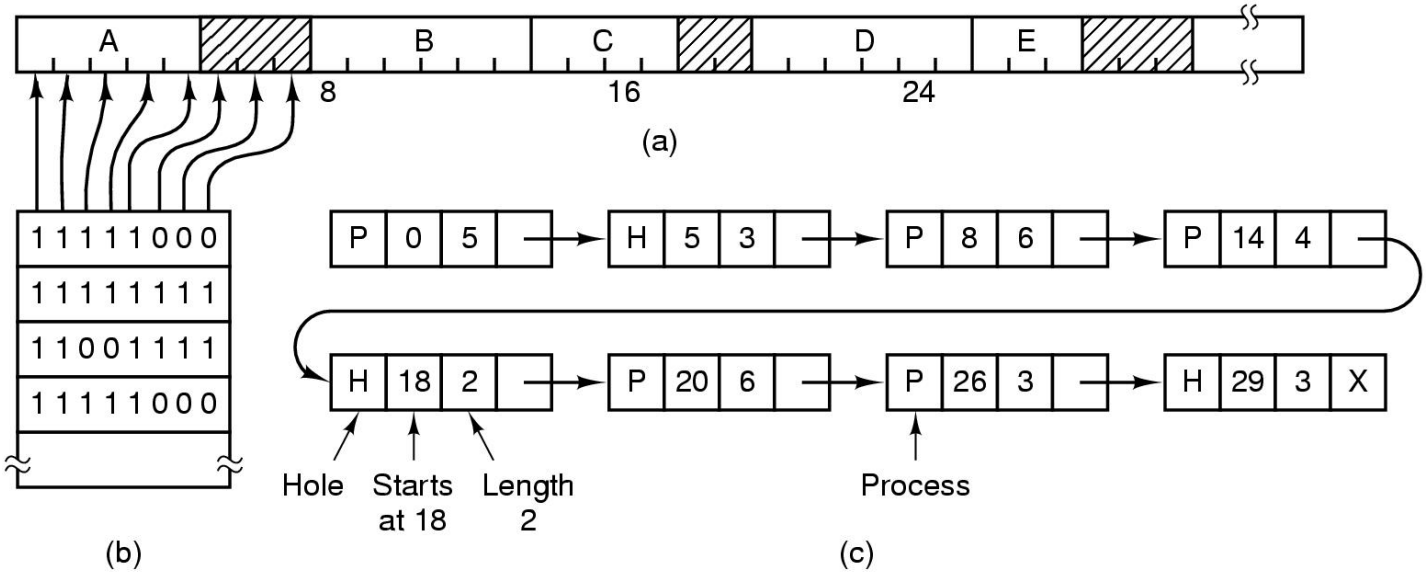
Looking for some space

- The operating system requires physical memory…
  - …when a page needs to be swapped in (major page fault)
  - …when a process (or the kernel itself) allocates new memory

- Two situations are possible
  - we have some space left: *where is it?*
  - memory is full: *we need to replace some of the pages*

Where to swap in: finding an empty spot

- To keep track of free pages the OS can use:
  - **a bitmap** (0 = unallocated, 1 = allocated) *(Figure b)*
  - **a linked list** *(Figure c)*



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Finding an empty spot: algorithms

- To find a suitable space the operating system can use different approaches:
  - **First fit**: the first empty space that fits the allocation request is used
  - **Next fit**: like first fit, but will not restart from the beginning at each request
  - **Best fit**: search from the start to the end, return the smallest memory range that fits the request
  - **Worst fit**: search from the start to the end, return the biggest memory range that fits the request
  - **Quick fit**: keep separate allocation lists (for example: 4KB, 8KB,...), return the closest amount

Page replacement algorithms

- When a page fault happens and the operating system needs to free some memory an algorithm chooses which page can be swapped-out:
  - examples:
    - **FIFO** (First In First Out)
    - **Second chance**
    - **LRU** (Least Recently Used)
    - **Working Set**
- Page replacement algorithms can look either in all page tables (**global page replacement**) or just in the page table of the process that generated the page fault (**local page replacement**)

The optimal algorithm

- An optimal algorithm tries to minimize page faults
  - ... but this will require some oracle to **predict future requests**...
  - ... or a **profiling mechanism** that analyzes the behavior of a program and records page requests for future use...provided that the process behaves in the same way!

- In general this is not possible, so we will only study some *approximations* of such an optimal algorithm

NRU (Not recently used)

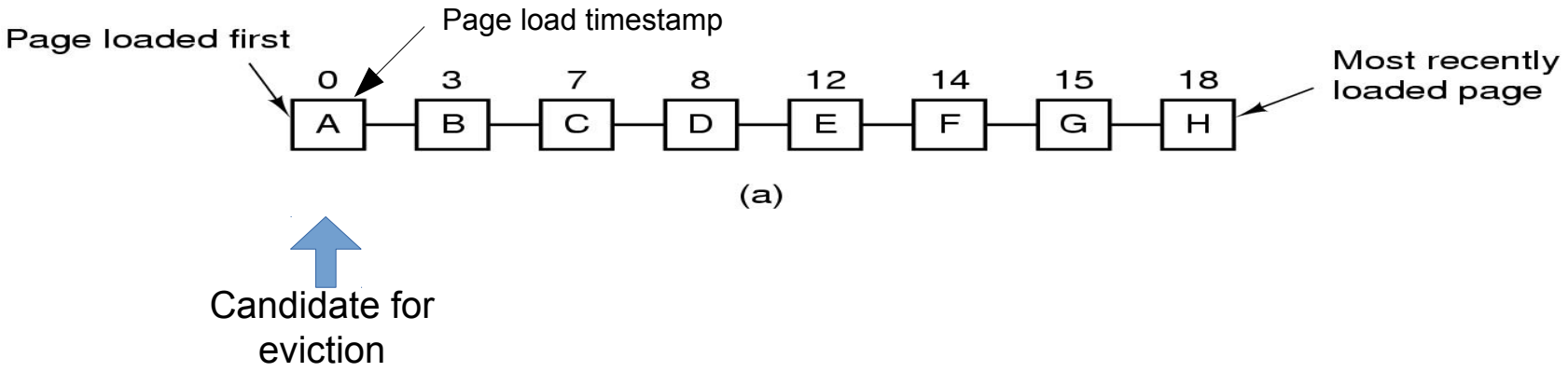
- The NRU algorithm employs the R and M bits of each page table entry:
  - Newly allocated pages have the R and M bits both set at 0
  - Periodically the R bit is cleared by the operating system
  - When a we need to replace a page, choose from the following classes (lower numbers equal most suitable choices)
    - **Class 0: not referenced (R=0), not modified (M=0)**
    - **Class 1: not referenced (R=0), modified (M=1)**
    - **Class 2: referenced (R=1), not modified (M=0)**
    - **Class 3: referenced and modified (R=M=1)**

Dirty pages

- Page eviction does not always come for free...
  - If the data on the page has not been modified (*M=0*) we can remove it from memory without any problem
  - If the data on the page has been modified (*M=1*) we need to first save those changes to disk before removing the page
- Pages with **M=1** are called "**dirty**" and are more expensive to swap-out (since they involve a disk operation)

FIFO

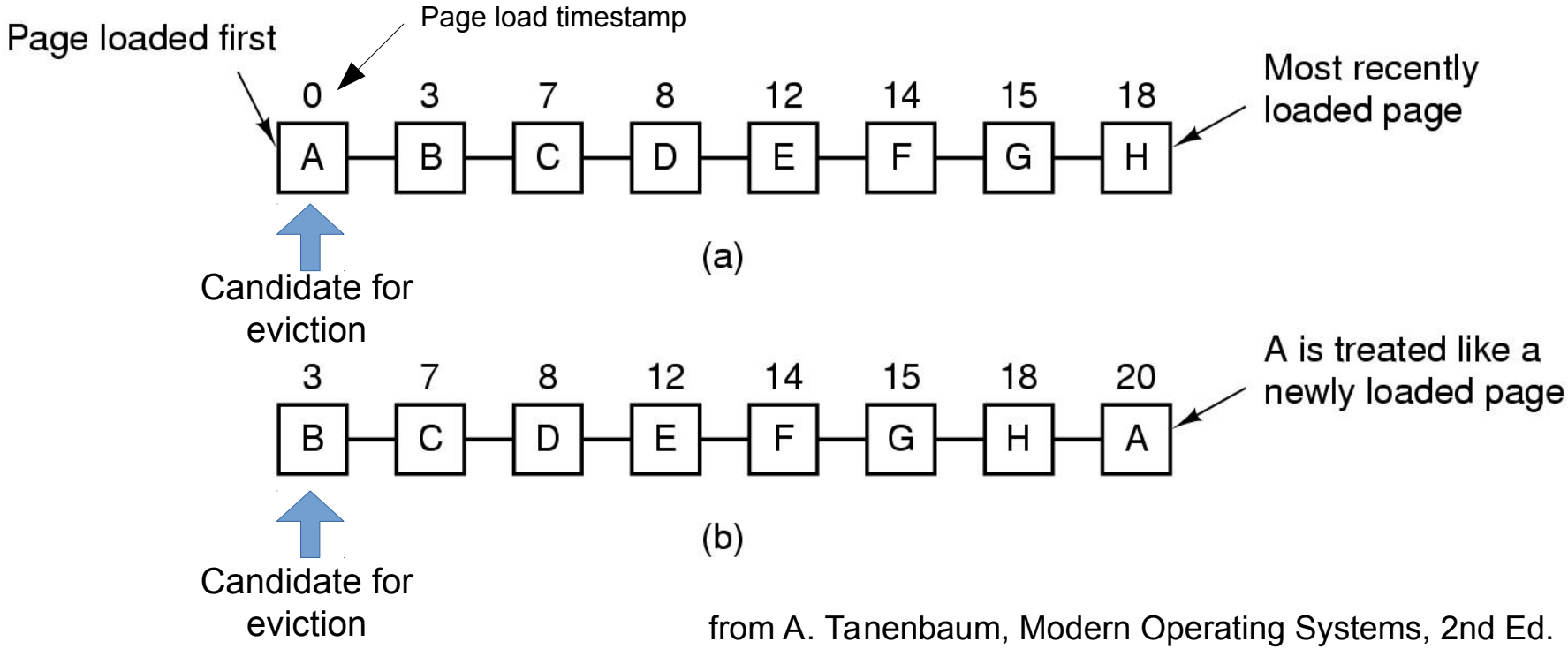
- We store the page loading order in a FIFO
  - The first loaded page is the first to be evicted



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Second chance

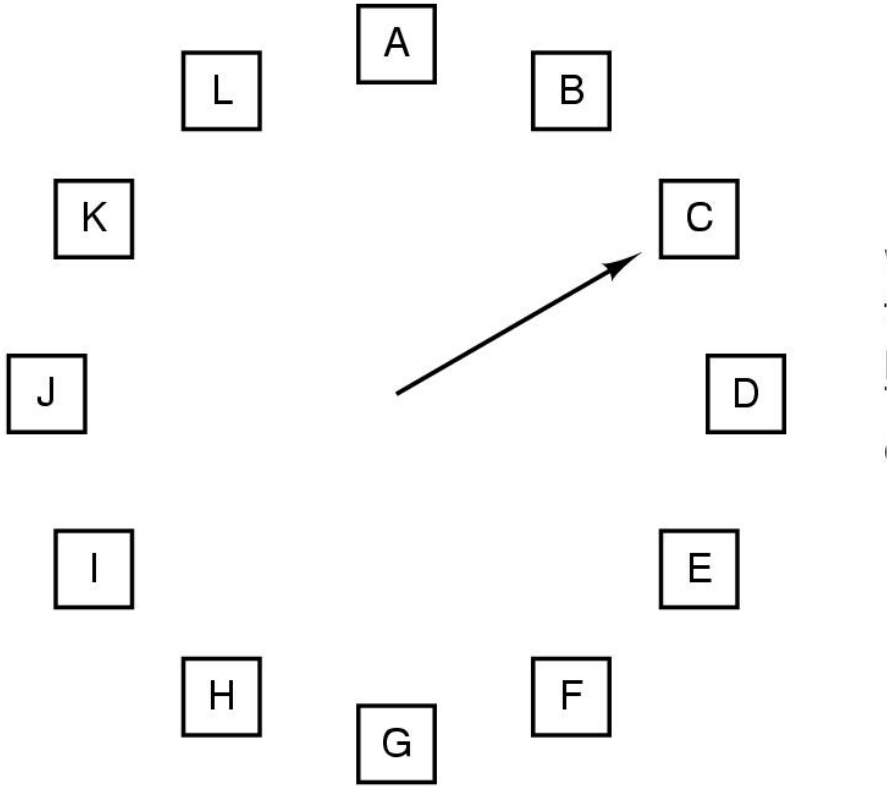
- To avoid evicting a frequently used page we can extend FIFO by considering the **R** bit when the page becomes a candidate for eviction:
  - If the **R** bit is set to 1, we reset it to 0 and put the page on the back of the list
  - If the **R** bit is set to 0, the page is evicted



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

The clock algorithm

- The clock algorithm is an efficient implementation of second chance
  - Does not require updating too many pointers



When a page fault happens, check the page under the hand. If **R=0** evict tha page, otherwise reset **R** and move hand forward.

from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

LRU (Least Recently Used)

- Due to the principle of locality, a page that has been recently used has a higher chance of being used in the near future
  - as such, recently used pages are a poor candidate for eviction

- The idea behind **LRU** (**L**east **R**ecently **U**sed) is to evict pages which have not been recently used

How to determine if a page has been recently used?

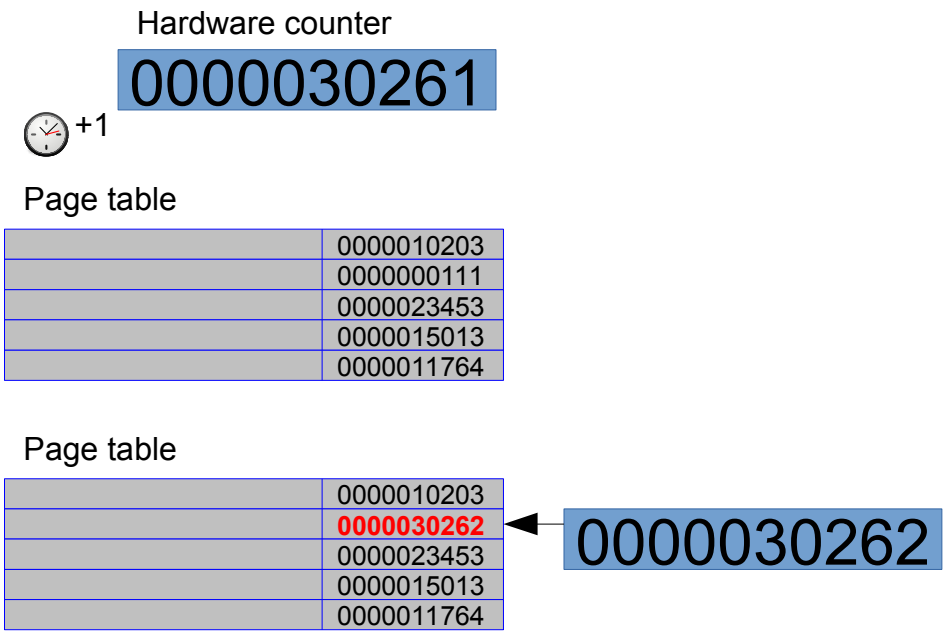
- **With a linked list**
  - Head: most recently used pages, Tail: least recently used pages
  - Problem: we need to update the list at each memory access!

- **With a timestamp associated to each page**
  - Timestamp is updated at each memory access
  - Smallest timestamp = Least recently used page

... too expensive! But we might approximate it...

LRU in hardware (with a counter)

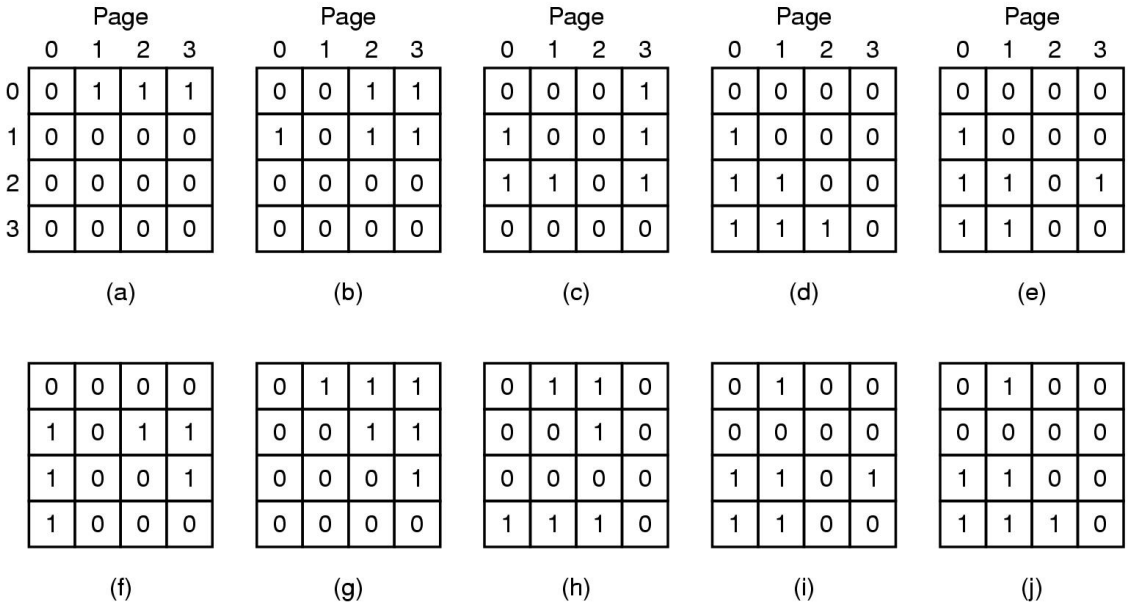
- We assume that we have an hardware 64bit counter which is incremented after each CPU instruction
- Each page table entry contains a field big enough to store the counter value
- After each memory reference, the current counter value is stored in the corresponding page table entry field
- Pages with the lowest counter value in they page table entry are candidate for eviction



LRU in hardware (with a matrix)

- Instead of a counter we can use a square matrix n x n bits, initialized to 0, where n is the number of page frames
- When the page loaded on **frame k** is referenced, all bits on **row k** are **set to 1** and subsequently all bits on **column k** are **set to 0**
- The row with the lowest binary value determines the least recently used frame (and page)

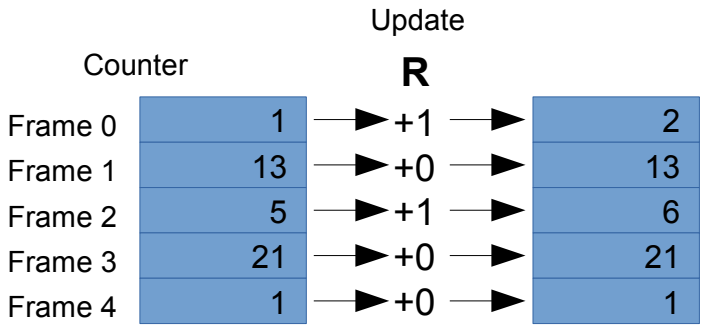
References:
0 1 2 3 2 1 0 3 2 3



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

LRU in software: NFU (Not Frequently Used)

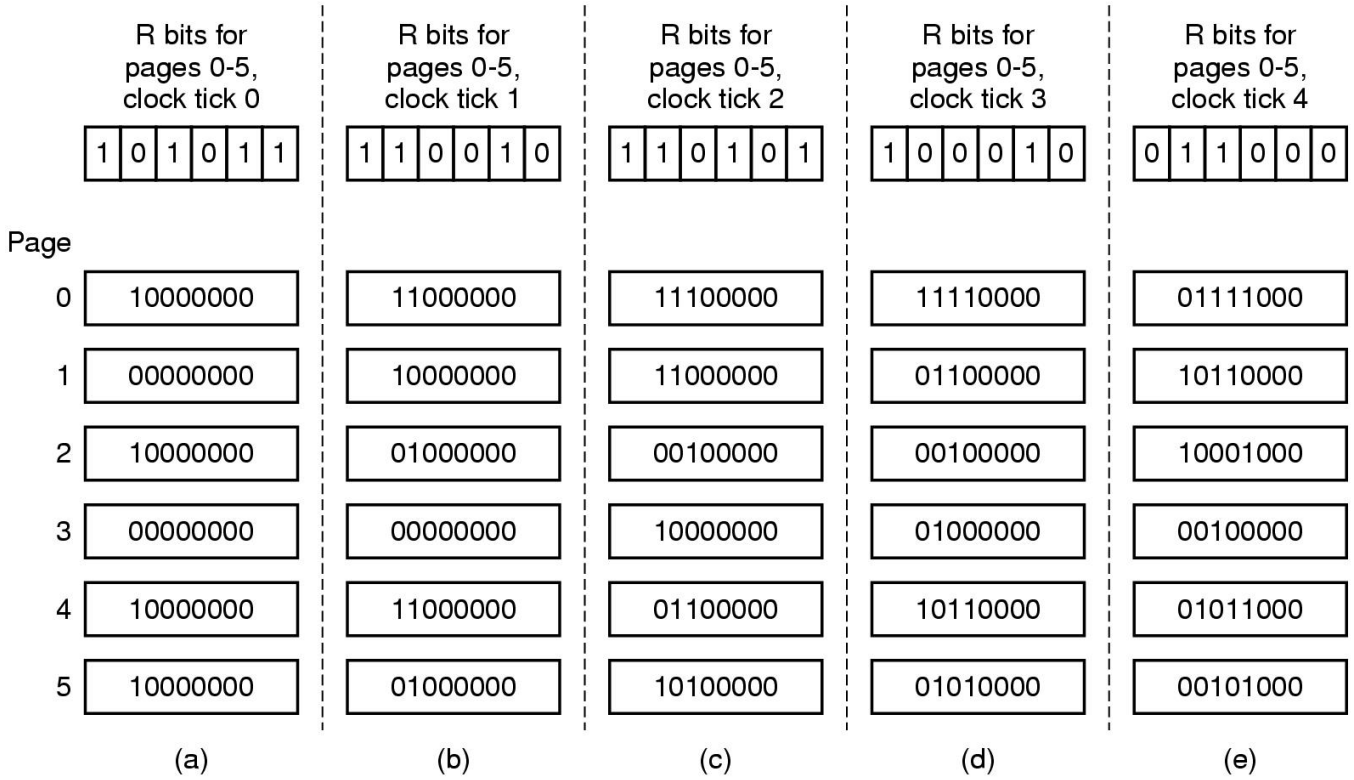
- **NFU** (**N**ot **F**requently **U**sed) approximates LRU without hardware support
- Each page frame has a counter, initially set at 0
- Periodically the value of the R bit is added to this counter (+ 0 or + 1) and R is set to 0
- The page frame with the lowest counter value contains the candidate for eviction



Catch: if a page has been referenced many times in the initial phase of the execution of a program, it will take some time before it gets evicted...

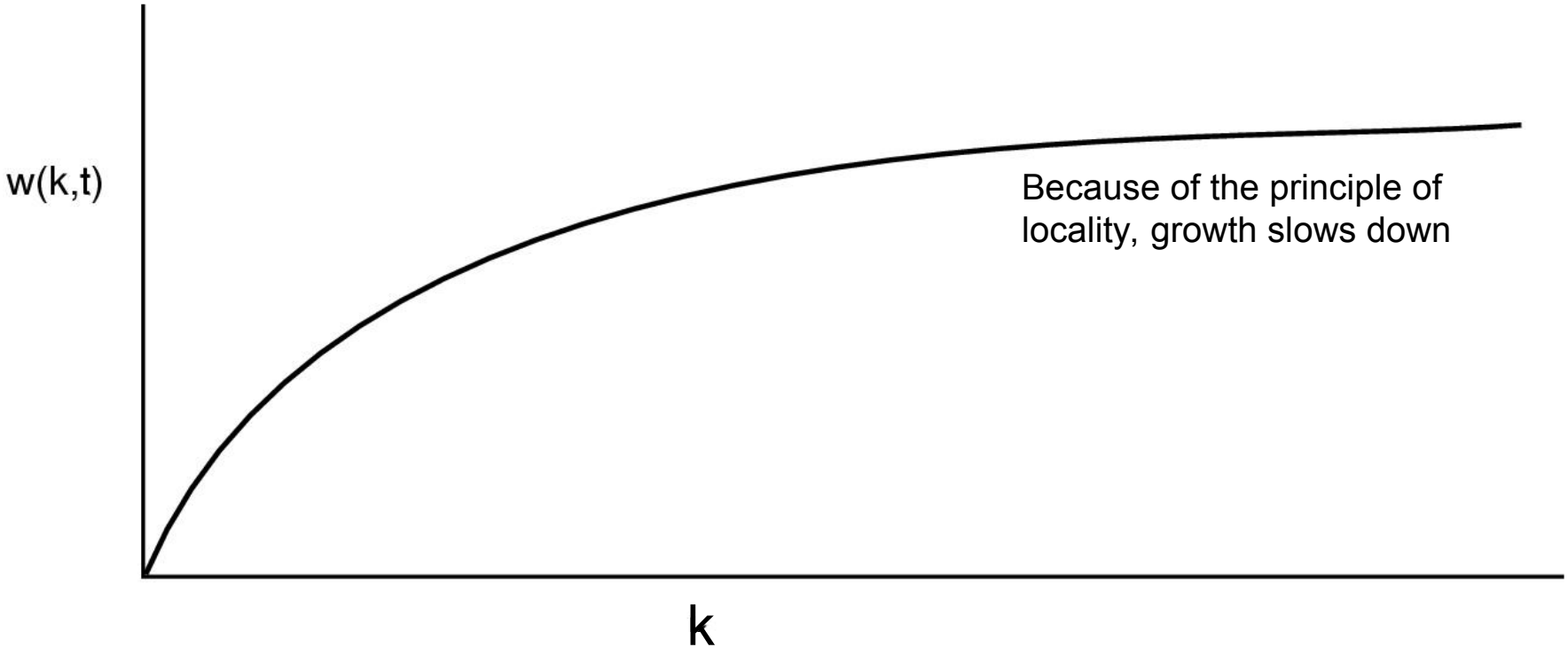
NFU (modified)

- We can implement an **aging** mechanism:
  - before adding the R bit we shift all counters to the right
  - we add the R bit on the left



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Optimize paging: the working set model



Working set: set of pages used by the k most recent memory addresses; **w(k,t)** is the size of the working set at time k

from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Optimize paging: what if we know what's in the working set

- We can move from **demand paging** (where pages are loaded when they are first referenced) to implement
  - a **prepaging** approach (load the pages before the process asks for them)
    - If the process is suspended and swapped out we know which pages must be reloaded
  - a page replacement algorithm which takes into account the working set and the **principle of locality**

- ...but how big is the working set (what's the value of *k*) ?
- ...which pages belong to the working set?

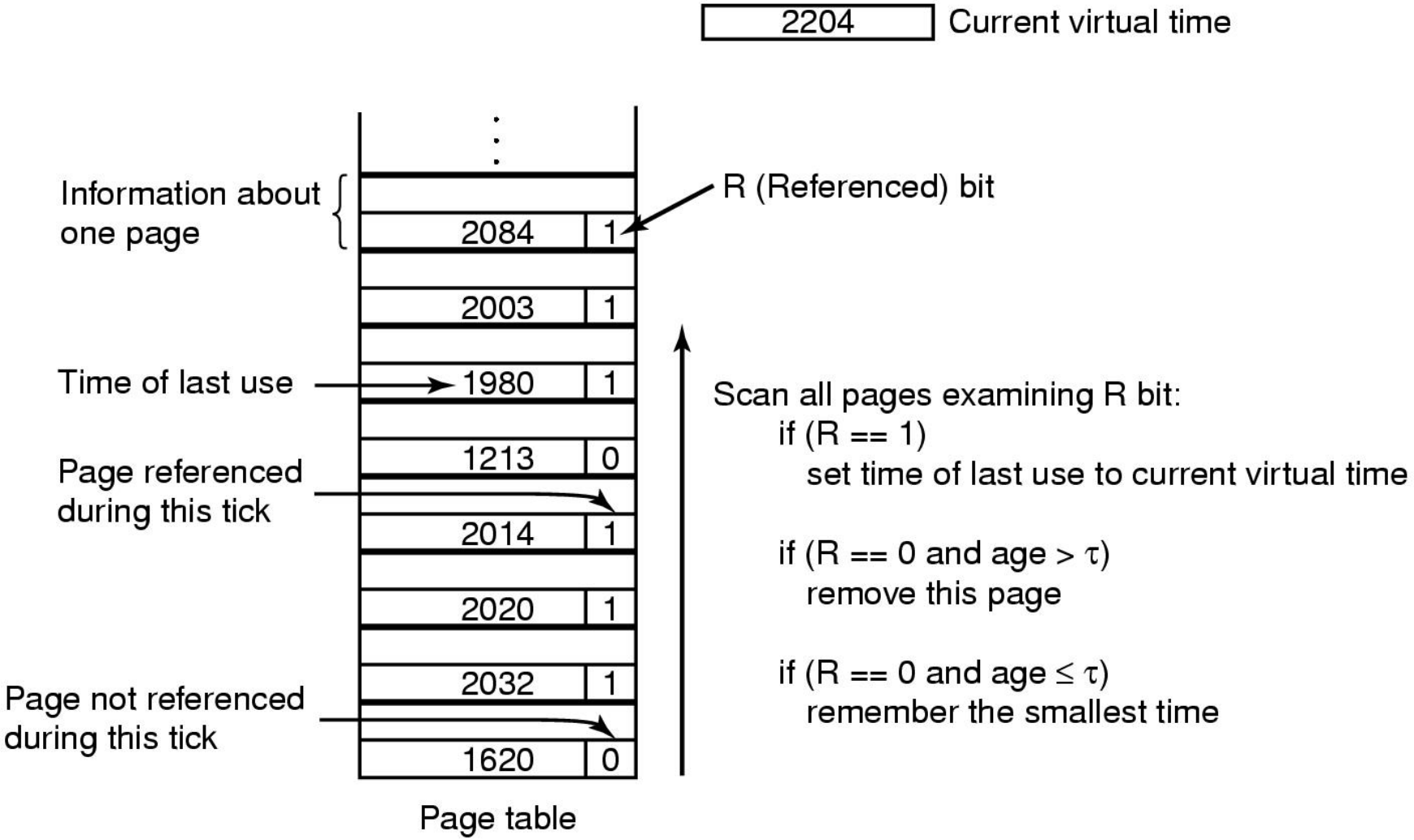
The working set algorithm

- We maintain a timestamp to keep track of the virtual execution time of a process
- In each page table entry a field is reserved to store such a timestamp

- When we need to remove a page, we search the page table:
  - if **R=1**, we copy the current timestamp into the entry and set R=0
  - if **R=0**, and the timestamp is older than a threshold τ the page is likely not part of the working set and becomes a candidate for eviction (the oldest candidate is subsequently chosen)

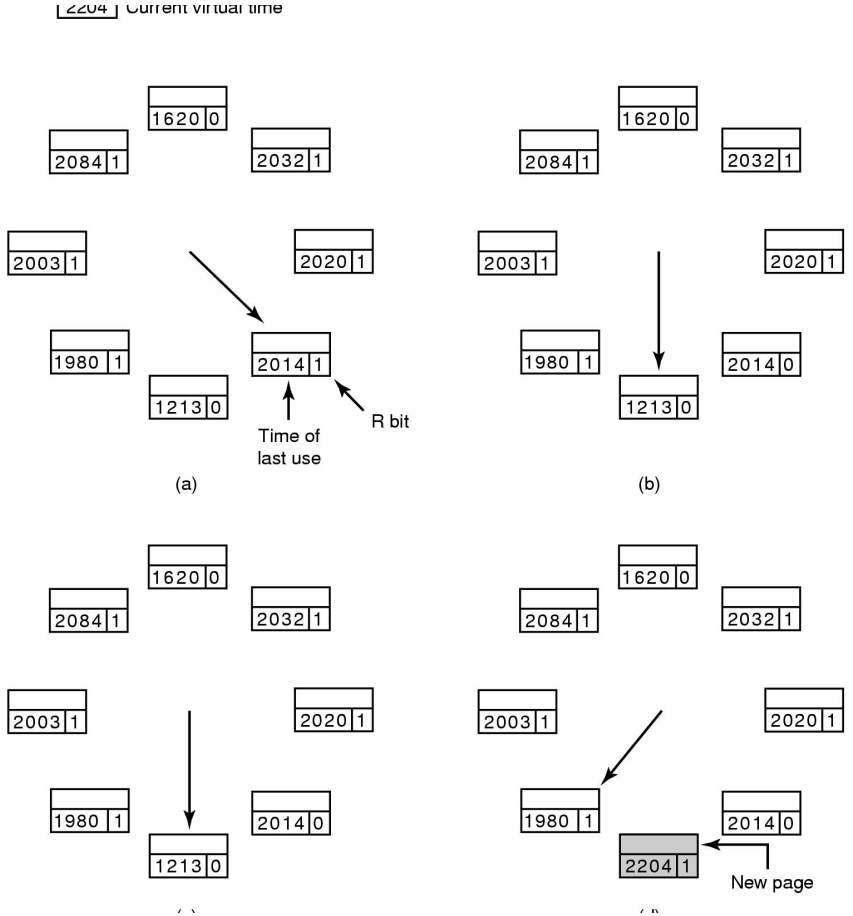
- If all pages had R=1, a random choice is performed

The working set algorithm: example



from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

Working set clock implementation (WSClock)



The working set algorithm can be efficiently implemented using a “clock” approach (called **WSClock**, **W**orking **S**et **Clock**)

from A. Tanenbaum, Modern Operating Systems, 2nd Ed.

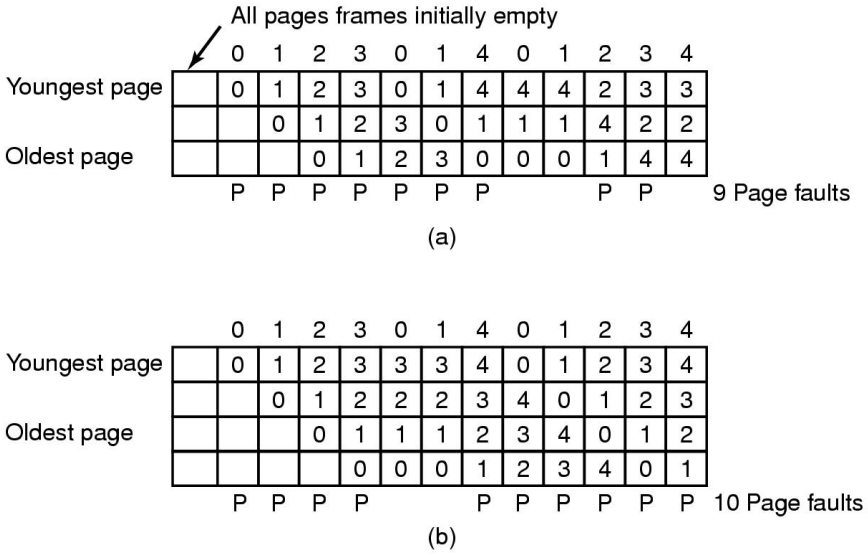
What if we continue to evict and reload pages

- Paging and page swapping is usually quick enough so the user won't notice...

- However if physical memory is not enough and page faults happen frequently (almost at every memory access), the operating system might start to do aggressive swapping
  - Since it involves heavy use of a slower secondary memory (harddisk) the whole system is affected
  - We call this situation **trashing**

When more memory leads to more page fault: Belady's anomaly

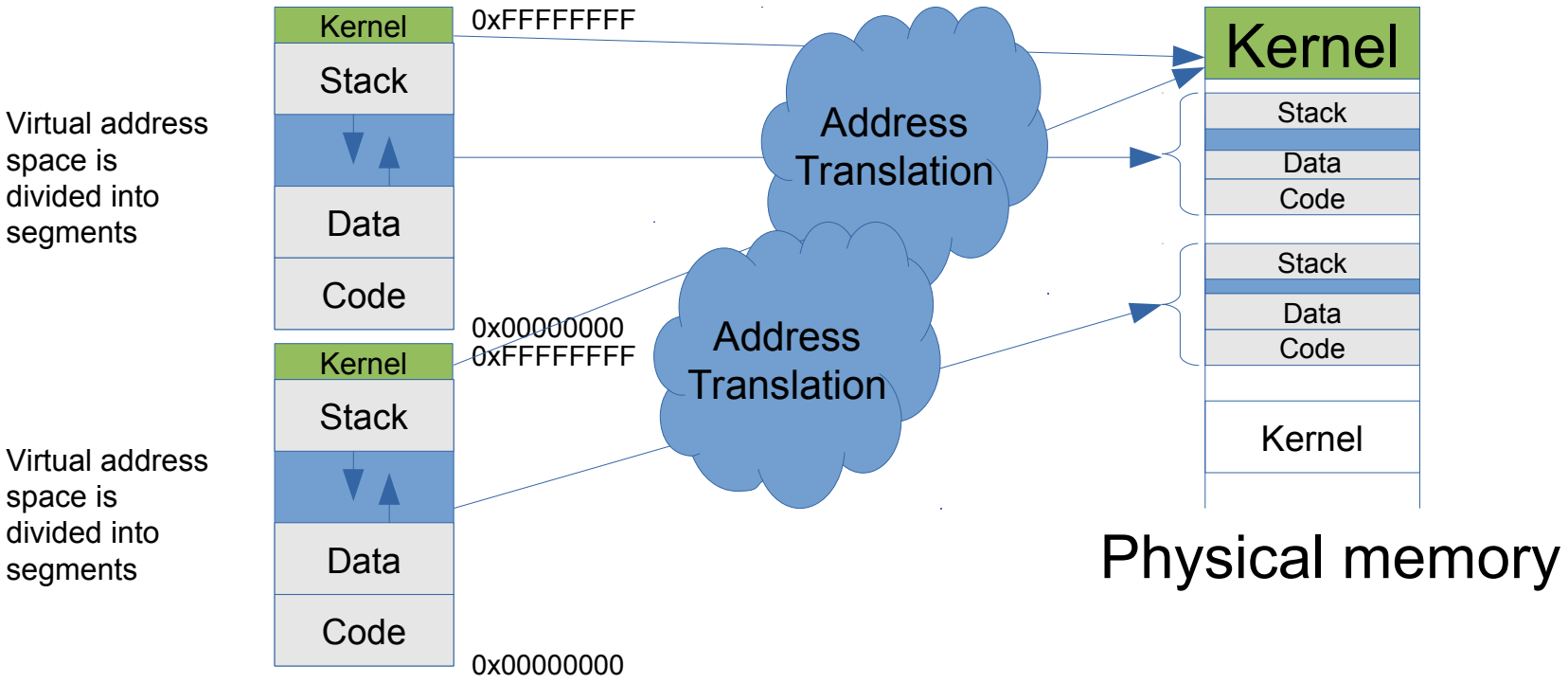
- The **Belady's anomaly** happens with FIFO (or second chance): with more memory (i.e. page frames) we end up with more page faults
- We can prevent it by using **stacking algorithms**, namely algorithms where the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames
  - for example: LRU



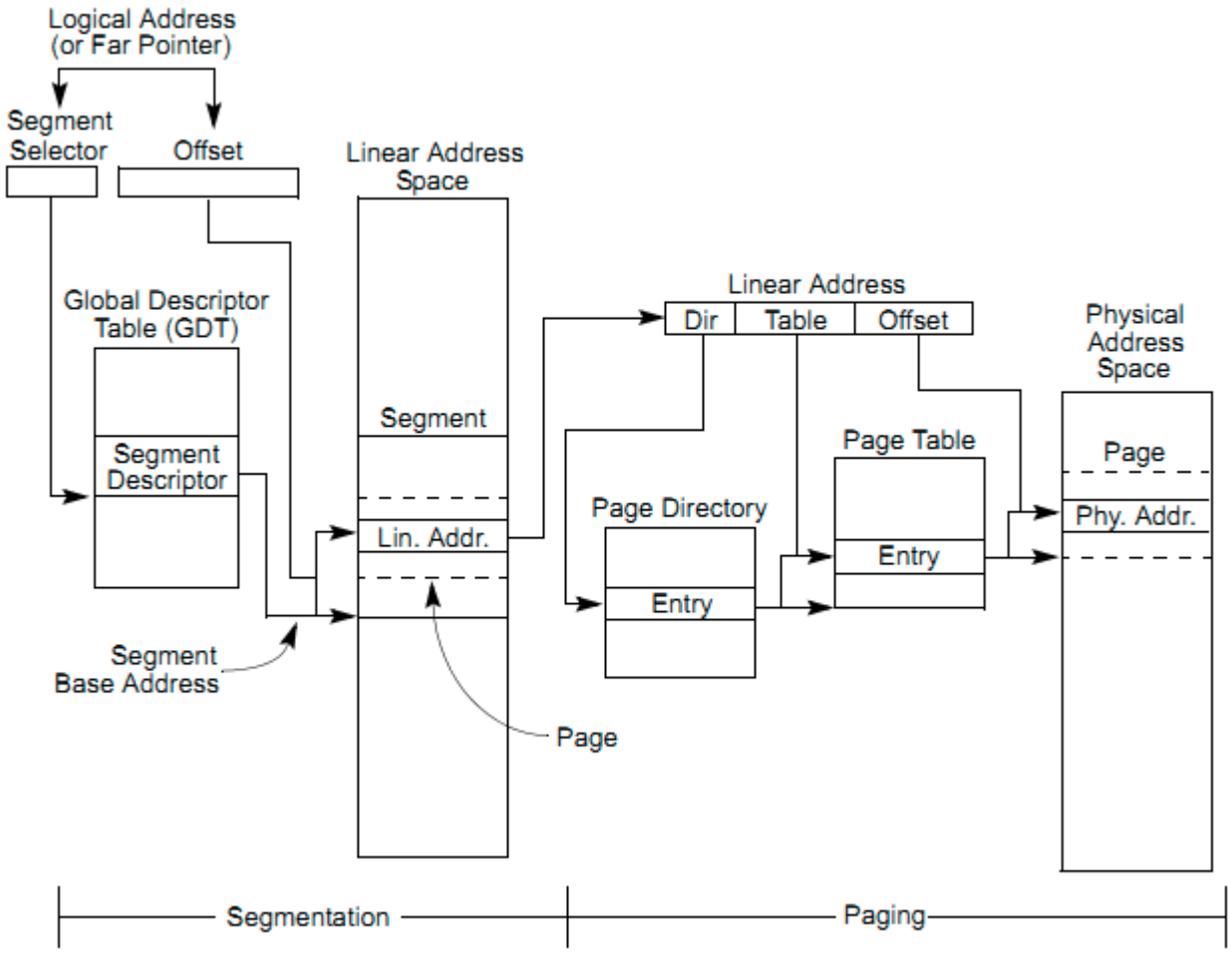
Example with FIFO

Segmentation and paging on the same system

- Segmentation can be enabled along with paging

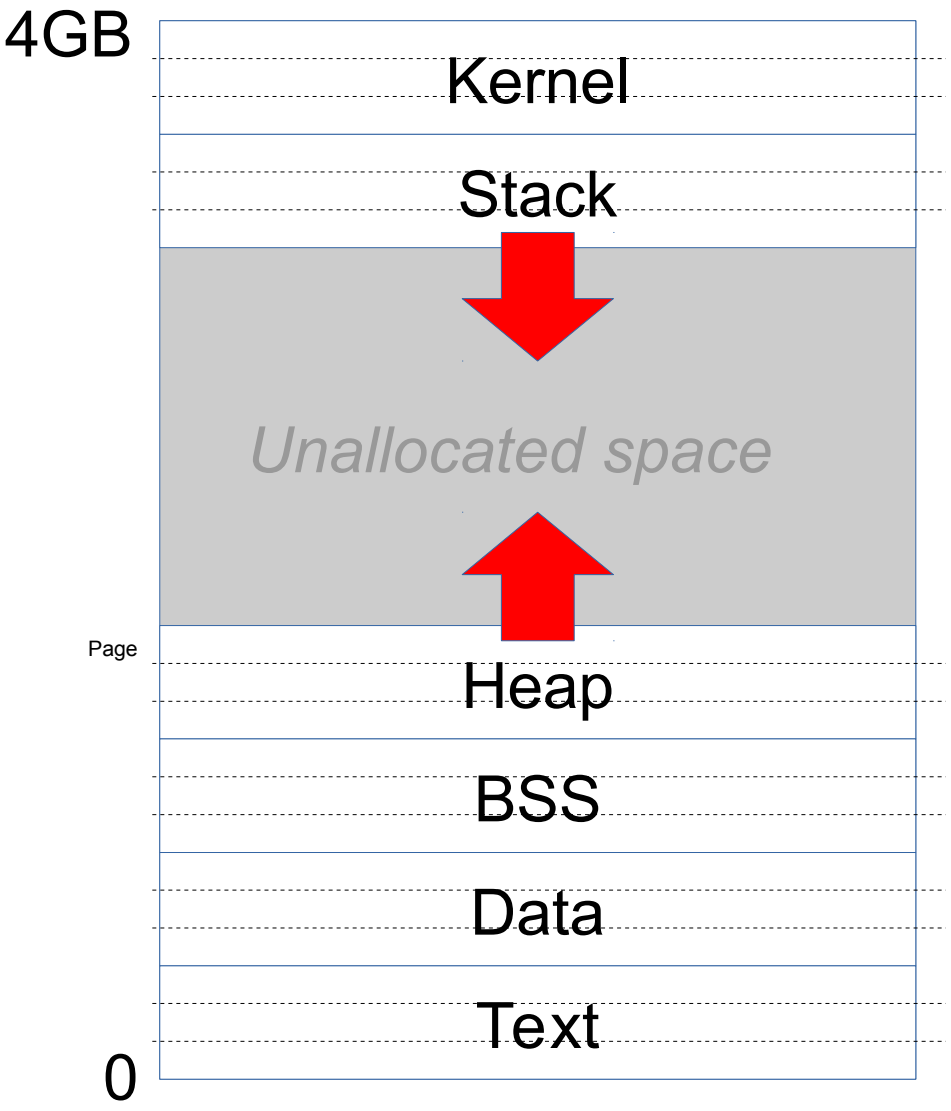


Segmentation and paging on Intel x86



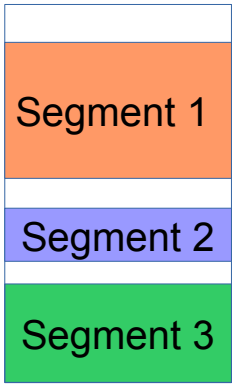
"Simulating" segments

- Using different page protection levels it is possible to define different areas of memory simulating segments
  - The size of those areas is rounded to the page size
  - Heap and stack can easily grow by allocating new pages (i.e. defining new valid entries in the page table)

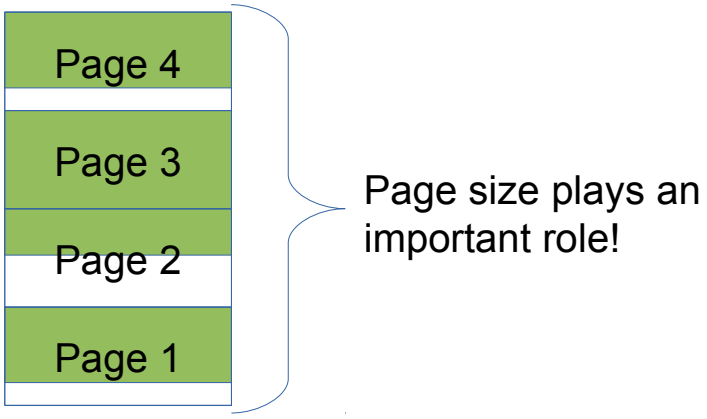


Fragmentation: segmentation vs paging

- Since segments have variable length
  - They can lead to **external fragmentation**



- Since pages have fixed length
  - They can lead to **internal fragmentation**



Segmentation and paging on the same system

- On AMD64 **segmentation is not available anymore** when running in 64 bit mode (long mode)
  - We can “simulate” segments by mapping ranges of pages with different protection levels and privileges
- Paging allows to achieve almost everything that was provided by segmentation
  - The last piece missing in paging in pre-AMD64 compared to segmentation was fine-grained protection (Protection ring vs User/ Supervisor Bit)
  - Paging was thus extended to support additional protection mechanisms at page level (for example, No Execute a.k.a **NX bit**)

An really big concern on today's systems

- **Security**
  - Processes can suffer from memory attacks (buffer overflows) which can affect the security of the whole system
    - An attacker can force a program to run some code on its behalf
- The operating system improve security by:
  - Employing **hardware-assisted mechanisms** to **protect** memory (ex. Page protection, NX bit) that was never intended to be executable (for example, the stack)
  - Implementing additional memory management techniques (such as Address Space Layout Randomization, **ASLR**) to prevent an attacker from reliably jumping to a particular exploited function in memory:
    - The position of memory areas (code, stack, heap, libraries) in the process' address space can be randomized

Memory management in RTOS

- On real-time operating systems we need to minimize the overhead caused by memory management
- Efficient memory management is also critical on RTOS since they are typically used on memory constrained platforms
- Key point: allocation speed → avoid fragmentation

Memory fragmentation

- Fragmentation happens when a task frees some memory
- A garbage collection mechanism or memory compaction is not (always) possible:
  - Requires CPU resources and time
  - It's not deterministic (we might not know when the GC will run, because it depends on the scheduling order)

Avoiding fragmentation

- To avoid fragmentation we can preallocate memory in blocks (chunks) of different fixed lengths
- For each length, we keep track of free and used chunks using linked lists
  - When we need to allocate memory we look for a suitable chunk in the free list
  - When we deallocate a chunk we put it back in the free list
- Allocation time is (practically) constant