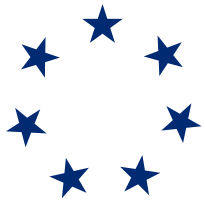
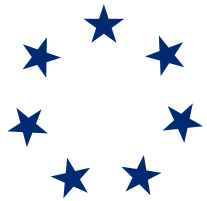
Memory: Paging System



Instructor: Hengming Zou, Ph.D.

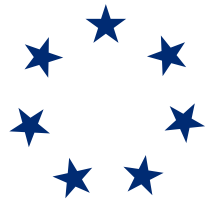
In Pursuit of Absolute Simplicity 求于至简，归于永恒

Content



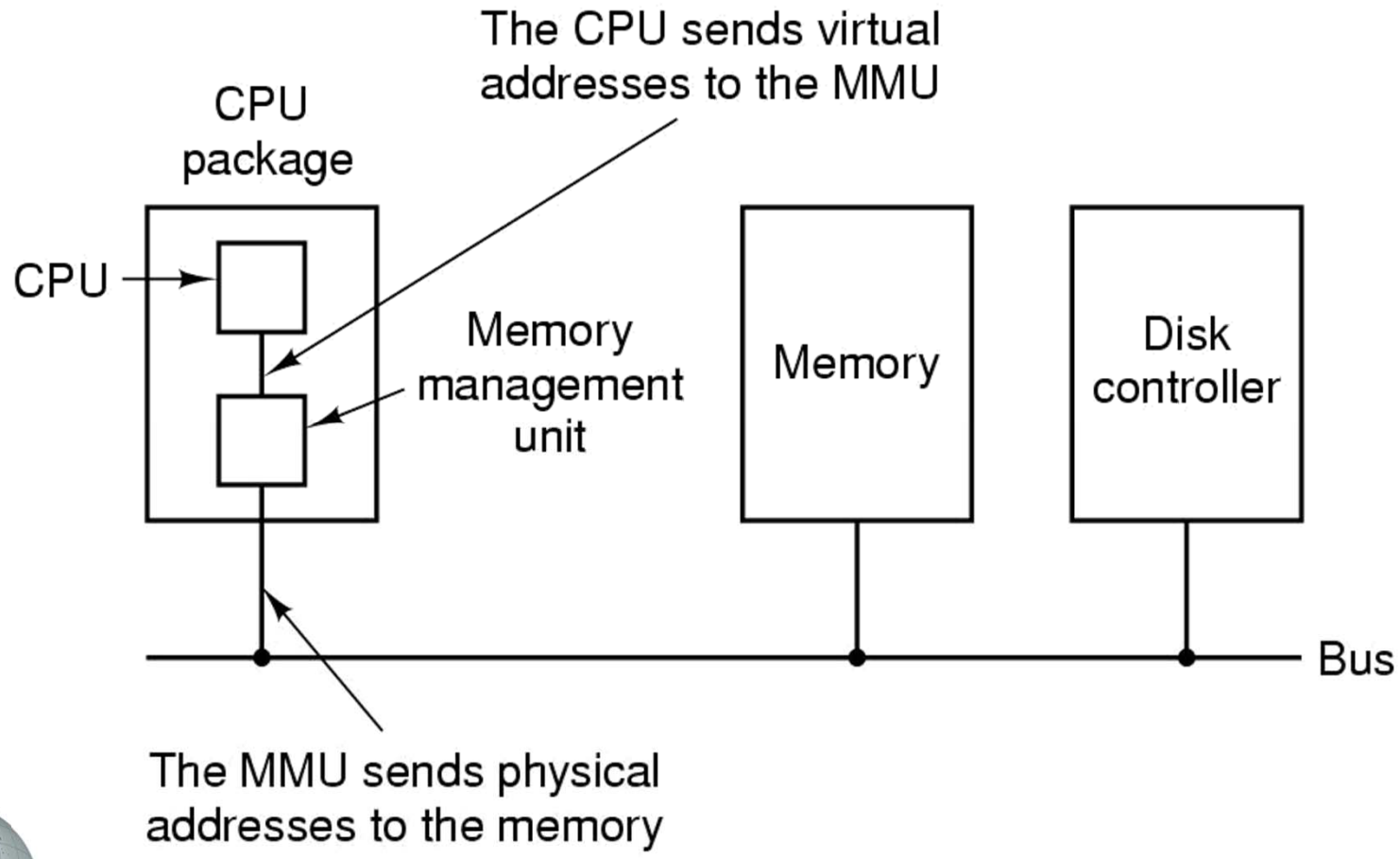
- Paging
- Page table
- Multi-level translation
- Page replacement

Paging

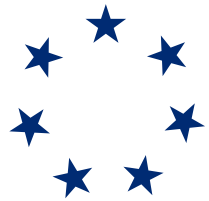


- Allocate physical memory in terms of fixed-size chunks (pages)
  - fixed unit makes it easier to allocate
  - any free physical page can store any virtual page
- Divide virtual memory into same-sized pages
  - Each virtual page can be in physical memory or paged out to disk
- Processes access memory by virtual addresses
  - Each VM reference is translated into PM reference by the MMU
- Virtual address
  - virtual page # (high bits of address, e.g. bits 31-12)
  - offset (low bits of addr, e.g. bits 11-0 for 4 KB page)

Paging


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

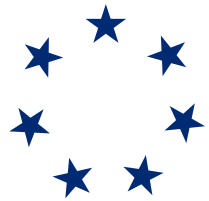
Paging



- Page translation process:
  - if (virtual page is invalid or non-resident or protected) {
  - trap to OS fault handler
  - } else {
  - physical page # = pageTable[virtual page #].physPageNum
  - }

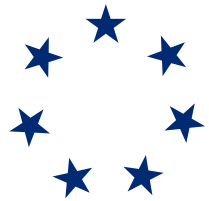
- What must be changed on a context switch?
  - Page table, registers, cache image
  - Possibly memory images

Paging



- How does processor know that virtual page is not in physical memory?
  - Through a valid/invalid bit in page table
- Pages can have different protections
  - e.g. read, write, execute
  - These information is also kept in page table

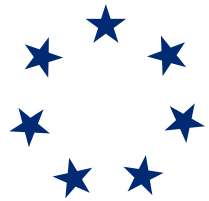
Valid vs. Resident



- Resident means a virtual page is in memory
  - NOT an error for a program to access non-resident pages
- Valid means a virtual page is currently legal for the program to access

- Who makes a virtual page resident/non-resident?
- Who makes a virtual page valid/invalid?
- Why would a process want one of its virtual pages to be invalid?

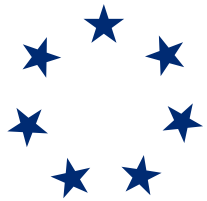
Page Table



- Key component in a paging system
- A hardware data structure
- Used to keep track of virtual-physical page map
  - One entry for each virtual page
  - Also keep information concerning other relevant information

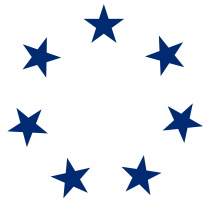
    →such as read, write, execute, valid, etc.
- MMU uses it to perform addresses translation

Page Table Contents

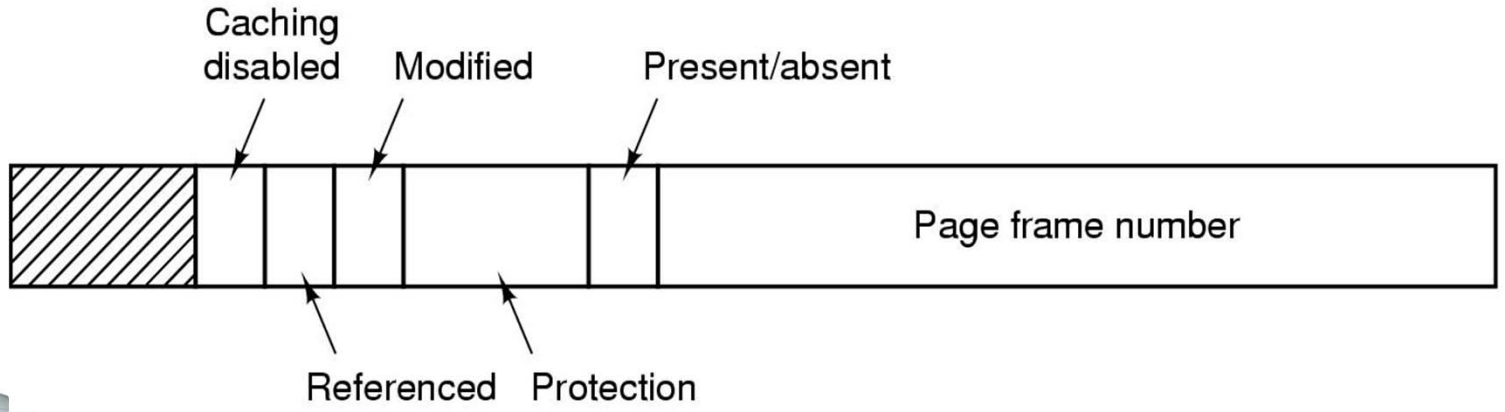


- Resident bit:
  - true if the virtual page is in physical memory
- Physical page # (if in physical memory)
- Dirty bit: set by MMU when page is written
- Reference bit: set by MMU when page is read or written
- Protection bits (readable, writable)
  - set by operating system to control access to page
  - Checked by hardware on each access

Page Table Contents



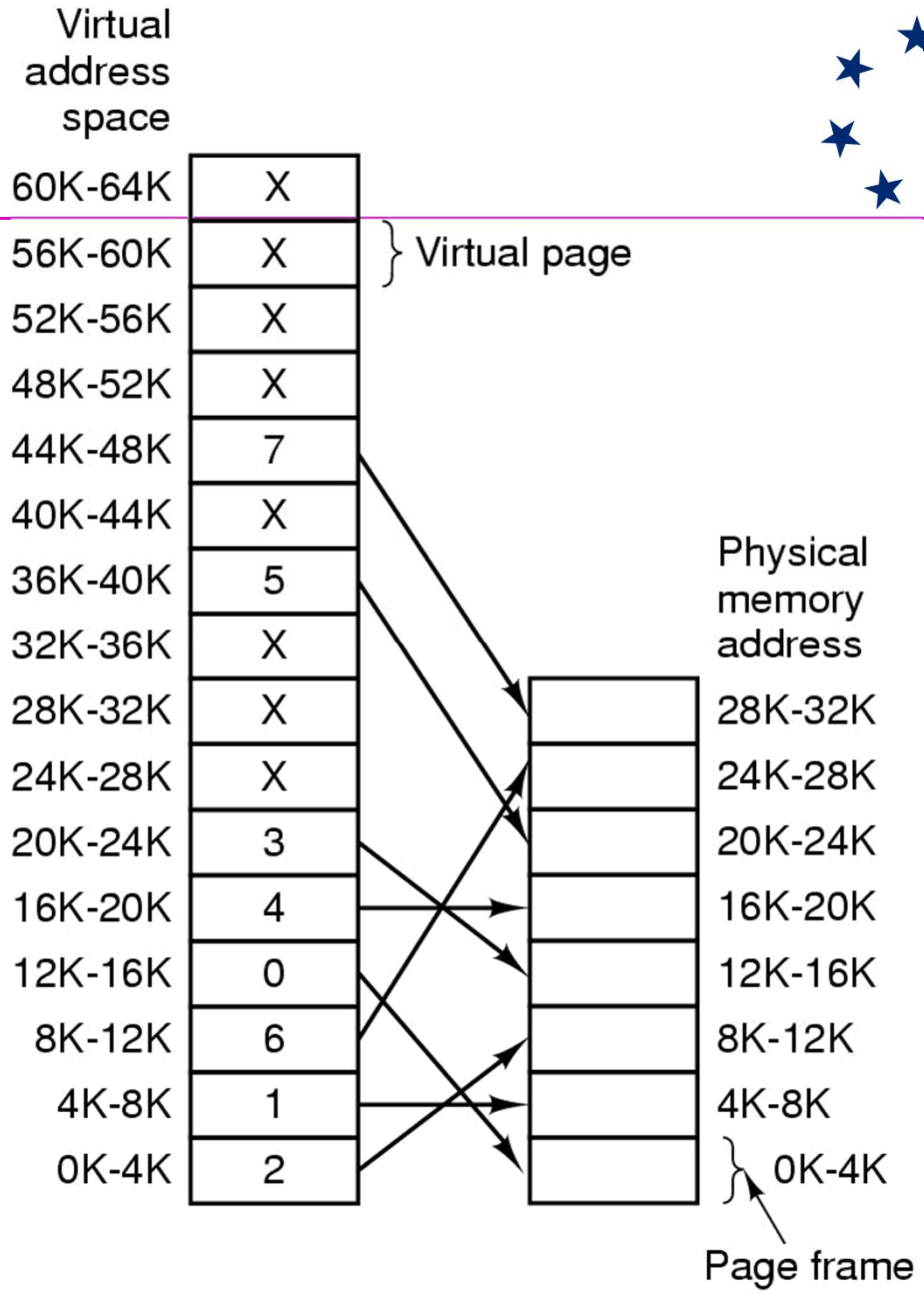
- Does the hardware page table need to store the disk block # for non-resident virtual pages?
- Really need hardware to maintain a “dirty” bit?
- How to reduce # of faults required to do this?
- Do we really need hardware to maintain a “reference” bit?



Typical page table entry

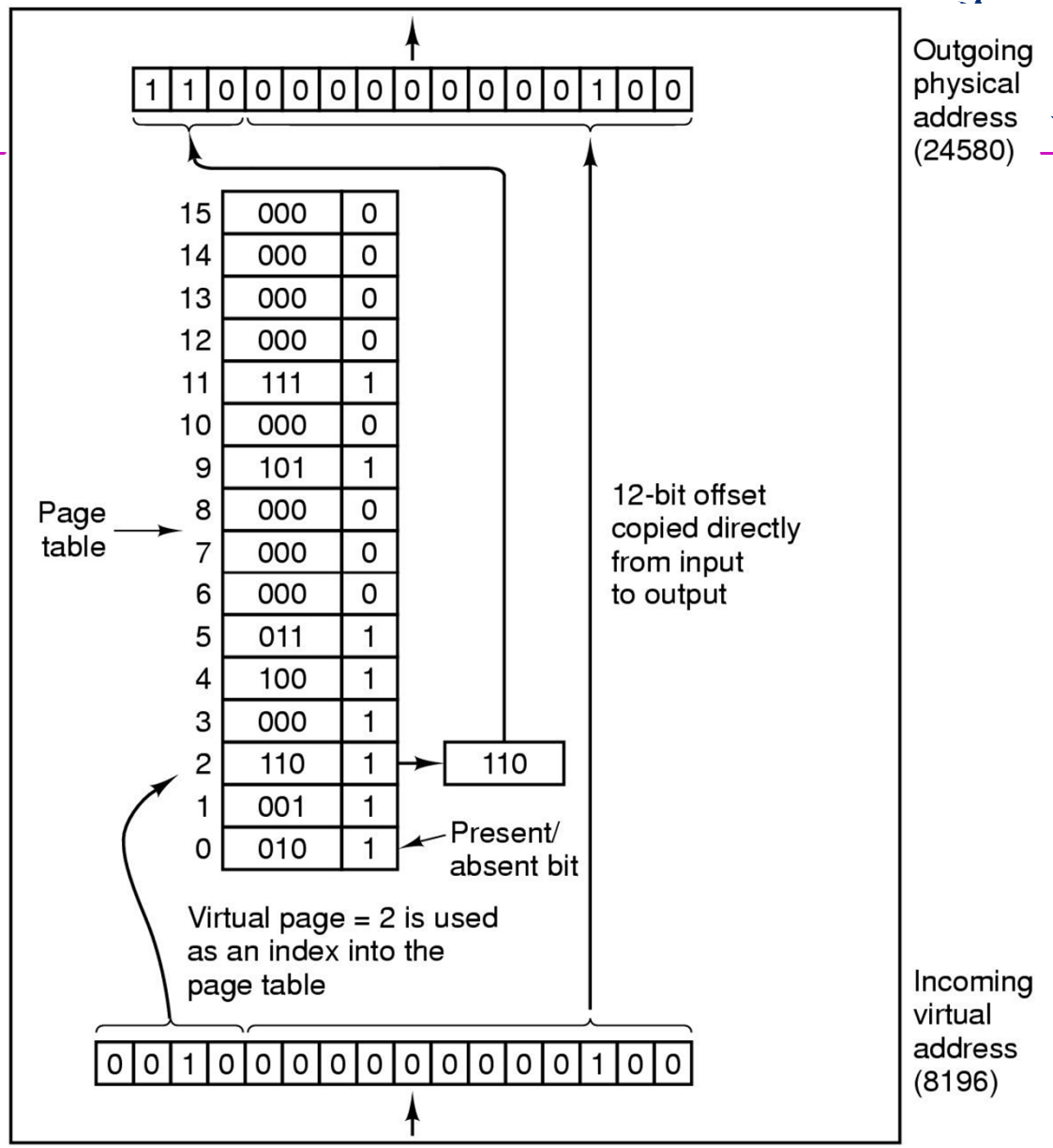
Paging Table Example

- The relation between
  - virtual addresses and
  - physical memory addresses
    - →given by page table

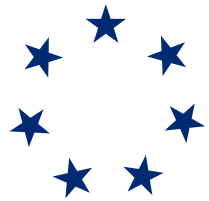


Paging

- Internal operation of
  - MMU with 16
  - 4 KB pages

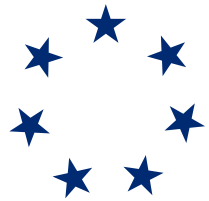


Paging Pros and Cons



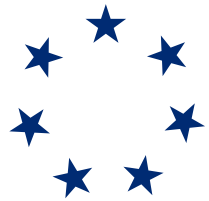
- + simple memory allocation
- + can share lots of small pieces of an address space
- + easy to grow the address space
  - Simply add a virtual page to the page table
  - and find a free physical page to hold the virtual page before accessing it

Paging Pros and Cons



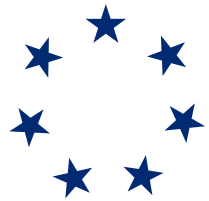
- Problems with paging?
- The size of page table could be enormous
- Take 32 bits virtual address for example
- Assume the size of page is 4KB
- Then there are 65536 virtual pages
- For a 64 bit virtual address?

Paging Pros and Cons



- The solution?
- Use multi-level translation!
- Break page tables into 2 or more levels
- Top-level page table always reside in memory
- Second-level page tables in memory as needed

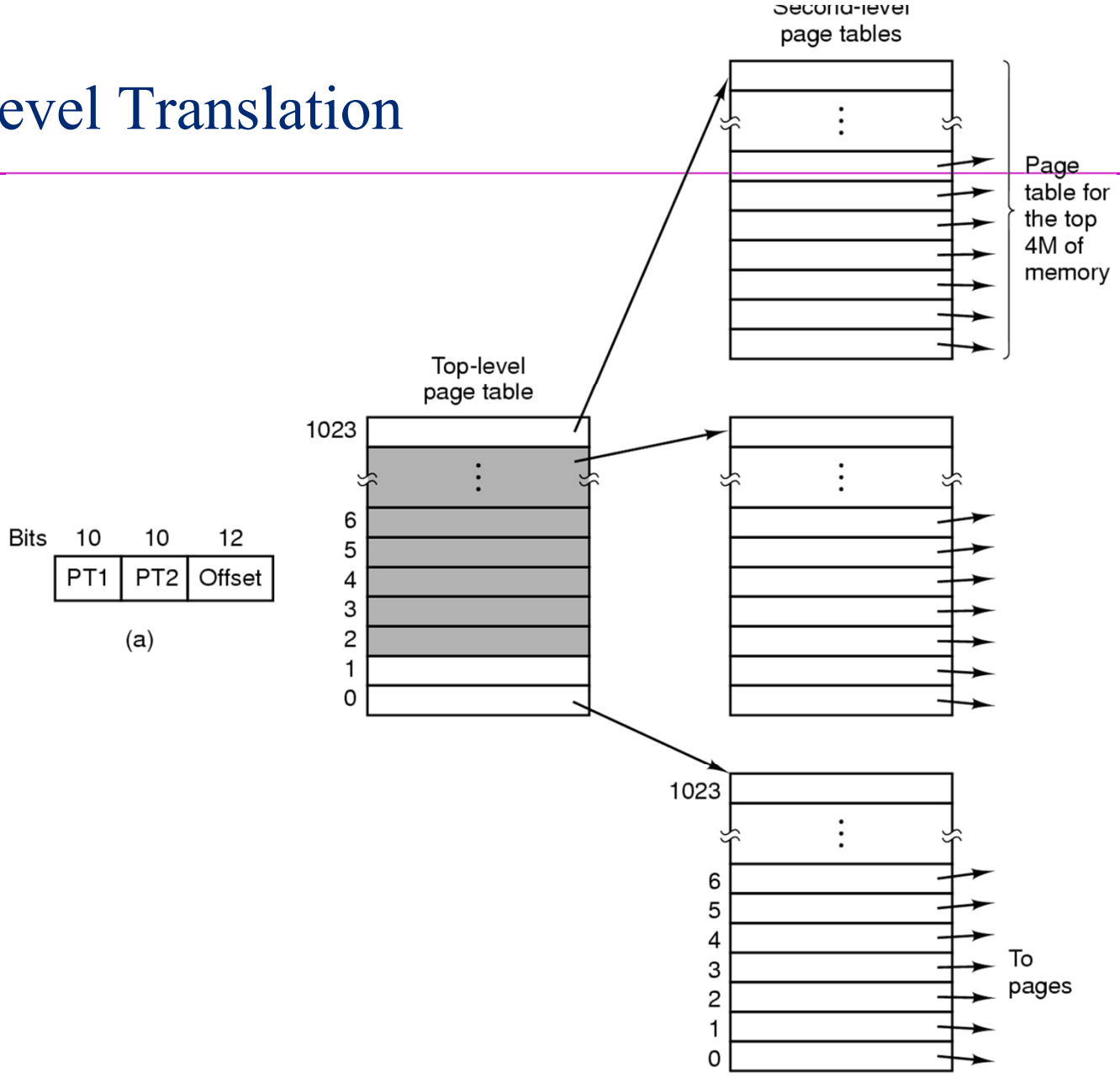
Multi-level Translation



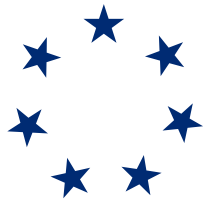
- Standard page table is a simple array
  - one degree of indirection
- Multi-level translation changes this into a tree
  - multiple degrees of indirection

- Example: two-level page table
  - Index into the level 1 page table using virtual address bits 31-22
  - Index into the level 2 page table using virtual address bits 21-12
  - Page offset: bits 11-0 (4 KB page)

Multi-level Translation



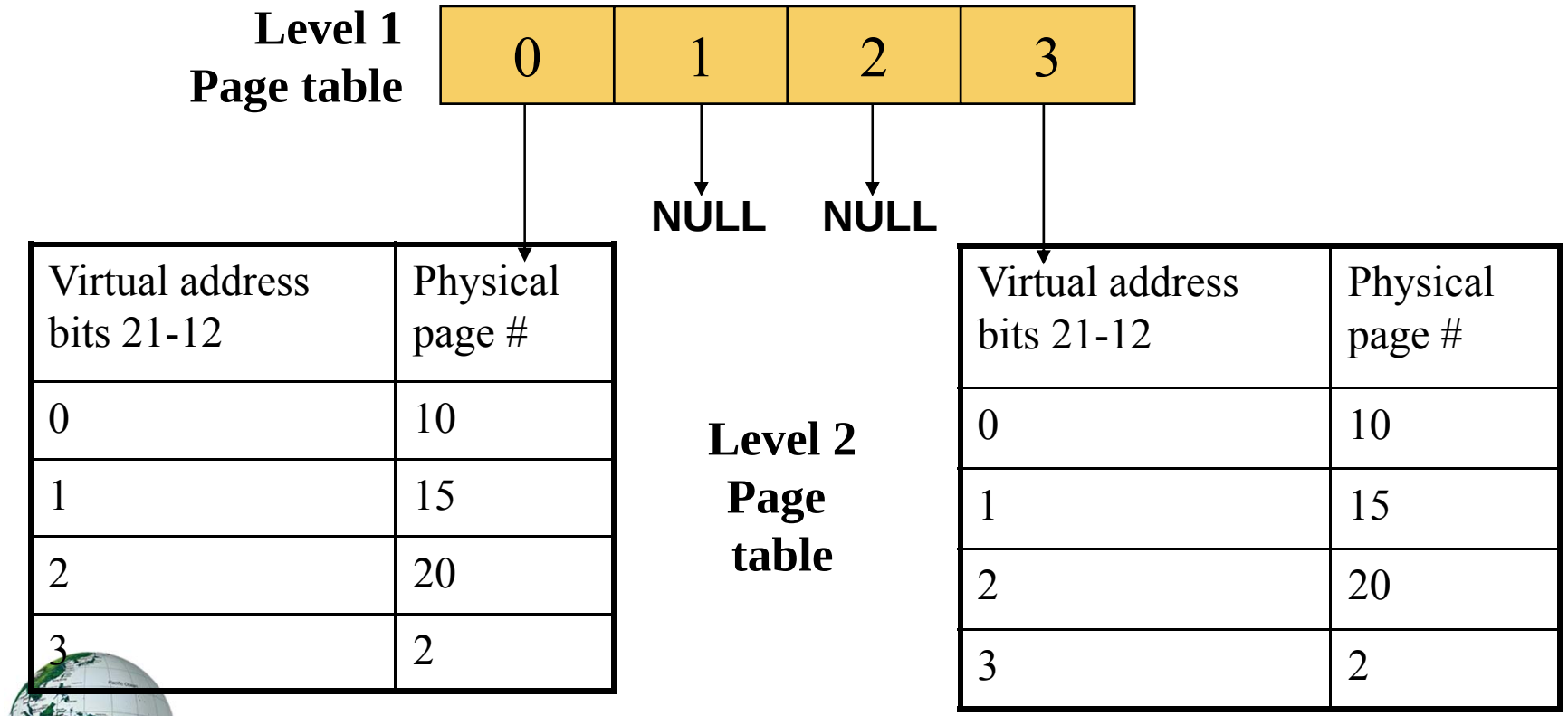
Multi-level Translation



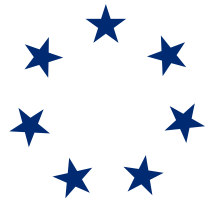
- What info is stored in the level 1 page table?
  - Information concerning secondary-level page tables
- What info is stored in the level 2 page table?
  - Virtual-to-physical page mappings

Multi-level Translation

- This is a two-level tree



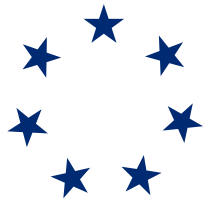
Multi-level Translation



- How does this allow the translation data to take less space?
- How to use share memory when using multi-level page tables?
- What must be changed on a context switch?

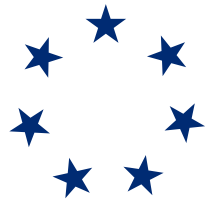
- Another alternative:
  - use segments in place of the level-1 page table
  - This uses pages on level 2 (i.e. break each segment

Multi-level Translation



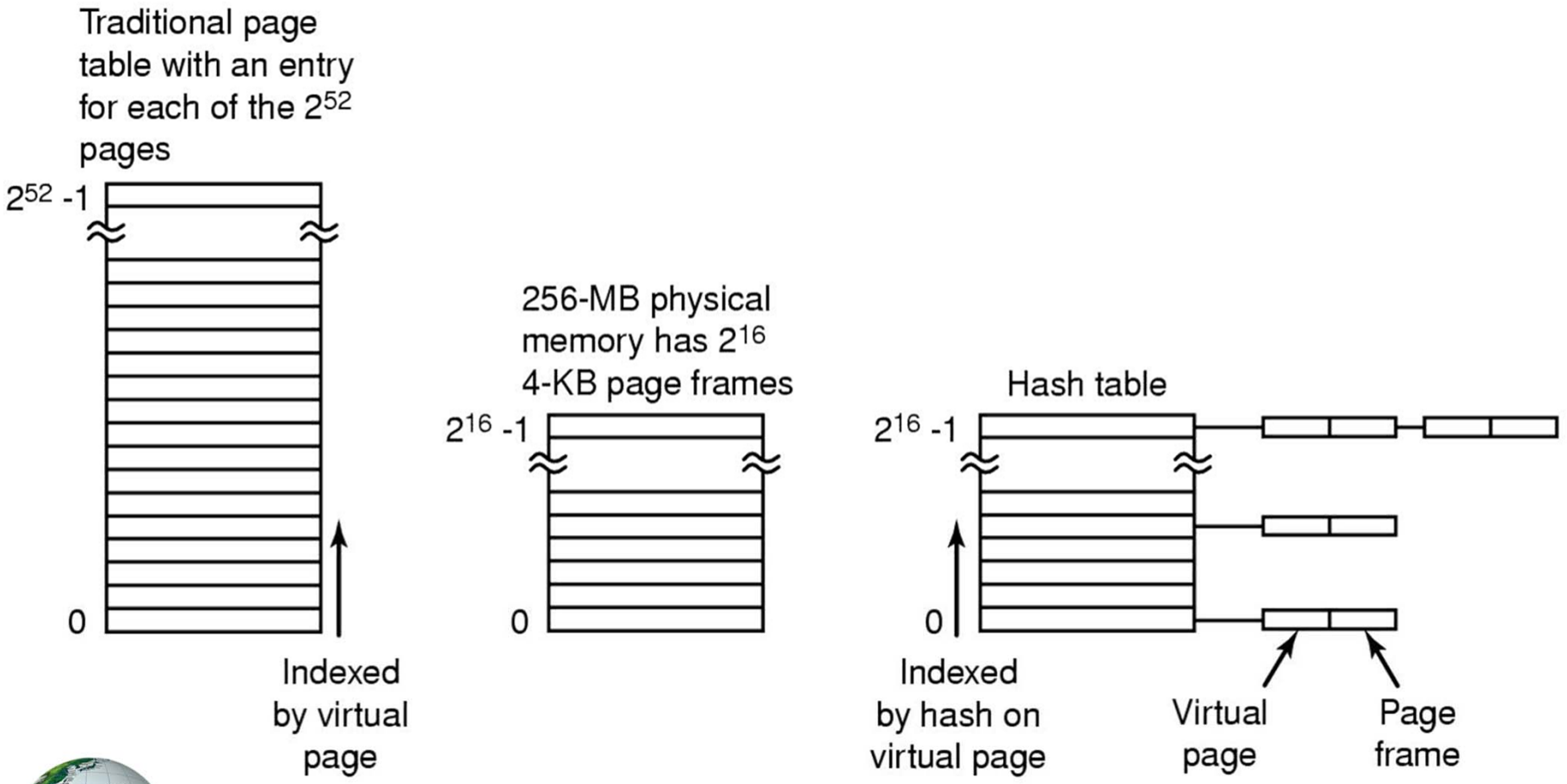
- Pros and cons
- + space-efficient for sparse address spaces
- + easy memory allocation
- + lots of ways to share memory
- - two extra lookups per memory reference

Inverted Page Table

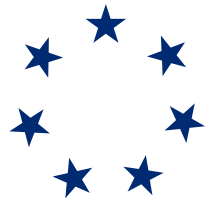


- An alternate solution to big table size problem
- Rather than storing virtual-physical mapping
- We store physical-virtual mapping
- This significantly reduce the page table size

Inverted Page Tables

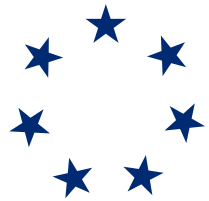


Comparing Basic Translation Schemes



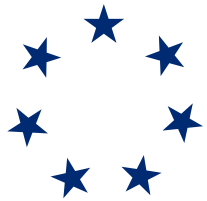
- Base and bound:
  - unit (and swapping) is an entire address space
- Segments: unit (and swapping) is a segment
  - a few large, variable-sized segments per address space
- Page: unit (and swapping/paging) is a page
  - lots of small, fixed-sized pages per address space
  - How to modify paging to take less space?

Translation Speed



- Paging involves 1 or more additional memory references
  - This can be a big issue if not taking care of
- How to speed up the translation process?
- Solution:
  - Translation look-aside buffer

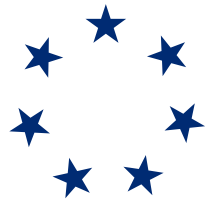
Translation Look-aside Buffer



- Facility to speed up memory access
- Abbreviated as TLB
- Caches translation from virtual page # to physical page #
- TLB conceptually caches the entire page table entry
  - e.g. dirty bit, reference bit, protection

- If TLB contains the entry you're looking for
  - can skip all the translation steps above
- On TLB miss, figure out the translation by
  - getting the user's page table entry,
  - store in the TLB, then restart the instruction

A TLB to Speed Up Paging

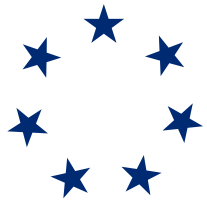


Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

- Does this change what happens on a context switch?

Page Replacement

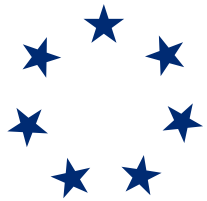
Replacement



- One design dimension in virtual memory is
  - which page to replace when you need a free page?
- Goal is to reduce the number of page faults
  - i.e. a page to be accessed is not in memory

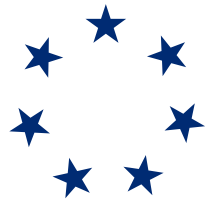
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Replacement Algorithms



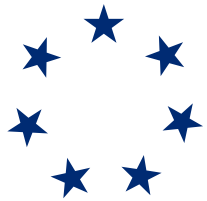
- Random replacement
- Optimal replacement
- NRU (not recently used) replacement
- FIFO (first in first out) replacement
- Second chance replacement
- LRU (least recently used) replacement
- Clock replacement
- Work set replacement
- Work set clock replacement

Random and Optimal Replacement



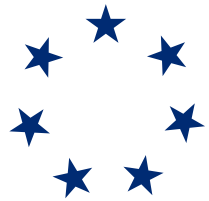
- Random replacement:
  - Randomly pick a page to replace
  - Easy to implement, but poor results
- Optimal Replacement:
  - Replace page needed at farthest point in future
    - →i.e. page that won't be used for the longest time
    - →this yields the minimum number of misses
    - →but requires knowledge of the future
  - Forecast future is difficult if at all possible

NRU Replacement



- Replace page not recently used
- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified into four classes:
  - not referenced, not modified
  - not referenced, modified
  - referenced, not modified
  - referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class

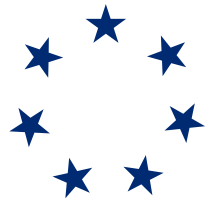
FIFO Replacement



- Replace the page that was brought into memory the longest time ago
- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced

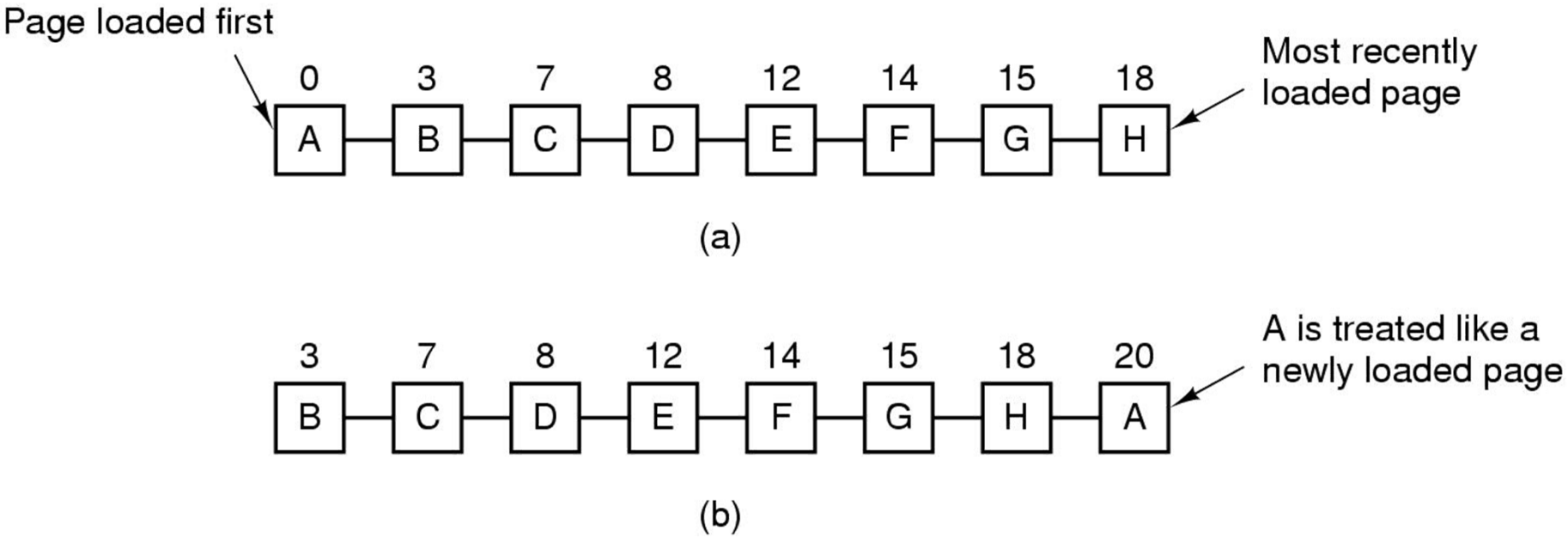
- Unfortunately, this can replace popular pages that are brought into memory a long time ago (and used frequently since then)

Second Chance Algorithm



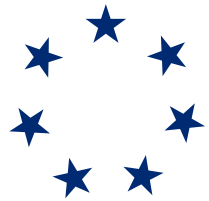
- A modification to FIFO
- Just as FIFO but page evicted only if R bit is 0
- If R bit is 1, the page is put behind the list
  - And the R bit is cleared
- i.e. page brought in the longest time ago but with R bit set is given a second chance

Second Chance Algorithm



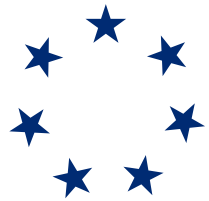
Page list if fault occurs at time 20, A has *R* bit set
(numbers above pages are loading times)

The Clock Algorithm



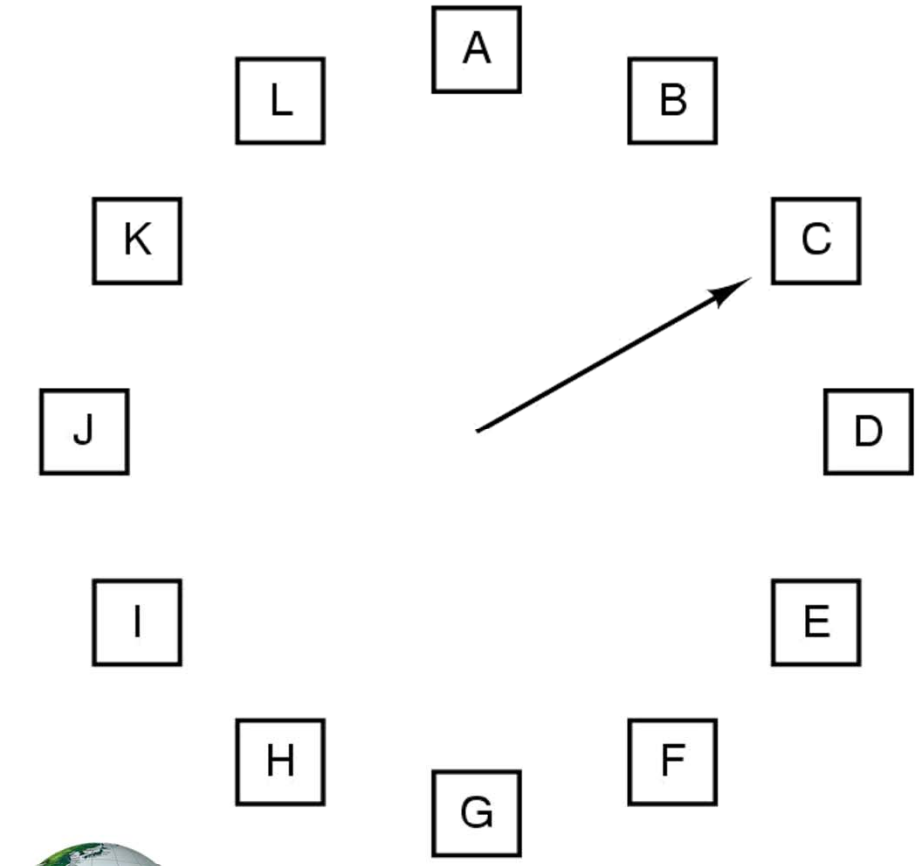
- Maintain “referenced” bit for each resident page
  - set automatically when the page is referenced
- Reference bit can be cleared by OS
- The resident page organized into a clock cycle
- A clock hand points to one of the pages

The Clock Algorithm



- To find a page to evict:
  - look at page being pointed to by clock hand
- reference=0:
  - means page hasn't been accessed in a long time (since last sweep)
- reference=1:
  - means page has been accessed since your last sweep. What to do?

The Clock Algorithm

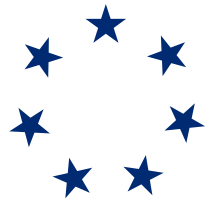


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

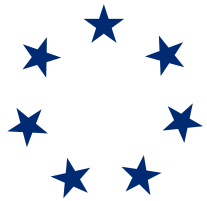
The Clock Algorithm



- Can this infinite loop?
- What if it finds all pages referenced since the last sweep?
- New pages are put behind the clock hand, with reference=1

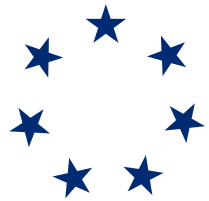
- Why is hardware support needed to maintain reference bit?
- What is the difference between clock and second chance algorithm?
  - How can you identify an “old” page?
  - The standard need not be time

LRU Replacement



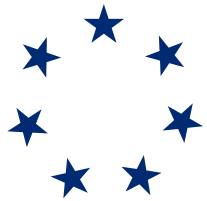
- LRU stands for Least Recently Used
- Use past references to predict the future
  - temporal locality
- If a page hasn't been used for a long time
  - it probably won't be used again for a long time

LRU Replacement



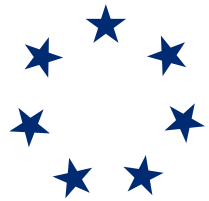
- LRU is an approximation to OPT
- Can we approximate LRU to make it easier to implement without increasing miss rate too much?
- Basic idea is to replace an old page
  - not necessarily the oldest page

LRU Replacement



- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!
  - Can be pure counting or pure timing scheme
  - If purely timing, reduce to a special FIFO
- Alternatively use counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter
  - **Pure counting scheme**

Implementing LRU with Matrix



- Another option is to use n×n matrix
  - Here n is the number of pages in virtual space
- The matrix is set to zero initially
- Whenever a page k is referenced:
  - Row k is to all one, then column k is set to all zero
- Whenever need to pick a page to evict
  - Pick the one with the smallest number (row value)

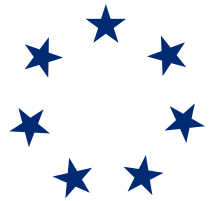
- **Pure timing scheme**

Implementing LRU with Matrix

- Pages referenced in order 0,1,2,3,2,1,0,3,2,3

	Page 0				Page 0				Page 0				Page 0				Page 0			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0
2	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	1	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0
0	0	0	0	0	0	1	1	1	0	1	1	0	0	1	0	0	0	1	0	0
1	1	0	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	1	0	0	0	0	1	1	0	1	1	1	0	0
3	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	0

Implementing LRU with Aging



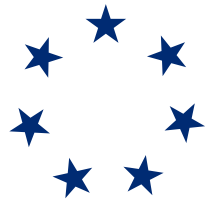
- Each page corresponding to a shift register
  - Shift register is initially set to zero
- At every clock tick:
  - the value of the shift is shifted one bit right, and
  - the R bit is added to the left most bit of corresponding shifter
- Whenever need to pick a page to evict
  - Pick the one with the smallest number

- **A combined timing and counting scheme**

Implementing LRU with Aging

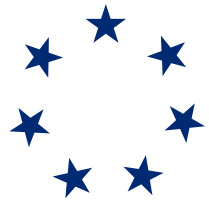
Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

The Working Set Algorithm



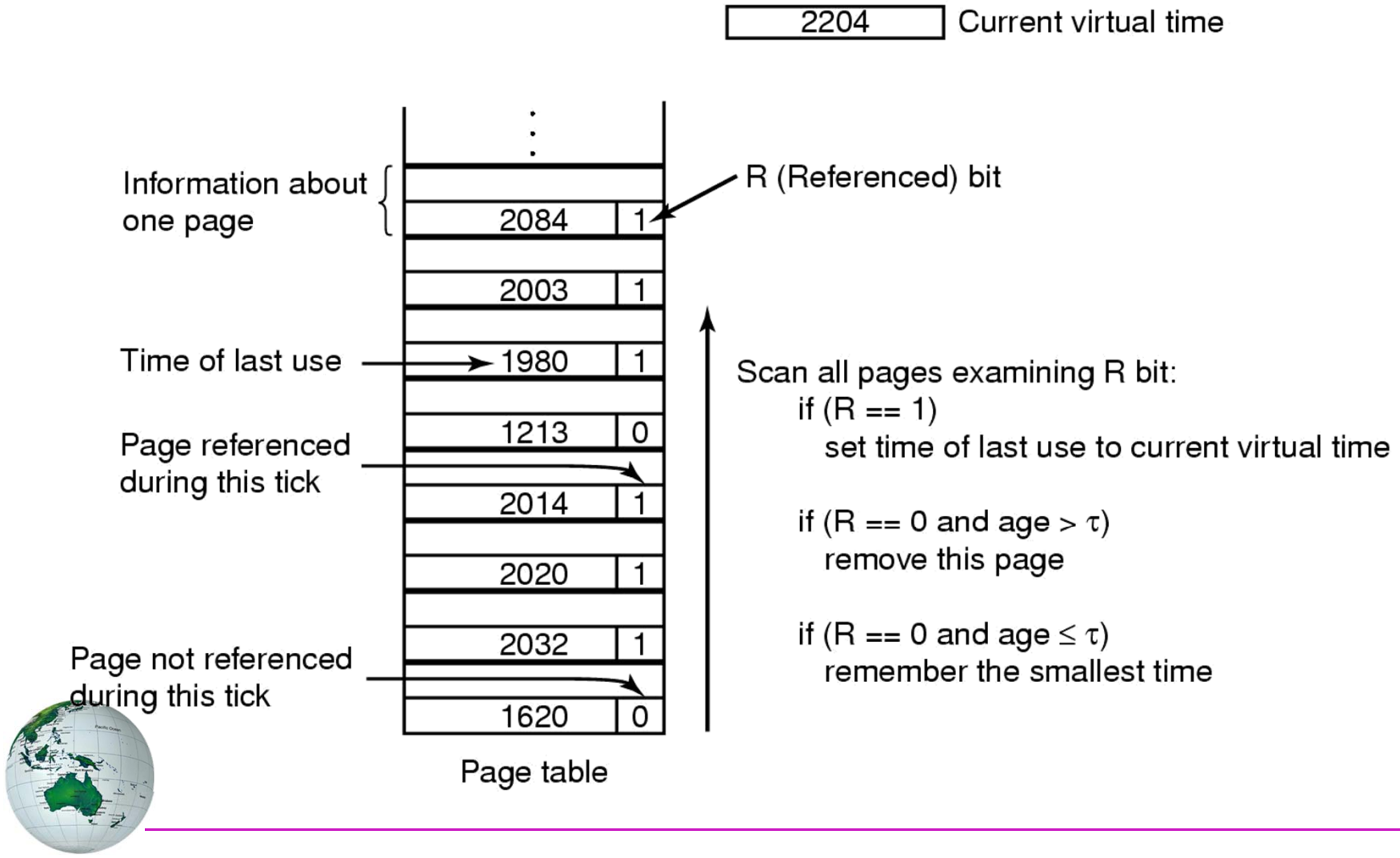
- The working set is
  - the set of pages used by the *k* most recent memory references
  - all pages used in last T seconds or T instructions
- *w*(*k*,*t*) is the size of the working set at time, *t*
- larger working set ==>
  - process needs more physical memory to run well
  - i.e. avoid thrashing

The Working Set Algorithm

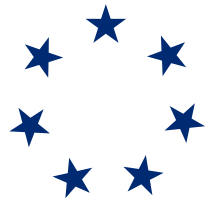


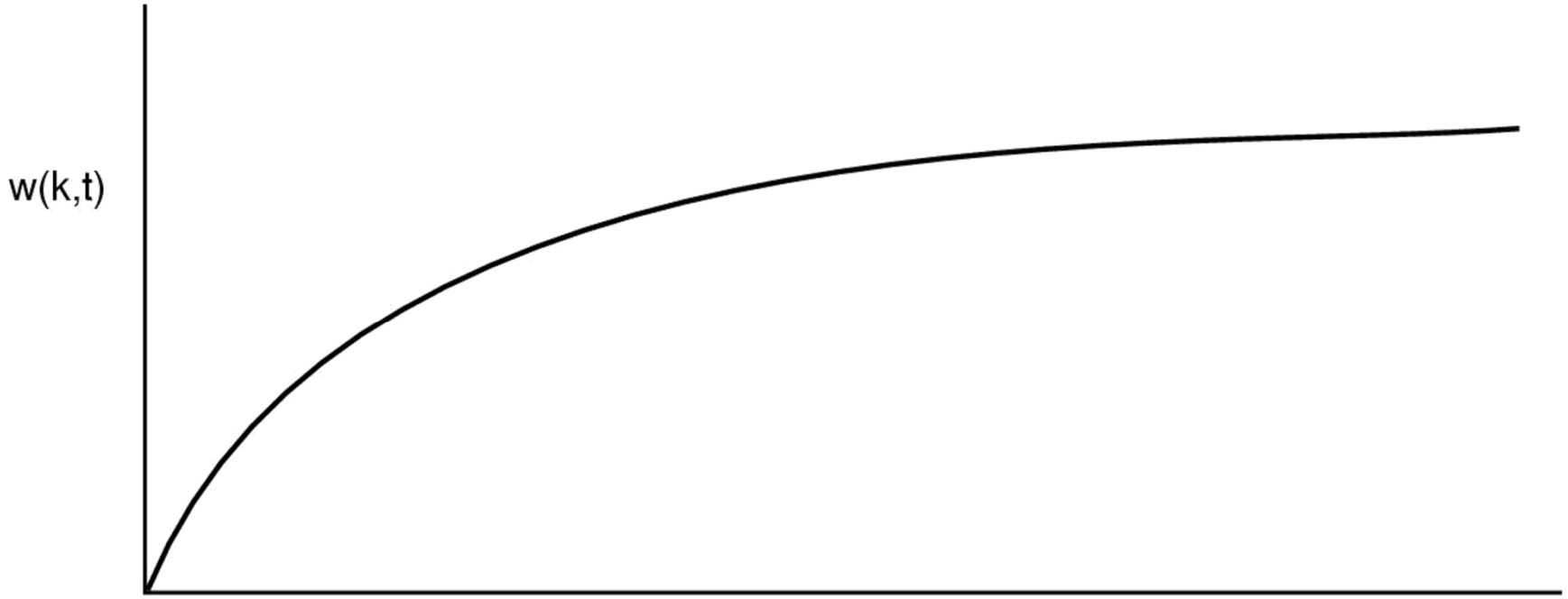
- Sum of all working sets should fit in memory
  - otherwise system will thrash
- Only run a set of processes whose working sets all fit in memory
  - this is called a “balance set”
- How to measure size of working set for a process?

The Working Set Algorithm



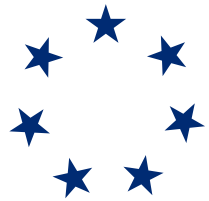
The Working Set Algorithm





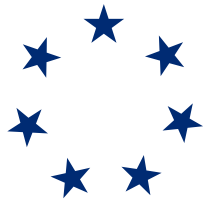
Work set changes as time passes but stabilizes after

The Work Set Clock Algorithm



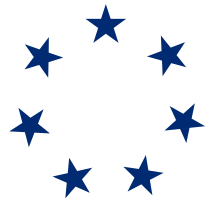
- Combine work set algorithm with clock algorithm
- Pages organized into a clock cycle
- Each page has a time of last use and R bit
- Whenever needs to evict a page:
  - Inspect from the page pointed by the clock hand
  - The 1st page that with 0 R bit & is outside work set is evicted

Page Replacement Algorithm Review



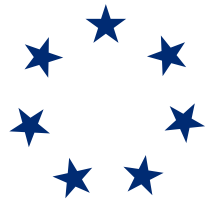
Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not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

Drawback of Paging



- In a paging system, each process occupies one virtual address space
- This may be inconvenient because
  - different sections of the process can grow or shrink independently

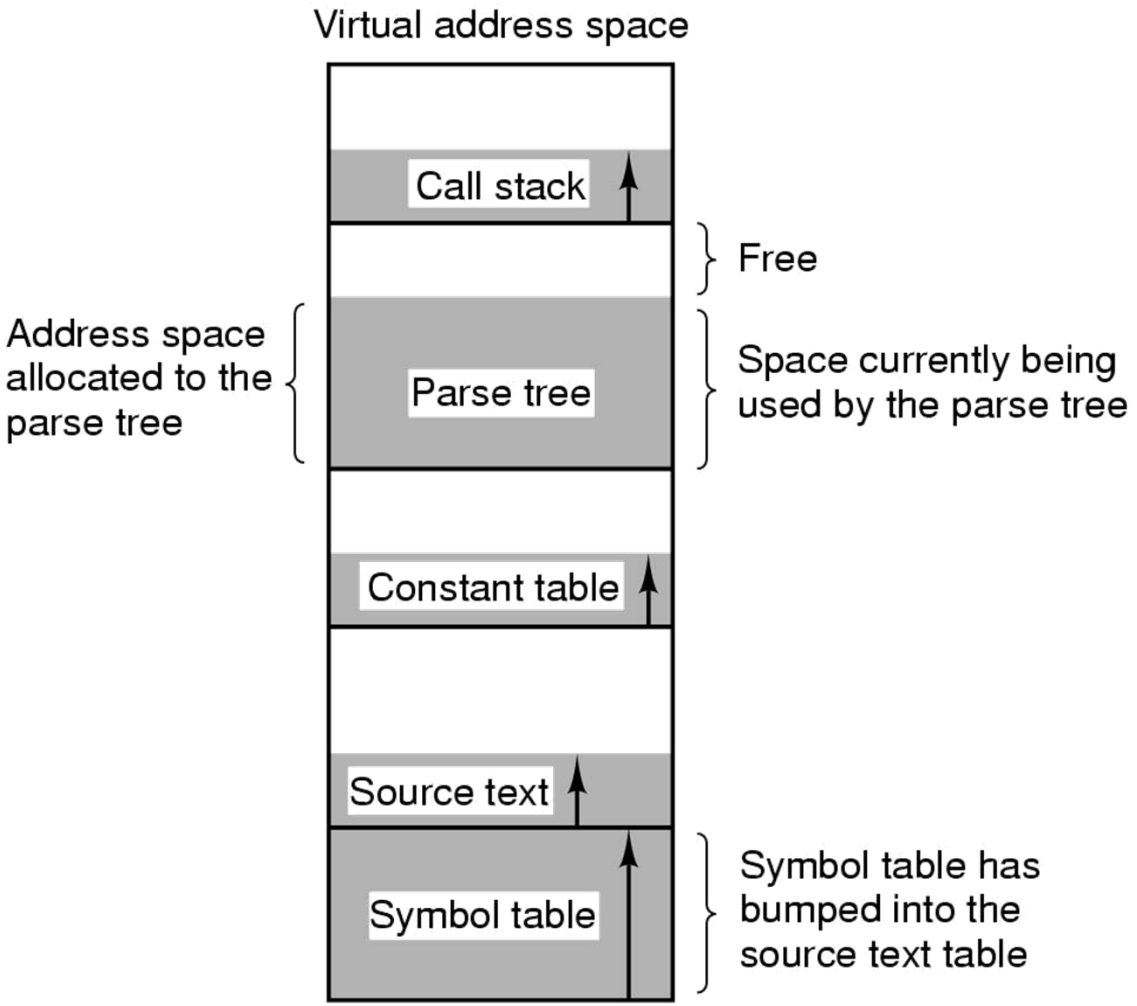
Drawback of Paging

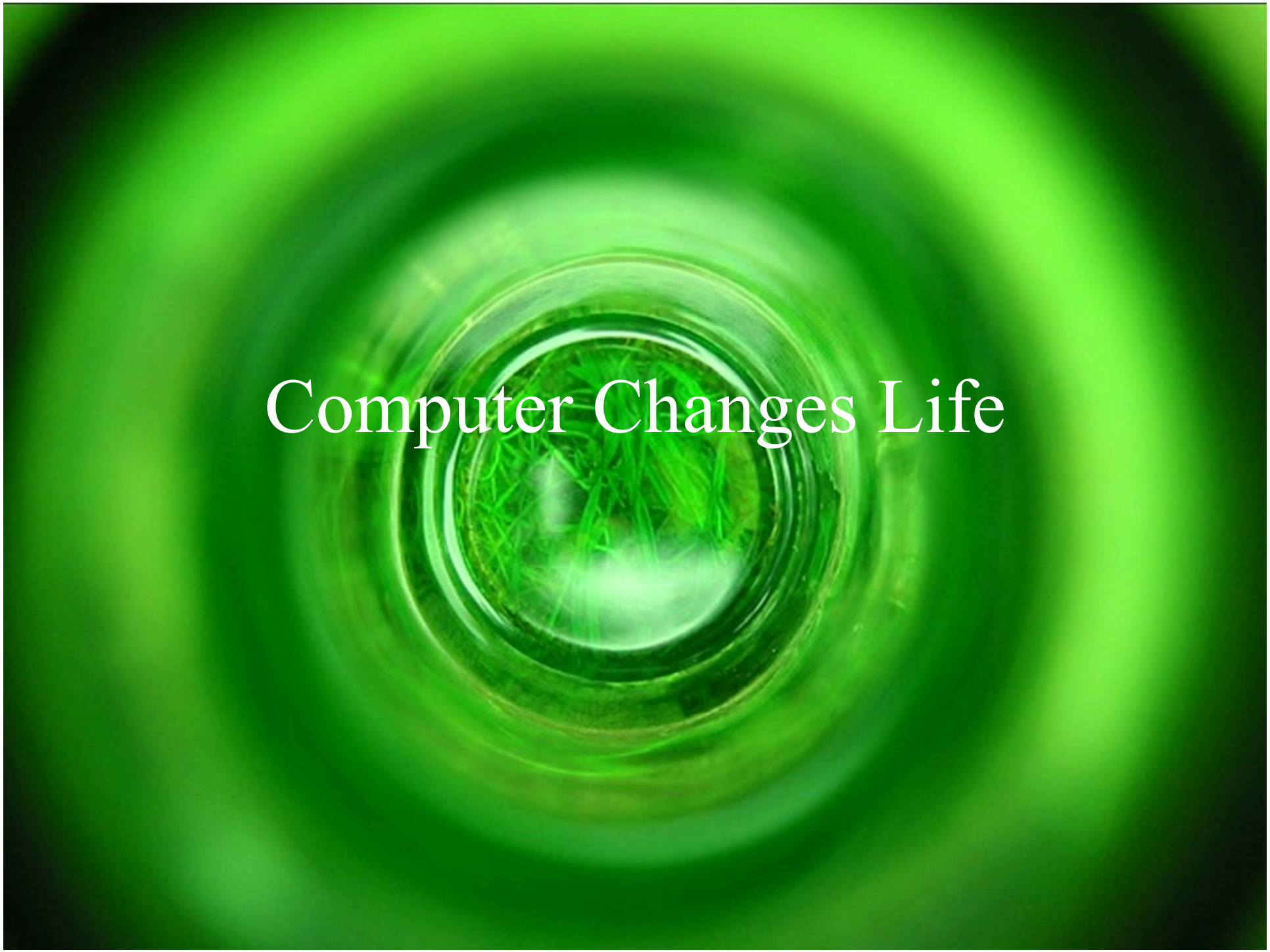


- One-dimensional address space with growing tables
- One table may bump into another

The Solution:

SEGMENTATION!




Computer Changes Life