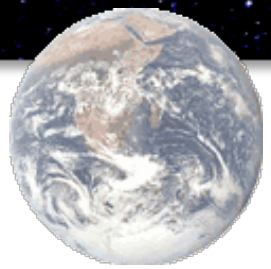
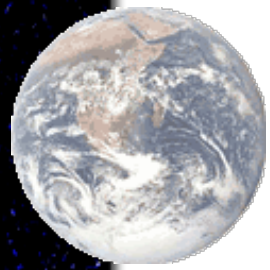


Virtual Memory



Virtual Memory

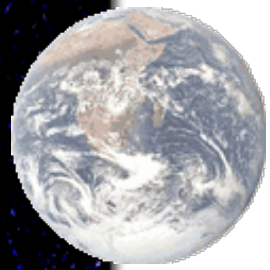
Motivation

Combined size of the program, data, and stack exceeds the amount of physical memory available.

Code needs to be in memory to execute, but entire program rarely used; entire program code not needed at same time.

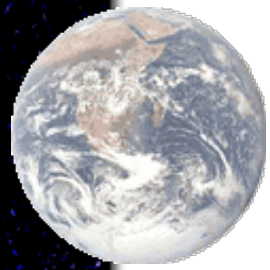
Virtual memory:

- Technique used by OS to keep only portions of program currently needed for execution in the memory and keep the rest on disk
- A storage location scheme in which secondary memory can be addressed as though it were part of main memory.



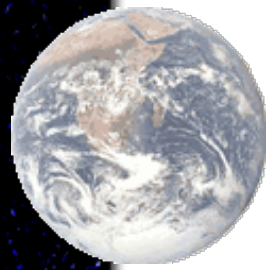
Virtual Memory

- Program-generated addresses are translated automatically to the corresponding physical addresses.
- The size of virtual storage is limited by the addressing scheme and by the amount of secondary memory, not by the size of main memory.
  - Logical address space can therefore be much larger than physical address space
- Each program takes less memory while running
  - More programs running concurrently
  - Increased CPU utilization and throughput with no increase
  - Less I/O needed to load or swap processes



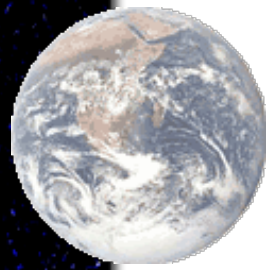
Principle of Locality

- Program and data references within a process tend to cluster.
- Only a few pieces of a process will be needed over a short period of time.
- Possible to make intelligent guesses about which pieces will be needed in the future.
- This suggests that virtual memory may work efficiently.



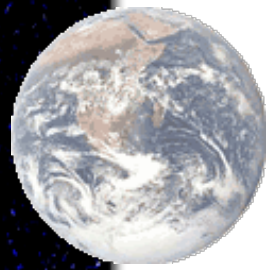
Execution of A Program

- All pieces of a process do not need to be loaded in main memory during execution.
- Operating system brings into main memory a few pieces of the program.
- Resident set - portion of process that is in main memory.
- An interrupt is generated when an address is needed that is not in main memory.
- Operating system places the process in a blocking state.



Execution of A Program

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state



Key Terms

- Virtual address/Logical address

  The address assigned to a location in virtual memory to allow location to be accessed as though it were part of main memory.

- Virtual address space

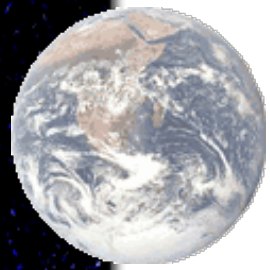
  The virtual storage assigned to a process

- Address space

  The range of memory addresses available to a process

- Physical address/Real address

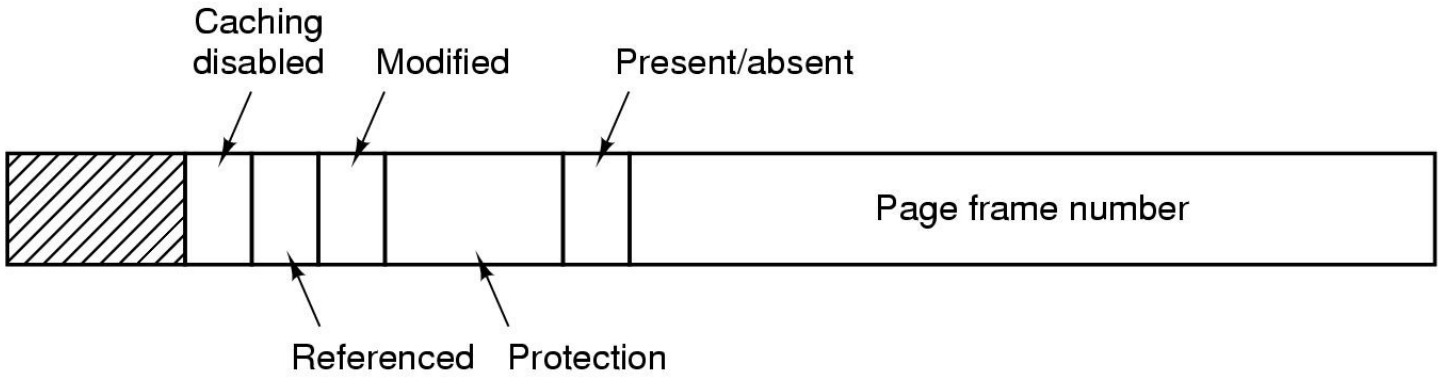
  The address of storage location in main memory

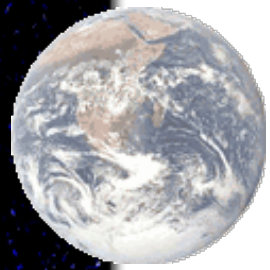


Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- Present-absent (Valid-invalid) bit attached to each entry in the page table:
  - "present" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "absent" indicates that the page is not in the process' logical address space
- Use page-table length register (PTLR)
- Any violations result in a trap to the kernel

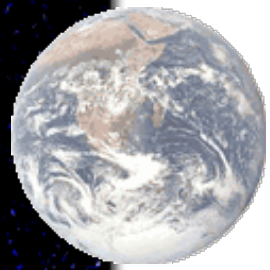
Page Table Entry





Structure of the Page Table

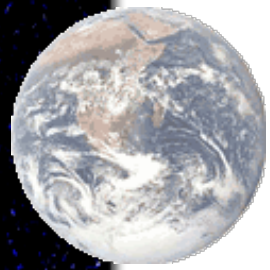
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space
    - Logical address space is of …… in bytes.
  - Page size of 4 KB = …… in bytes.
  - Number of entries in the page table is ……
  - If each page table entry is 4 bytes
    - Page table size is ……
- That amount of memory used to cost a lot.
- Don't want to allocate that contiguously in main memory



Virtual Memory

Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

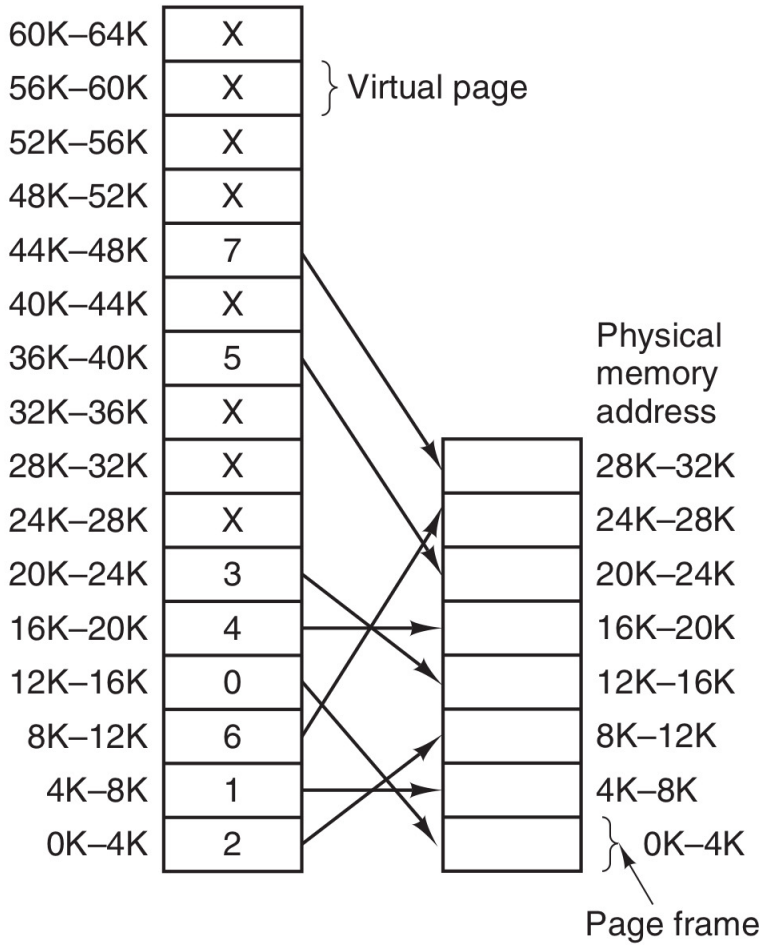


Demand Paging

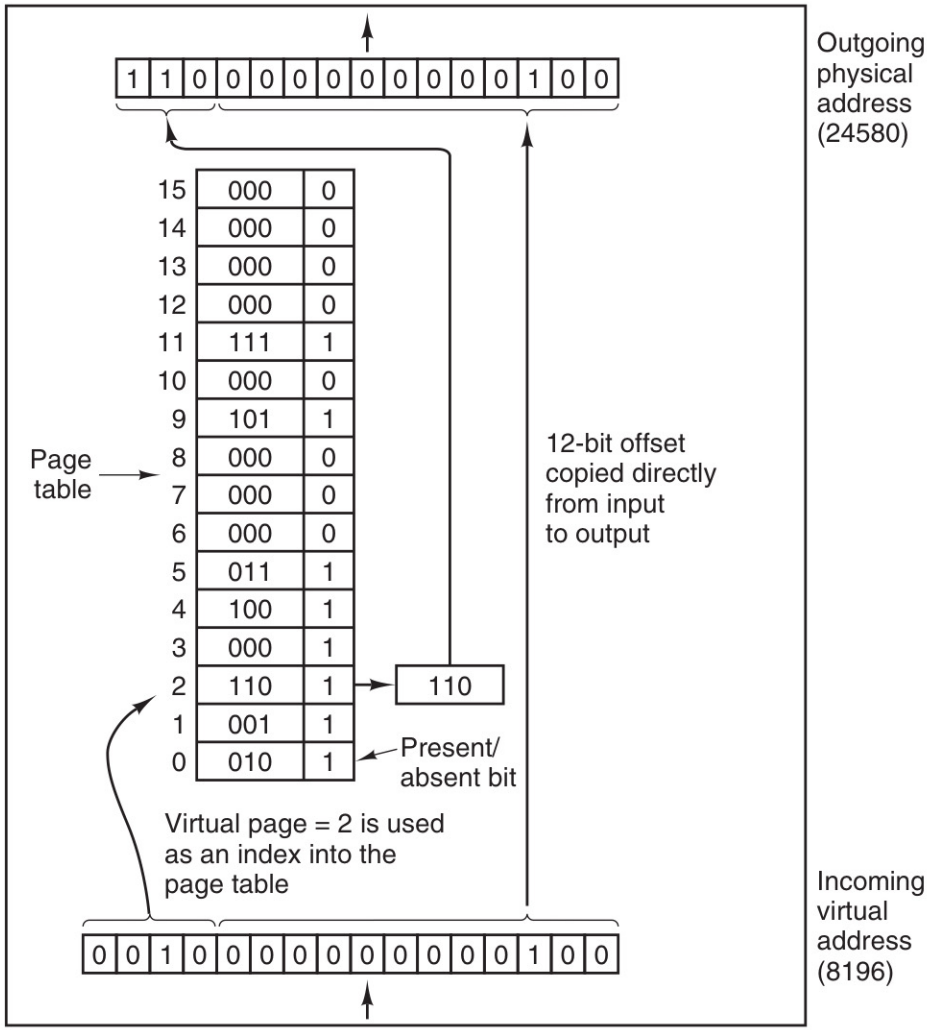
- Demand paging brings a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

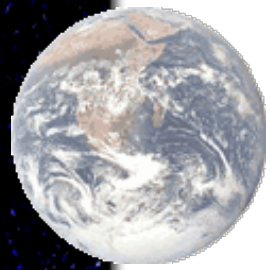
Demand Paging and Address Tranlation

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



Internal operation of MMU with 16 4 KB pages



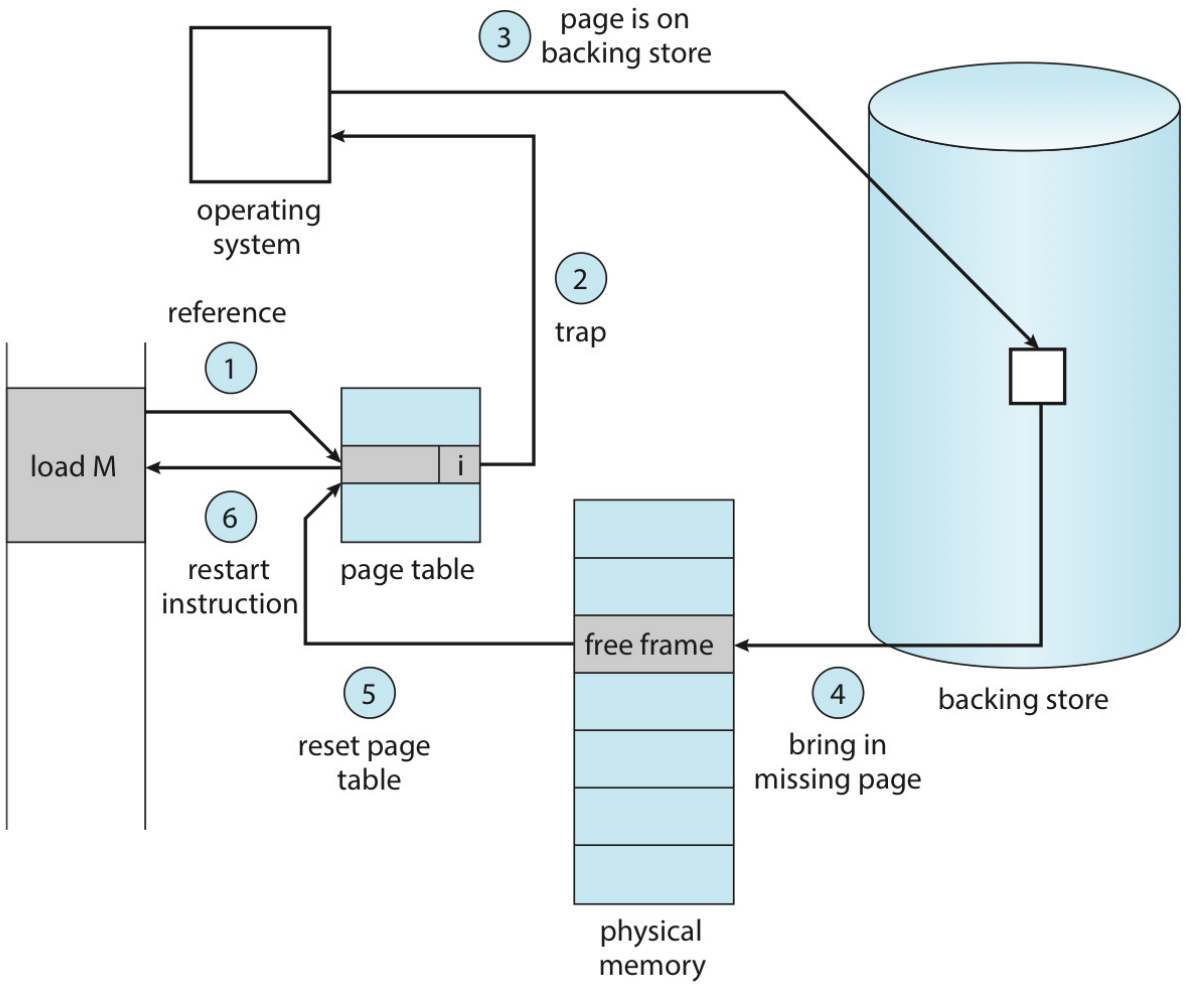


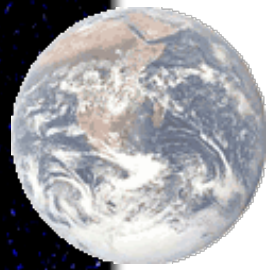
Page Fault

If there is a reference to a page, first reference to that page will trap to operating system, which is called page fault.

1. Operating system looks at another table to decide:
   - Invalid reference → abort
   - Just not in memory
2. Find free frame.
3. Swap page into frame via scheduled disk operation.
4. Reset tables to indicate page now in memory; set present/absent bit to 1.
5. Restart the instruction that caused the page fault.

Steps in Handling a Page Fault

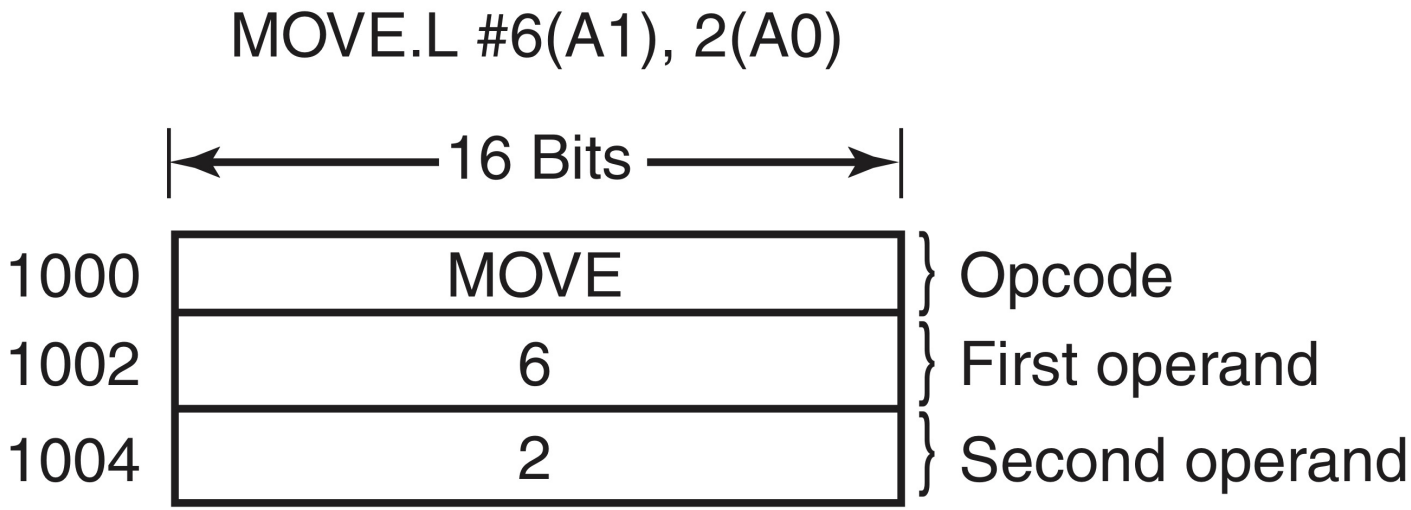


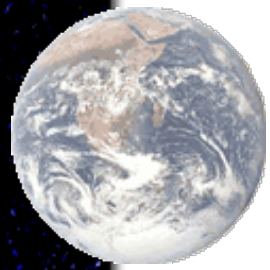


Aspects of Demand Paging

- Pure demand paging – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
- Hardware support needed for demand paging
  - Page table with present/absent bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

Instruction Restart

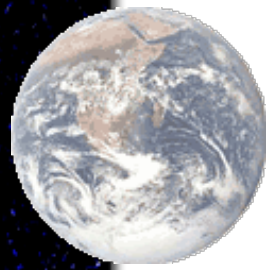




Stages in Demand Paging (Worst Case)

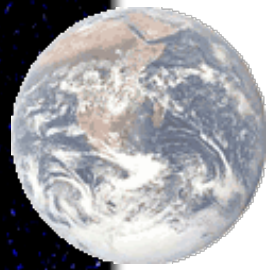
For the worst case, there are 12 stages in demand paging:

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   - 5.1 Wait in a queue for this device until the read request is serviced
   - 5.2 Wait for the device seek and/or latency time
   - 5.3 Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user



Stages in Demand Paging (Worst Case)

7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

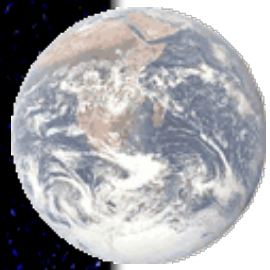


Performance of Demand Paging

- Three major activities:
  - Service the interrupt – several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page fault rate $0 \leq p \leq 1$
  - If $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = $(1 - p)$ x memory access

  + p x (page fault overhead + swap page out + swap page in)



Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

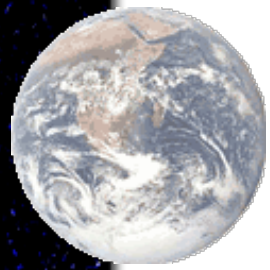
  = (200 + 7,999,800 x p) nanoseconds
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds → This is a slowdown by a factor of 40!!
- If we want performance degradation < 10%

  220 > (200 + 7,999,800 x p)

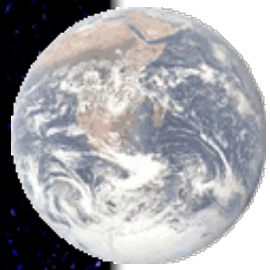
  20 > 7,999,800 x p

  p < 0.0000025 → one page fault in every 400,000 memory accesses



What Happens if There is no Free Frame?

- Where free frames of physical memory go?
  - Used up by process pages
  - In demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Performance – want an algorithm which will result in minimum number of page faults
- Some pages may be brought into memory several times.

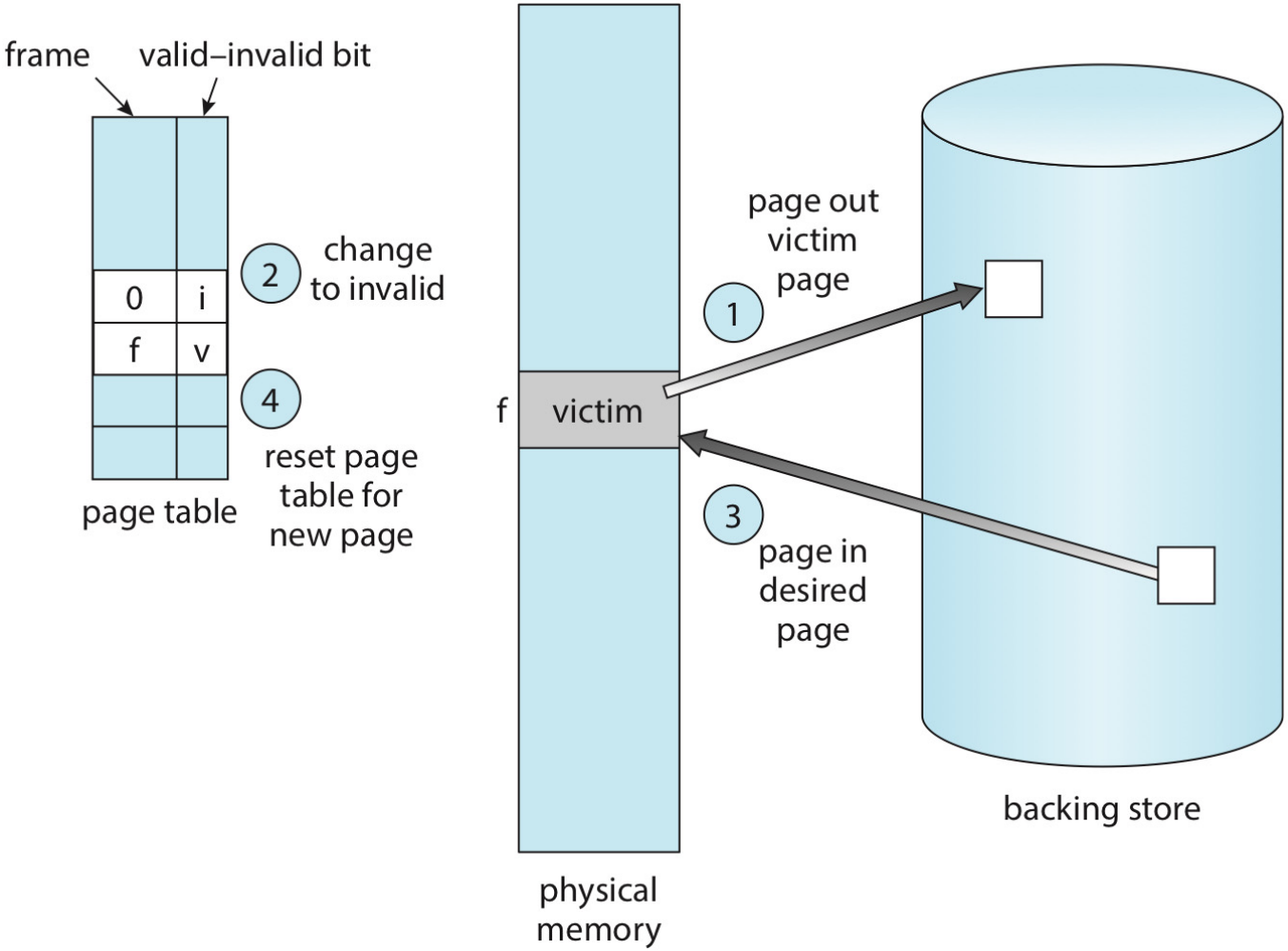


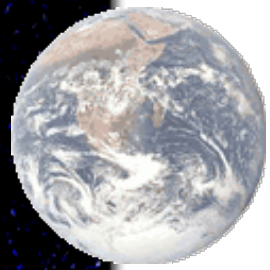
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note: now potentially 2 page transfers for page fault – increasing EAT

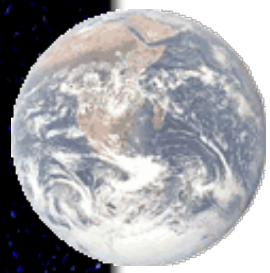
Basic Page Replacement



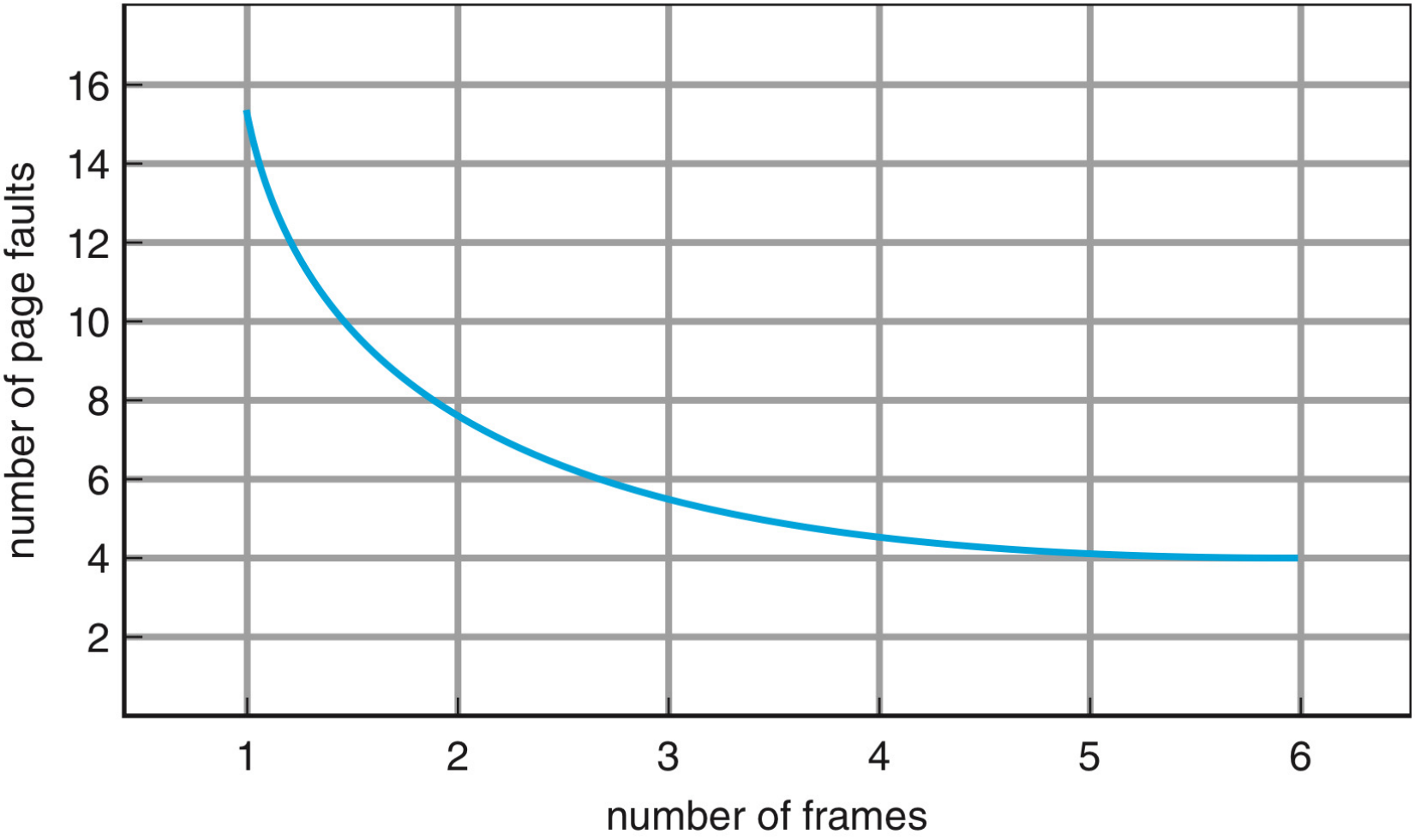


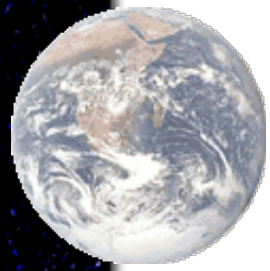
Frame Allocation and Page Replacement Algortithm

- Frame allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page replacement algorithm
  - Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - Reference string is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available



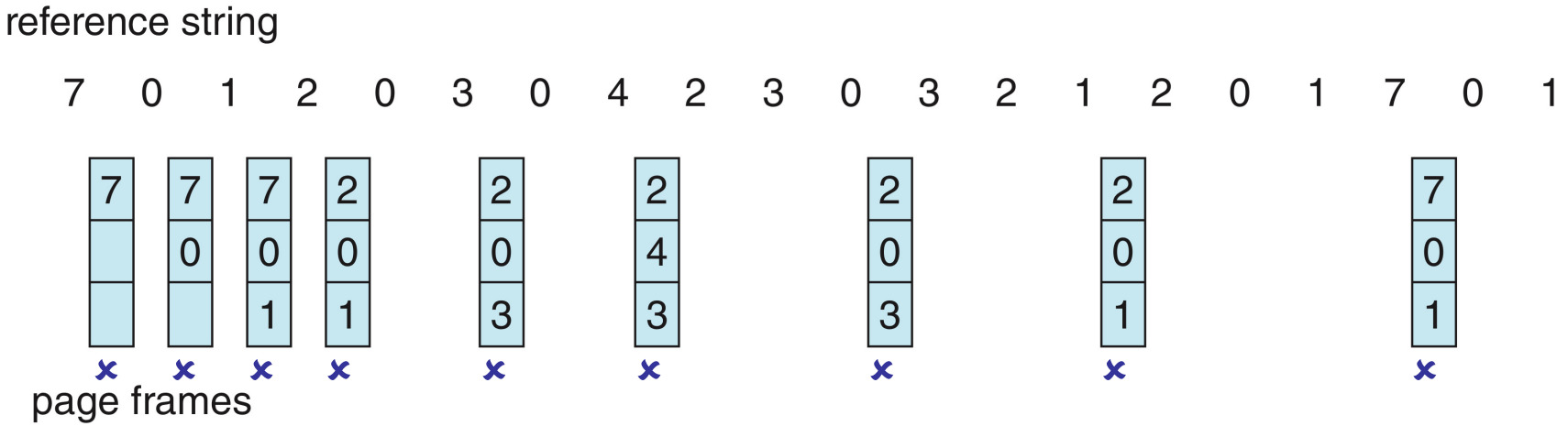
Page Faults vs. The Number of Frames



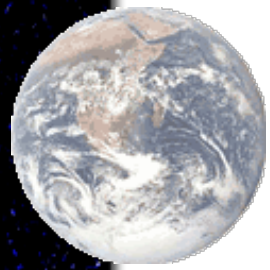


1. Optimal Algorithm

- Replace page that will not be used for longest period of time.
- Each page is labeled with the number of instructions that will be executed before that page is first referenced.
  - The page with highest label
- Best algorithm but impossible to implement; unrealizable algorithm.
- Used for measuring how well your algorithm performs.

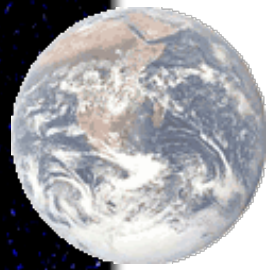


9 page faults



2. Not Recently Used

- Two status bits are associated with each page table entry:
  - R bit is set whenever the page is referenced
  - M bit is set when the page is modified
- The bits are updated on every memory reference by hardware
- Once a bit is set to 1, it stays 1 until the OS resets it
- Periodically, the R bit is cleared (e.g., every clock interrupt)

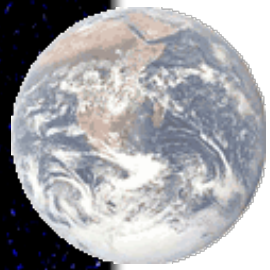


Not Recently Used

- When page fault occurs, pages can be categorized into 4 classes:

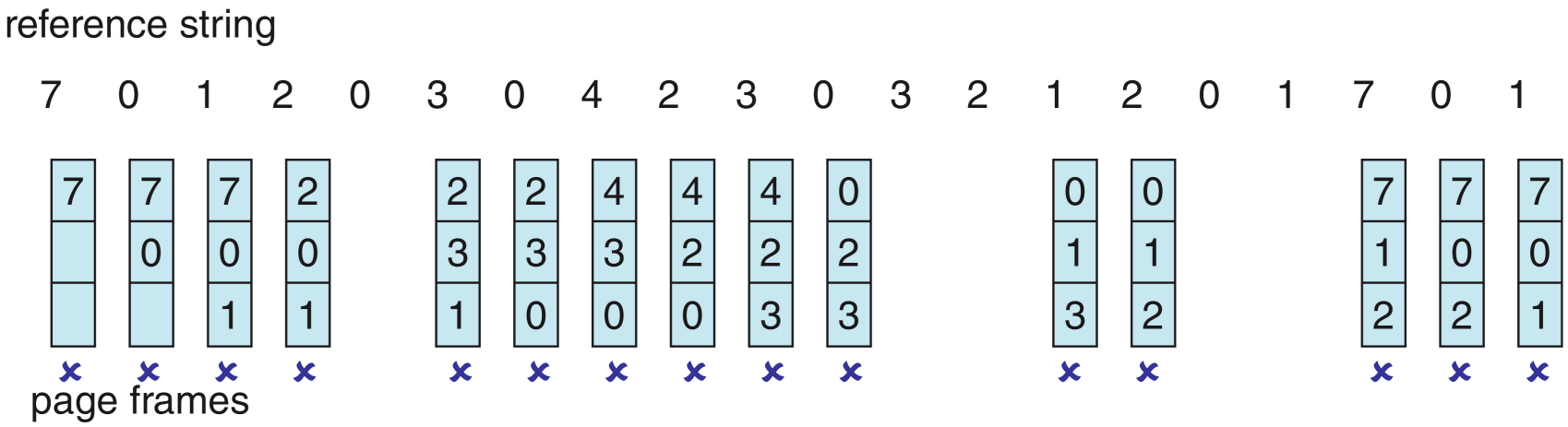
	Referenced	Modified	
class 0	✘	✘	
class 1	✘	✓	How can this happen?
class 2	✓	✘	
class 3	✓	✓	

- Removes a page at random from the lowest number of nonempty class.
- Better to remove a modified page that has not been referenced in at least one clock tick than a clean page that is in heavy use.
- Easy to understand, efficient to implement with adequate performance.

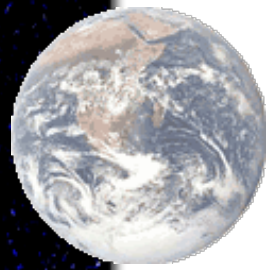


3. First-In-First-Out (FIFO) Algorithm

- OS maintains a list of all pages currently in memory, sorted by load time.
- Removes the oldest page according to its load time and the new page is added to the end of the list.
- Usually throws out a heavily used page, rarely used.



15 page faults

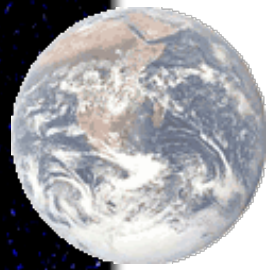


FIFO with Belady's Anomaly

Consider reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

	1	2	3	4	1	2	5	1	2	3	4	5	
	1	2	3	4	1	2	5	5	5	3	4	4	youngest page
		1	2	3	4	1	2	2	2	5	3	3	
			1	2	3	4	1	1	1	2	5	5	oldest page
	x	x	x	x	x	x	x			x	x		9 page faults

	1	2	3	4	1	2	5	1	2	3	4	5	
	1	2	3	4	4	4	5	1	2	3	4	5	youngest page
		1	2	3	3	3	4	5	1	2	3	4	
			1	2	2	2	3	4	5	1	2	3	oldest page
				1	1	1	2	3	4	5	1	1	10 page faults
	x	x	x	x			x	x	x	x	x	x	

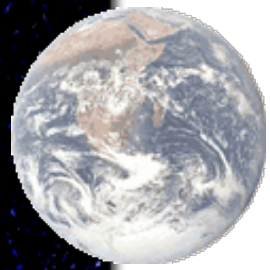


FIFO without Belady's Anomaly

0	1	2	3	1	4	0	2	3	2	reference string
	0	1	2	3	3	4	0	2	3	3	youngest page
		0	1	2	2	3	4	0	2	2	
			0	1	1	2	3	4	0	0	oldest page
✘	✘	✘	✘		✘	✘	✘	✘		8 page faults

0	1	2	3	1	4	0	2	3	2	reference string
	0	1	2	3	3	4	0	0	0	0	youngest page
		0	1	2	2	3	4	4	4	4	
			0	1	1	2	3	3	3	3	
				0	0	1	2	2	2	2	oldest page
✘	✘	✘	✘		✘	✘				6 page faults

- More page frame, less page fualt.

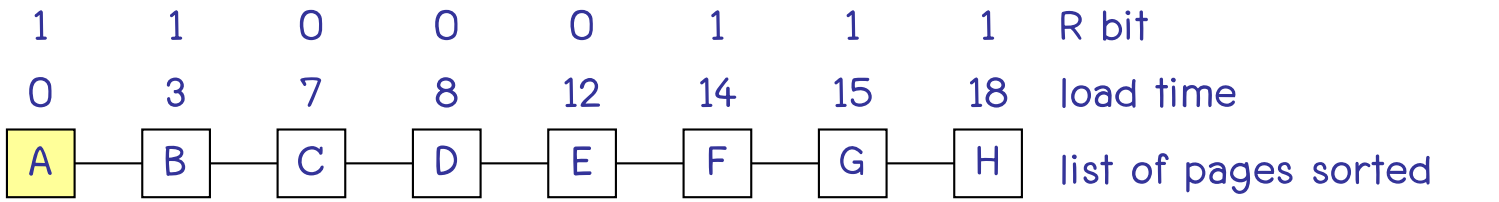


4. Second Chance Algorithm

- Same as FIFO replacement algorithm, but also take the R bit value into consideration.
- Removes old and unused page in previous clock interval:
  - R bit = 0 → the page is old and unused
  - R bit = 1 → reset R bit, put the page onto the end of the list and update its load time to the current time
- What happens when all pages have been referenced in the previous clock interval?

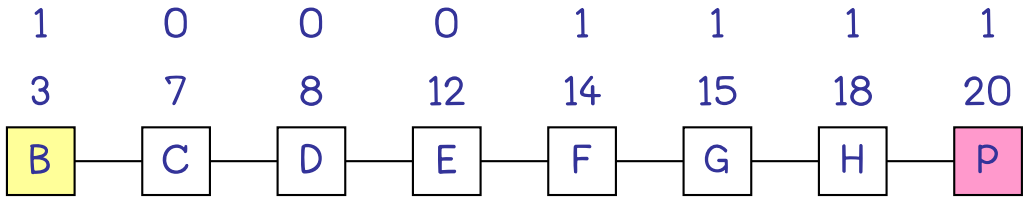
FIFO vs. Second Chance Algorithm

Initial state

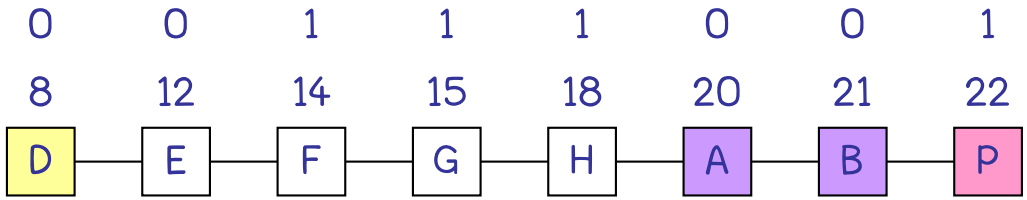


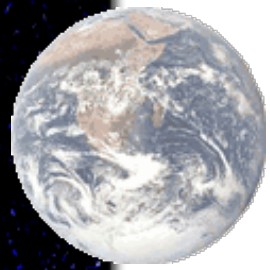
Assume at time 20 page fault occurs to bring in page P

FIFO



Second chance



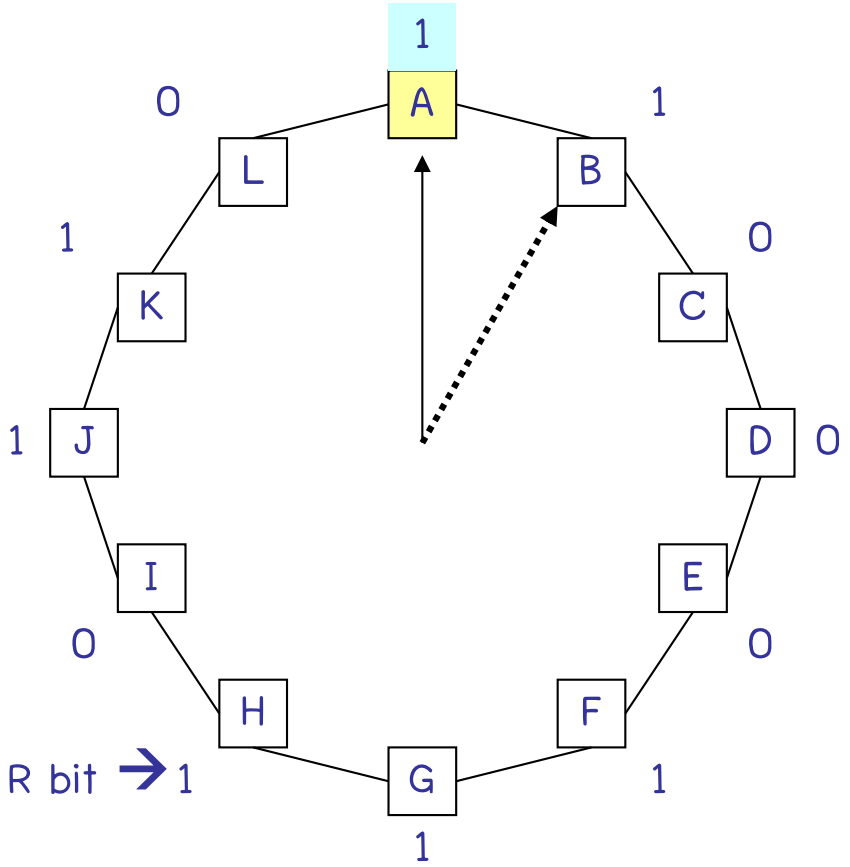


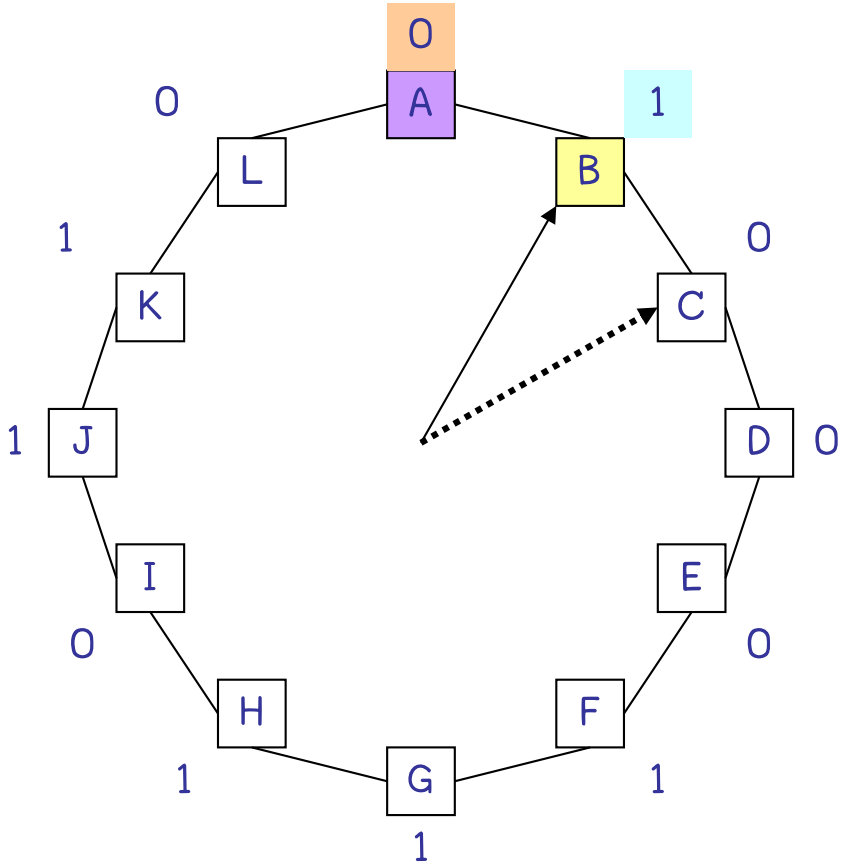
4.1. Clock Algorithm

- Maintains all the page on a circular list in the form of a clock with a hand points to the oldest page.
- The page being pointed to by the hand is inspected. The action taken depends on the R bit value.
  - R bit = 0 → evict the page
  - R bit = 1 → reset R bit and advance a hand (clockwise)

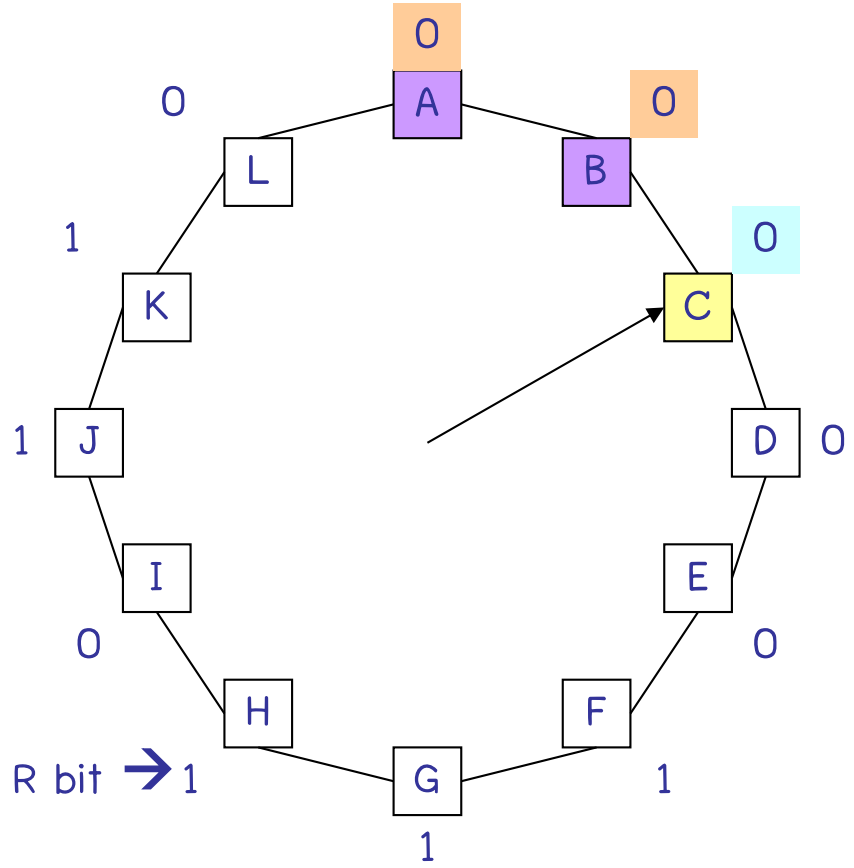
Clock Algorithm

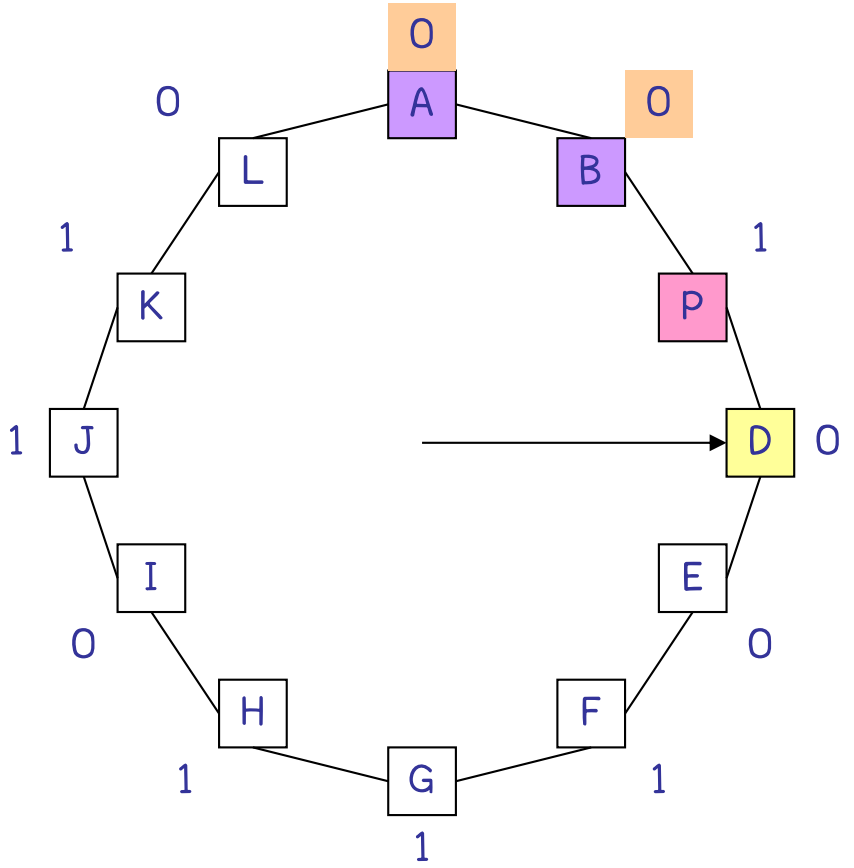
Assume at time t_i page fault occurs to bring in page P.

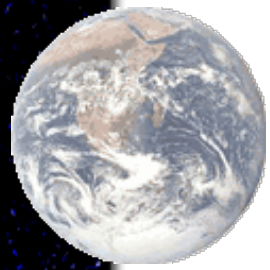




Clock Algorithm

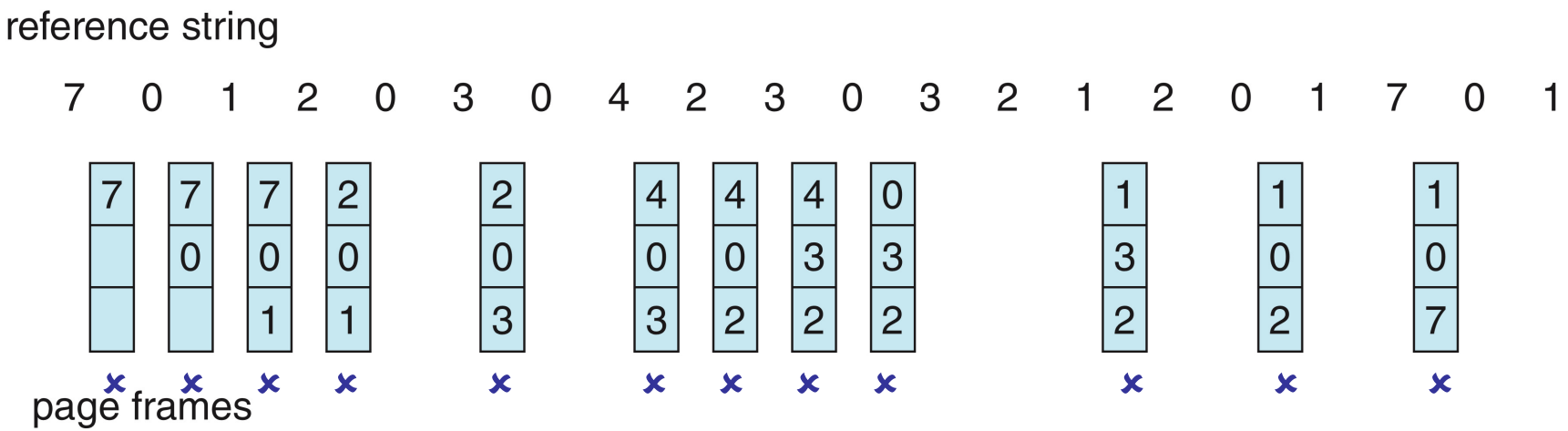






5. Least Recently Used (LRU) Algorithm

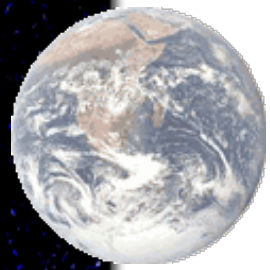
- Assume pages used recently will used again soon
  - throw out page that has been unused for longest time
- Associate time of last use with each page, updated at every memory reference.



12 faults – better than FIFO but worse than optimal algorithm.

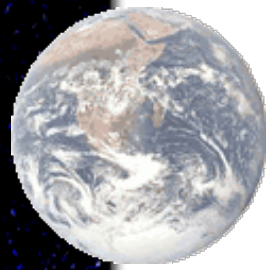
Theoretically realizable, but not cheap.

Generally good algorithm and frequently used, but how to implement?



The LRU Implementations

- Hardware implementations of LRU:
  - A 64-bit counter
  - An $n \times n$ matrix
- Software implementations of LRU:
  - The not frequently used algorithm
  - The aging algorithm

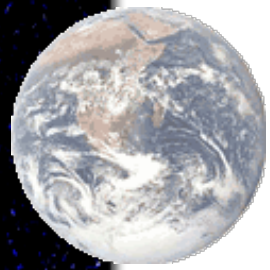


LRU: 64-bit Counter

- A 64-bit counter is associated with each page table entry, initially zero.
- The counter of the page referenced is automatically incremented after every instruction.
- Removes the page with the lowest counter.

page 0	0000 0000 0000 ... ... 0000 0000 0110
page 1	0000 0000 0000 ... ... 0000 0101 0001
page 2	0000 0000 0000 ... ... 0000 0011 0000
page 3	0000 0000 0000 ... ... 1000 1000 1000
page 4	0000 0000 0000 ... ... 0000 0000 0101
page 5	0000 0000 0000 ... ... 0001 0001 0001

64-bit hardware counter



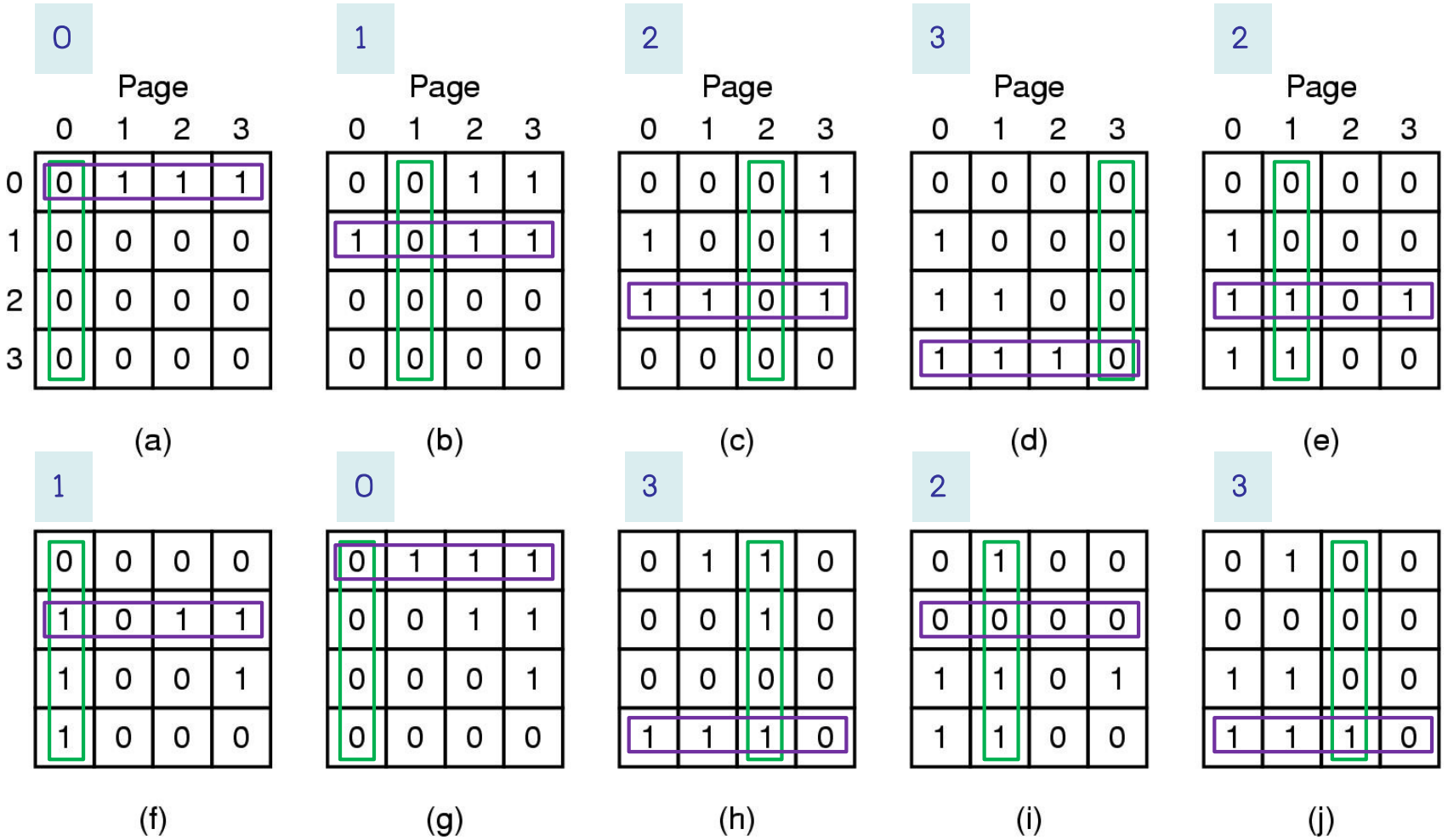
LRU: $n \times n$ Matrix

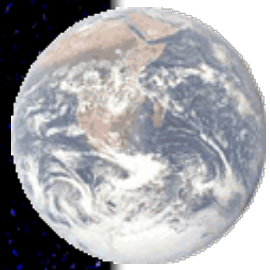
- A matrix of $n \times n$ bits, initially all zeroes, for n page frame machine.
- When a page frame k is referenced set all bits of row k to one and set all bits of column k to zero.
- Removes the page corresponding to a row with lowest binary value.

- Why page with lowest binary value is the LRU page?

LRU: $n \times n$ Matrix

Reference string: 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

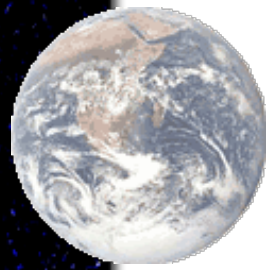




5.1. LRU: Not Frequently Used Algorithm

- A software counter associated with each page table entry, initially zero.
- At each clock interrupt the R bit, either 0 or 1, of each page is added to the counter.
- Removes the page with the lowest counter.
- The algorithm never forgets anything.

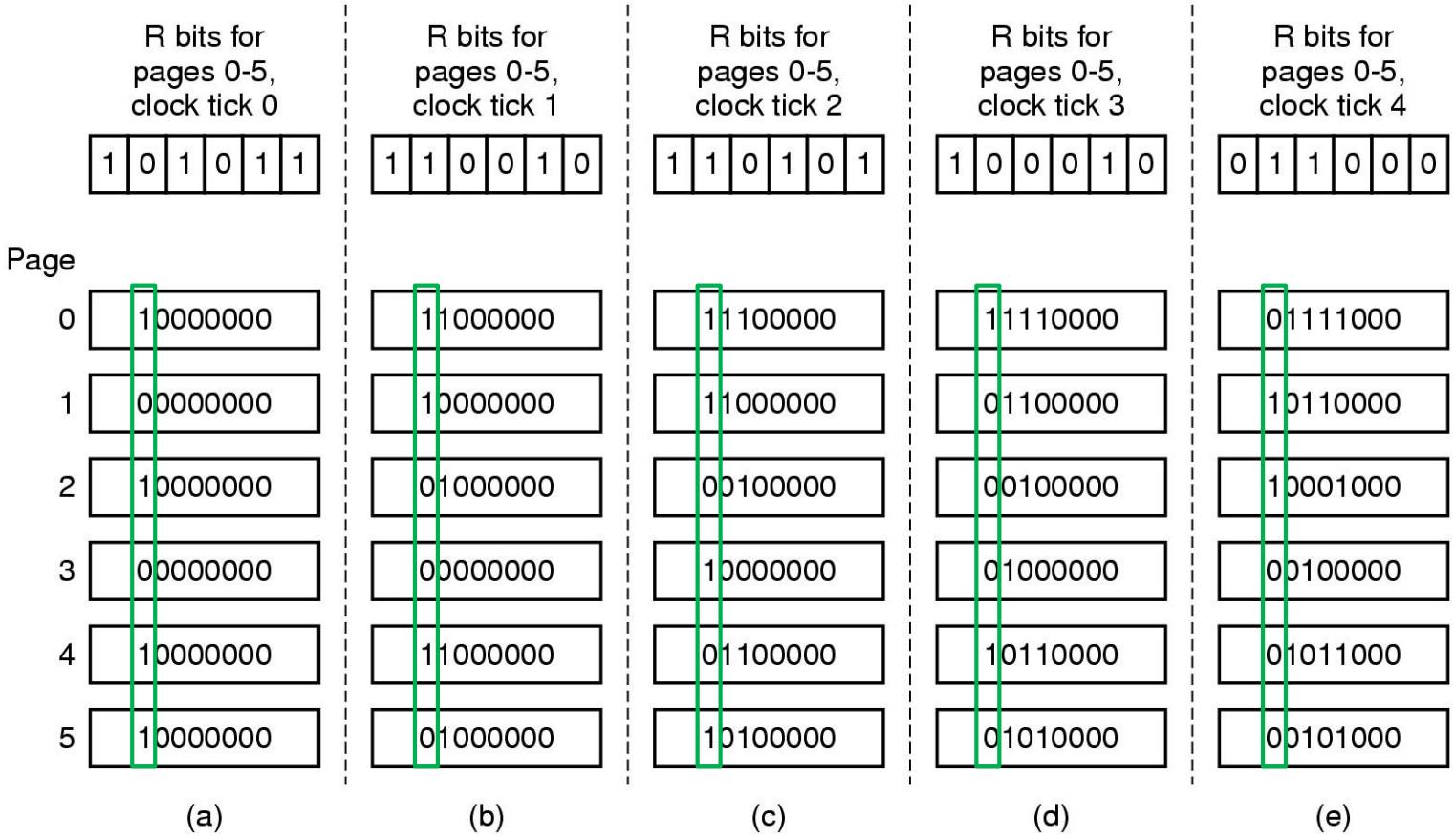
page 0	0000 0010
page 1	0000 0100
page 2	0000 0010
page 3	0001 0000
page 4	0010 0000
page 5	1000 0000

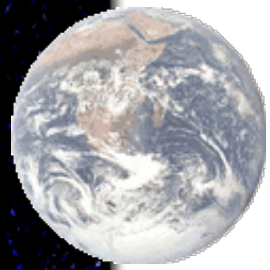


5.2. LRU: Aging Algorithm

- A software counter associated with each page table entry, initially zero.
- Shift the counter right one bit before adding R bit.
- R bit is added to the leftmost bit.
- Removes the page with the lowest counter.
- Lost the ability to distinguish references occur earlier in the same time interval since it records one bit per time interval.
- The counter is of limited size so it can have limited memorization.

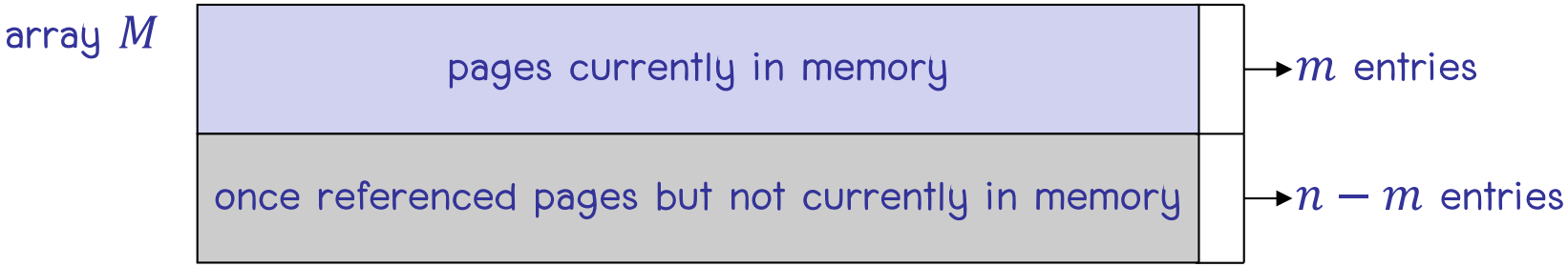
LRU: Aging Algorithm



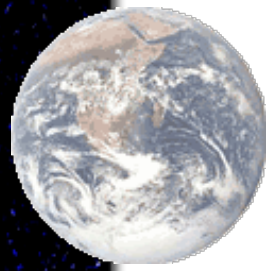


Stack Algorithm

- Array M is used to keep track of the state of memory.
- M contains n elements, where n is a number of the process virtual pages, and m is a number of page frames.



- Array M is initially empty
- Newly referenced page is loaded into the empty slot in the top part.
- when the top part is full, invoke a replacement algorithm and remove a page from the top part to the bottom part.

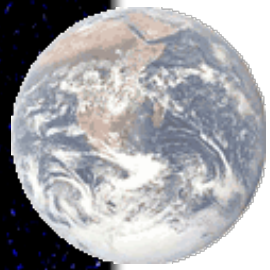


Stack Algortihm

- consider a paging system with four page frames, $m = 4$, and eight virtual pages, $n = 8$.

Reference string

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
	×	×	×	×	×	×	×		×					×			×						×	

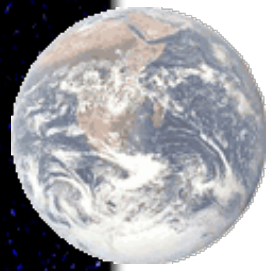


Property of The Stack Algorithm

$$M(m,r) \subseteq M(m+1,r)$$

where r is an index into the reference string.

"Pages in the top part of r for memory with n page frames after r memory references are also in M for memory with m+1 page frames"



Stack Algorithm with More Frames

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
	✘	✘	✘	✘	✘	✘	✘		✘					✘			✘						✘	

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
	✘	✘	✘	✘	✘	✘	✘		✘								✘							

Stack Algorithm vs FIFO

$M(3,7) \subseteq M(4,7)$

$r = 7$

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	2	1	2	3	4
			1	2	3	4	1	5	5	1	2	3
	×	×	×	×	×	×	×			×	×	×

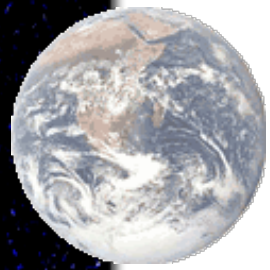
$M(3,7) \nsubseteq M(4,7)$

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
			1	2	3	4	1	1	1	2	5	5
	×	×	×	×	×	×	×			×	×	

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
	×	×	×	×			×			×	×	×

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
				1	1	1	2	3	4	5	1	1
	×	×	×	×			×	×	×	×	×	×

LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

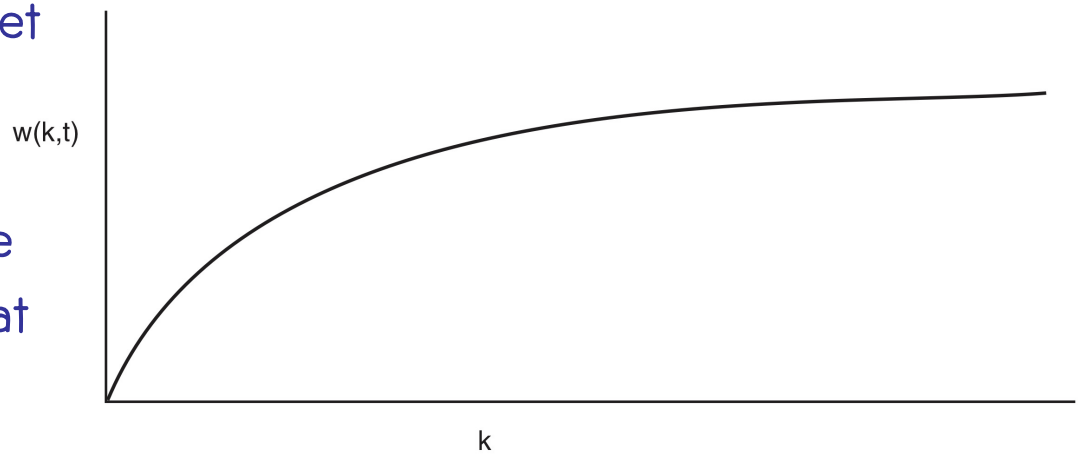


Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames → approximation of locality
- if $D > m$ → Thrashing (m is number of available frames)
- Policy if $D > m$ then suspend or swap out one of the processes

Working-Set Model

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

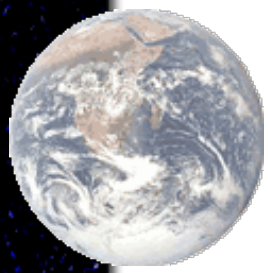


page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

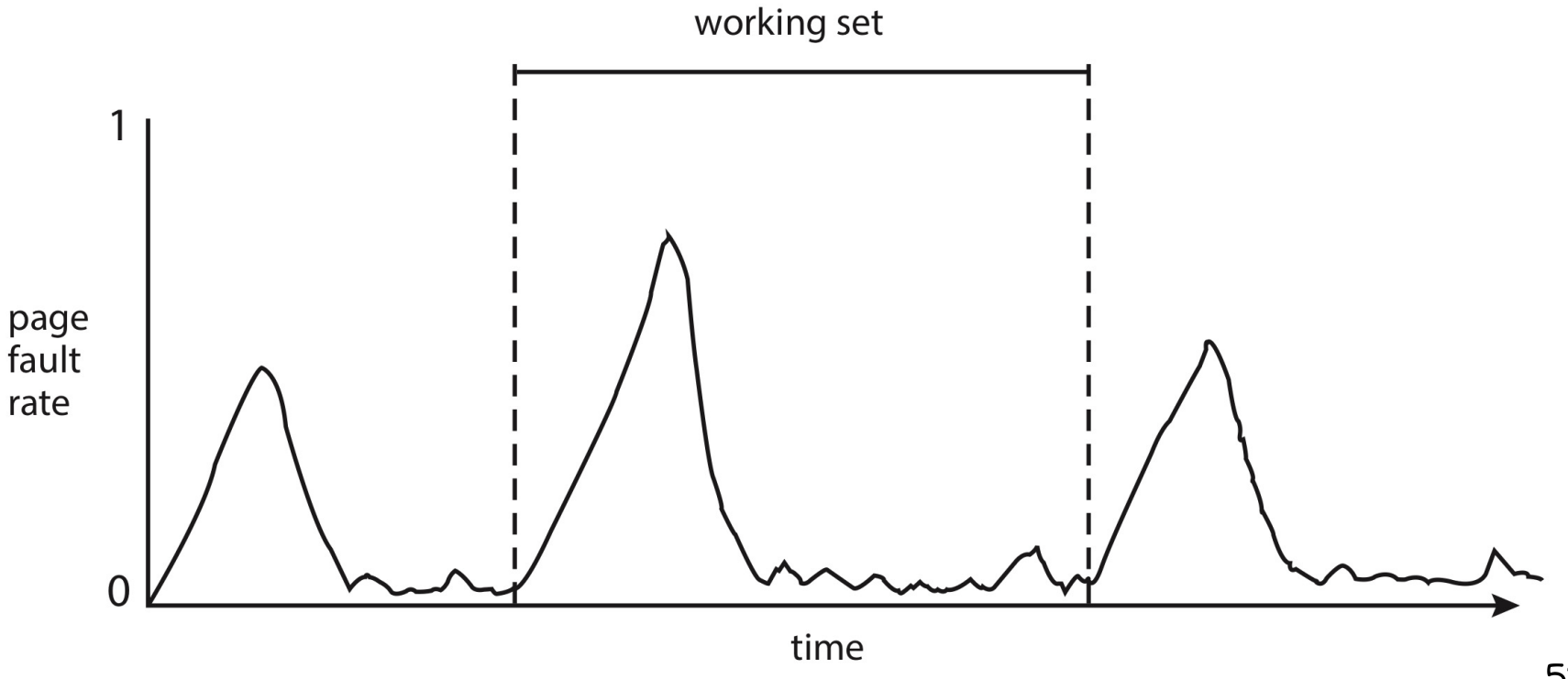
WS(t_1) = {1,2,5,6,7}

WS(t_2) = {3,4}

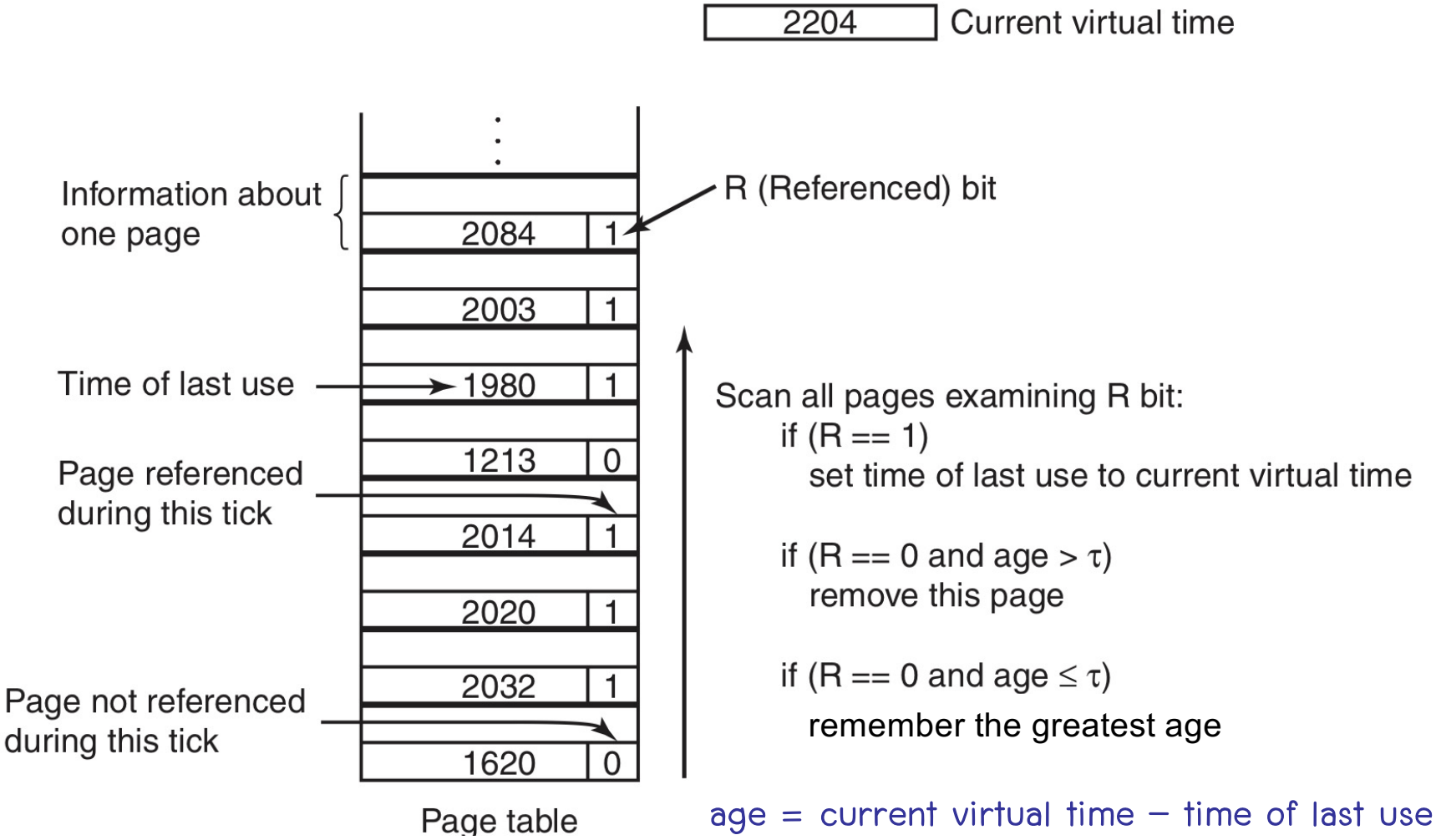


Working Sets and Page Fault Rates

- The working set of a process changes over time as references to data and code sections move from one locality to another.
- Assume no thrashing, the page-fault rate of the process will transition between peaks and valleys over time.

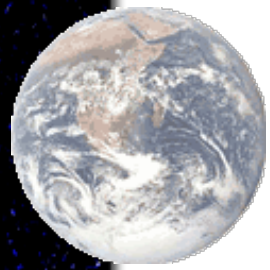


6. The Working Set Page Replacement Algorithm



age = current virtual time − time of last use

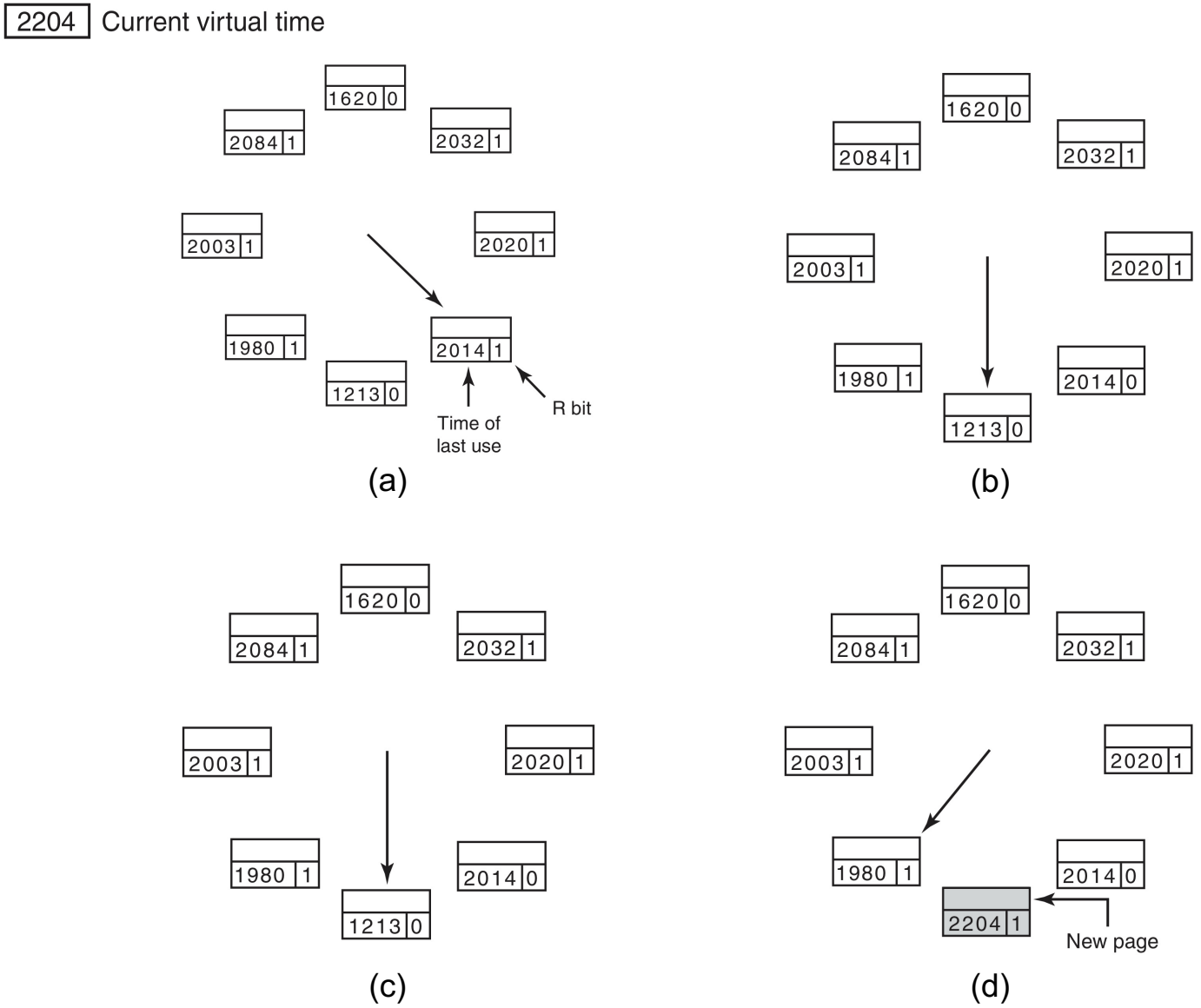
Assume, τ spans multiple clock ticks.

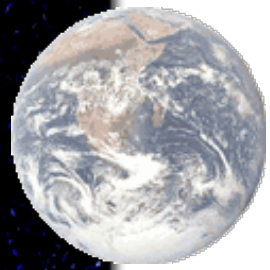


6.1. The Working Set Clock Page Replacement Algorithm

- To improve efficiency of the working set algorithm, which needs to search entire page table to find victim on every page fault.
- Based on the clock policy.

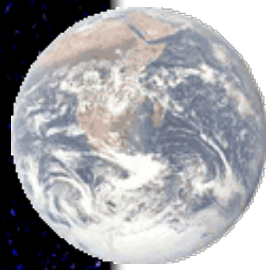
The Working Set Clock Algorithm





Summary of Page Replacement Algorithms

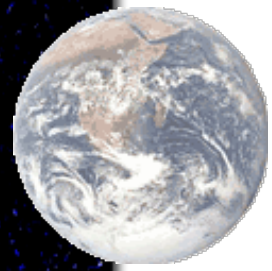
Algorithms	Comment
Optimal	Unrealizable, but useful as a benchmark
NRU	Very crude approxamation of LRU
FIFO	Might throw out important pages
Second Chance	Big improvement over FIFO
LRU	Excellent, but difficult to implement exactly
NFU	Fairly crude approxamation of LRU
Aging	Efficient algorithm that approximate LRU well
Working Set	Somewhat expensive to implement
WSClock	Good efficient algorithm



Frame Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- Local replacement – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Local versus Global Allocation Policies



	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

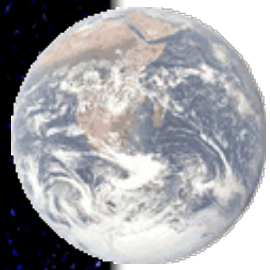
A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

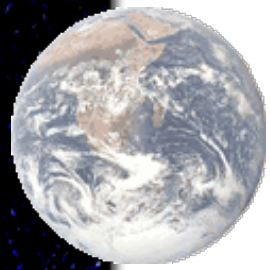
Original configuration Local page replacement Global page replacement



Load Control

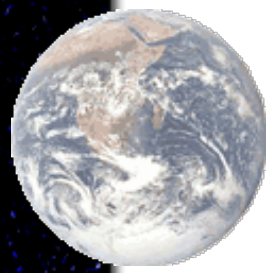
- Despite good designs, system may still thrash
- When PFF algorithm indicates
  - some processes need more memory
  - but no processes need less
- Solution :

  Reduce number of processes competing for memory
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming



Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system...
    - and the cycle continues ...
- Thrashing ≡ a process is busy swapping pages in and out

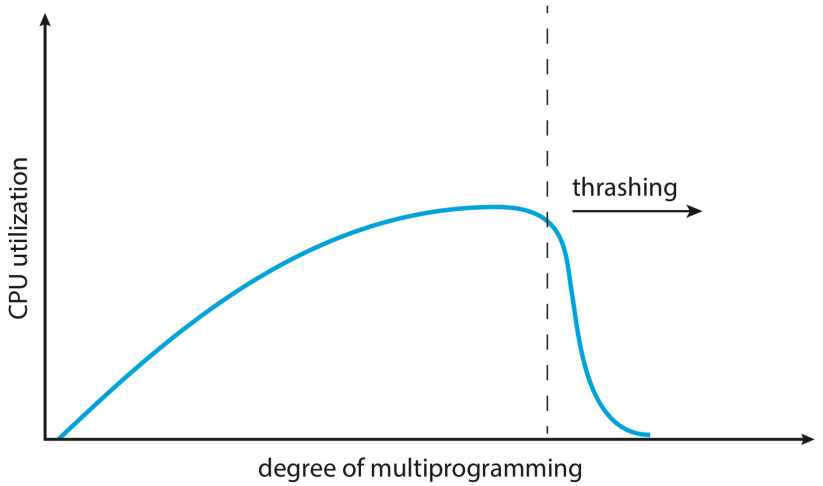


Thrashing

- Swapping out a piece of a process just before that piece is needed
- The processor spends most of its time swapping pieces rather than executing user instructions
- Why does paging work?

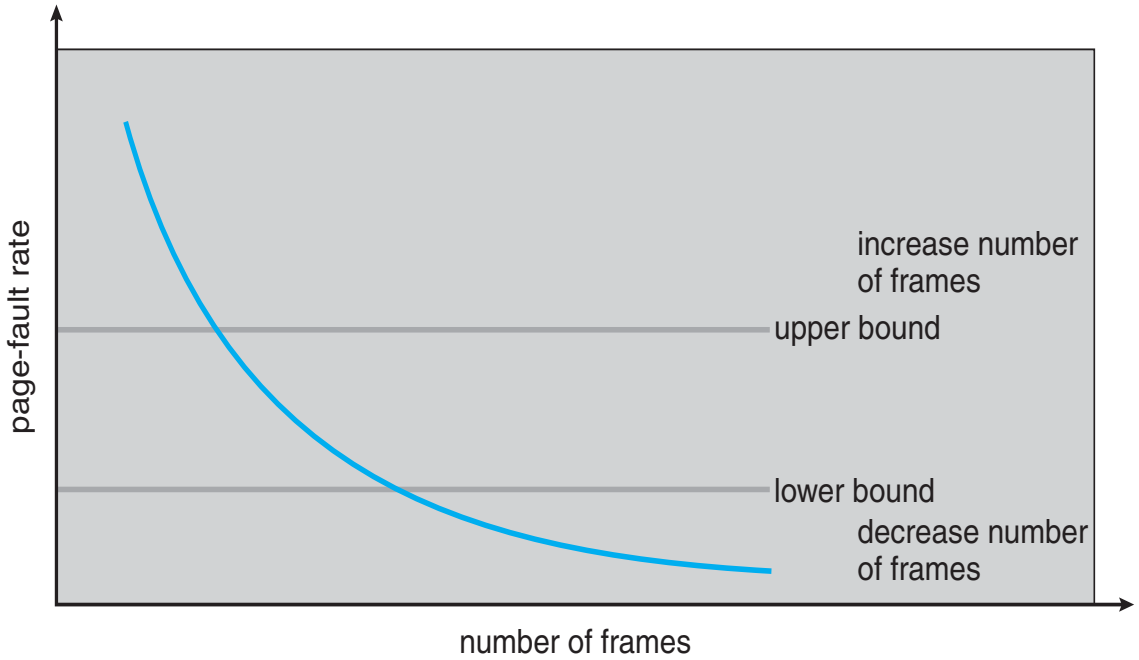
  Locality model
  - Process migrates from on locality to another.
  - Localities may overlap

$\sum size\ of\ locality >$ total memory size

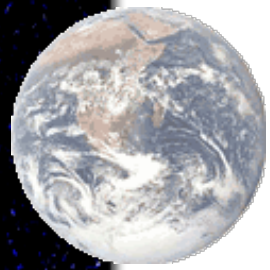


Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" page-fault frequency (PFF) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

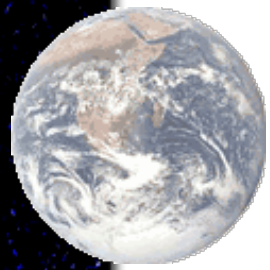


Page fault rate as a function of the number of page frames assigned



Process Suspension for Thrashing Avoidance

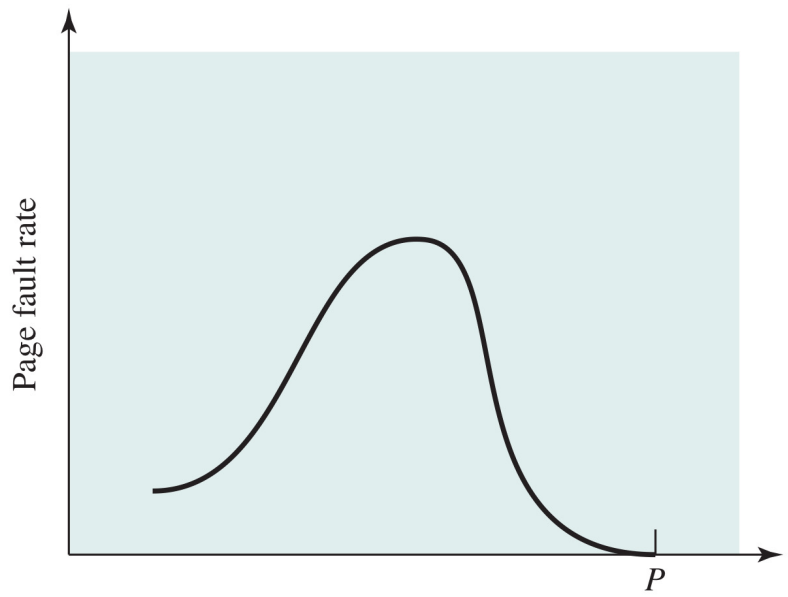
- Lowest priority process
- Faulting process
  - does not have its working set in main memory so will be blocked anyway
- Last process activated
  - this process is least likely to have its working set resident
- Process with smallest resident set
  - this process requires the least future effort to reload
- Largest process
  - obtains the most free frames
- Process with the largest remaining execution remaining execution window



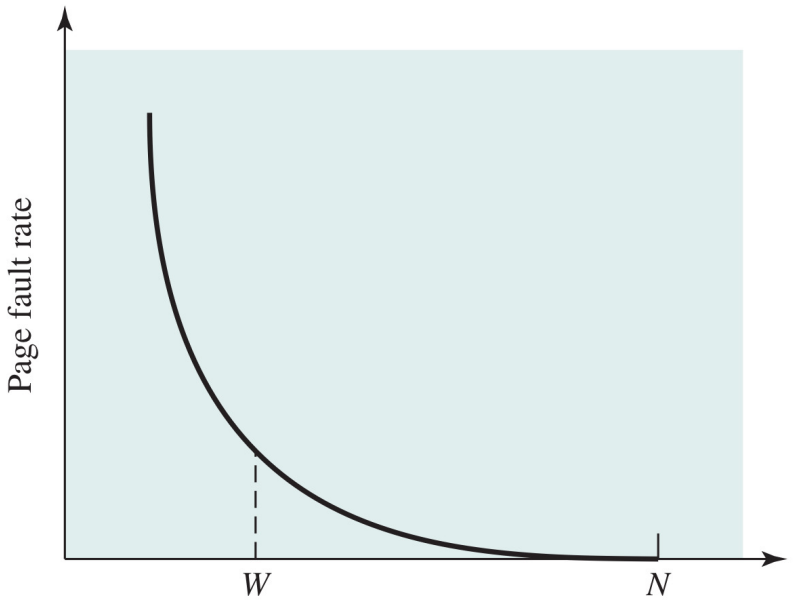
Page Size

- Small page size:
  - less internal fragmentation
  - more pages required per process
  - larger page tables (may not be always in main memory)
- Large page size:
  - large number of pages in main memory; as time goes on during execution the pages in memory will contain portions of the process near recent references. Page faults low.
  - Increased page size causes pages to contain locations further from recent reference. Page faults rise.
  - Page size approaching the size of the program: Page faults low again.

Typical Paging Behavior of a Program



(a) Page size

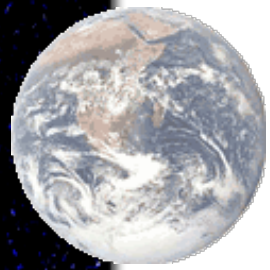


(b) Number of page frames allocated

P = size of entire process
W = working set size
N = total number of pages in process

Secondary memory designed to efficiently transfer large blocks

- favors large page size



Page Size

- Overhead due to page table and internal fragmentation

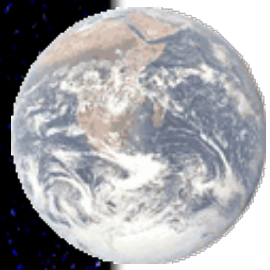
$$overhead = \frac{s \cdot e}{p} + \frac{p}{2}$$

where, s = average process size in bytes

p = page size in bytes

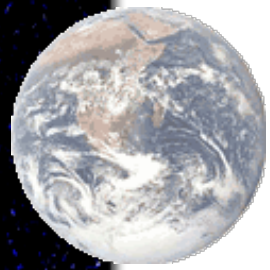
e = page entry length in bytes

- Optimized when $p = \sqrt{2se}$
- Multiple page sizes provide the flexibility needed (to also use TLBs efficiently):
  - Large pages can be used for program instructions
  - Small pages can be used for threads



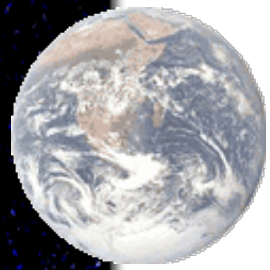
Cleaning Policy

- Need for a background process, paging daemon
  - periodically inspects state of memory
- Pre-cleaning: pages are written out in batches, off-line, periodically
- When too few frames are free, paging daemon
  - selects pages to evict using a replacement algorithm
  - can use same circular list (clock)
    - as regular page replacement algorithm but with different pointers



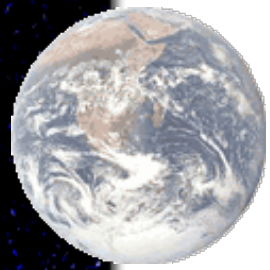
Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk



Fetch Policy

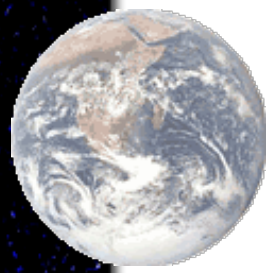
- Determines when a page should be brought into memory
- Demand paging only brings pages into main memory when a reference is made to it
  - Many page faults when process first started
- Pre-paging brings in more pages than needed
  - More efficient to bring in pages that reside contiguously on the disk



OS Involvement with Paging

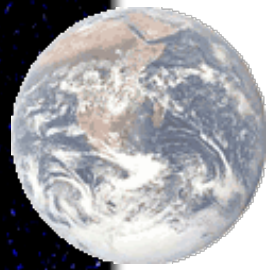
Four times when OS involved with paging:

1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages



Page Fault Handling

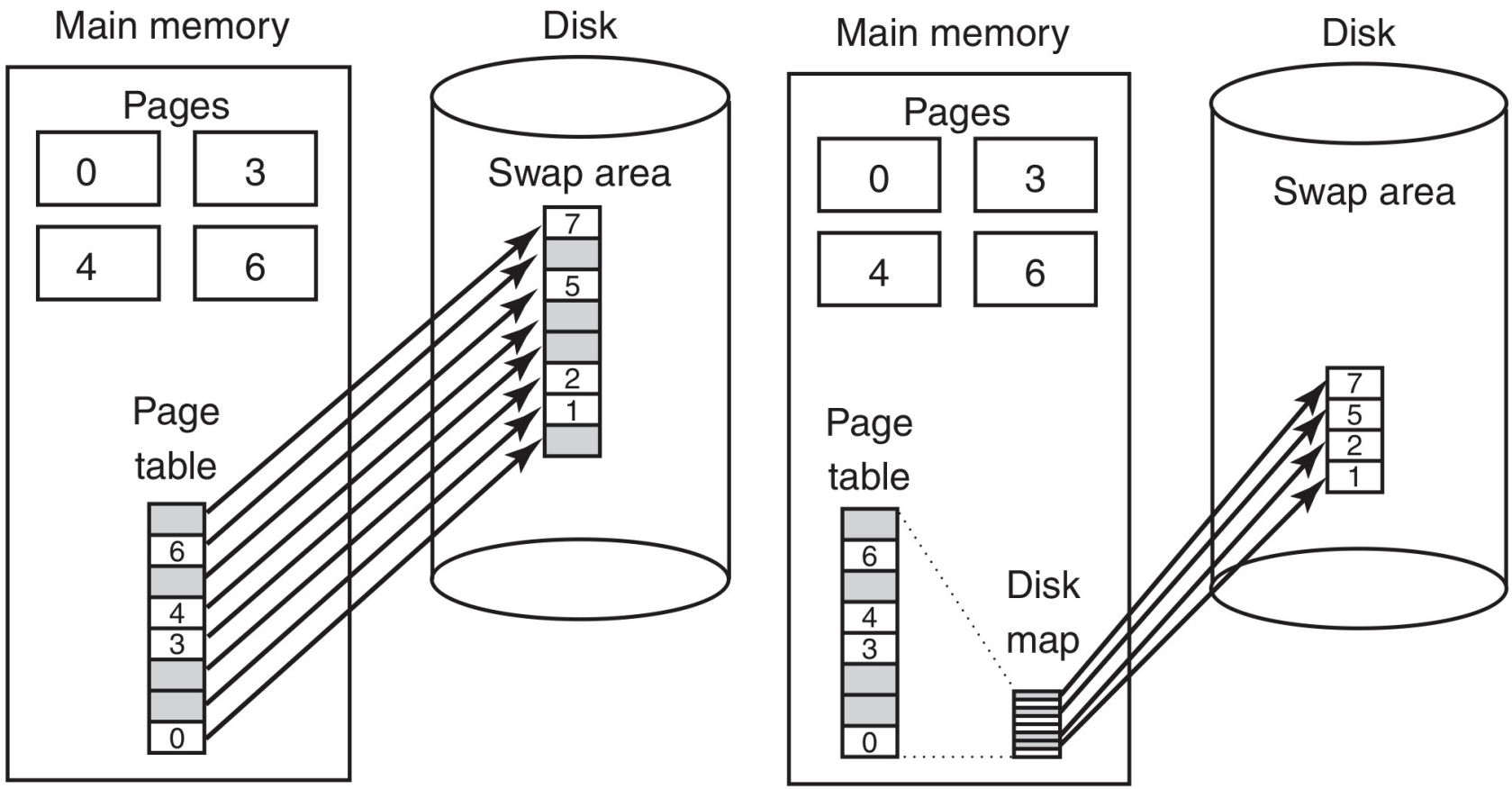
1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS brings schedules new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored
11. Program continues



Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Process issues call for read from device into buffer
  - while waiting for I/O, another processes starts up
  - has a page fault
  - buffer for the first proc may be chosen to be paged out
- Need to specify some pages locked
  - exempted from being target pages

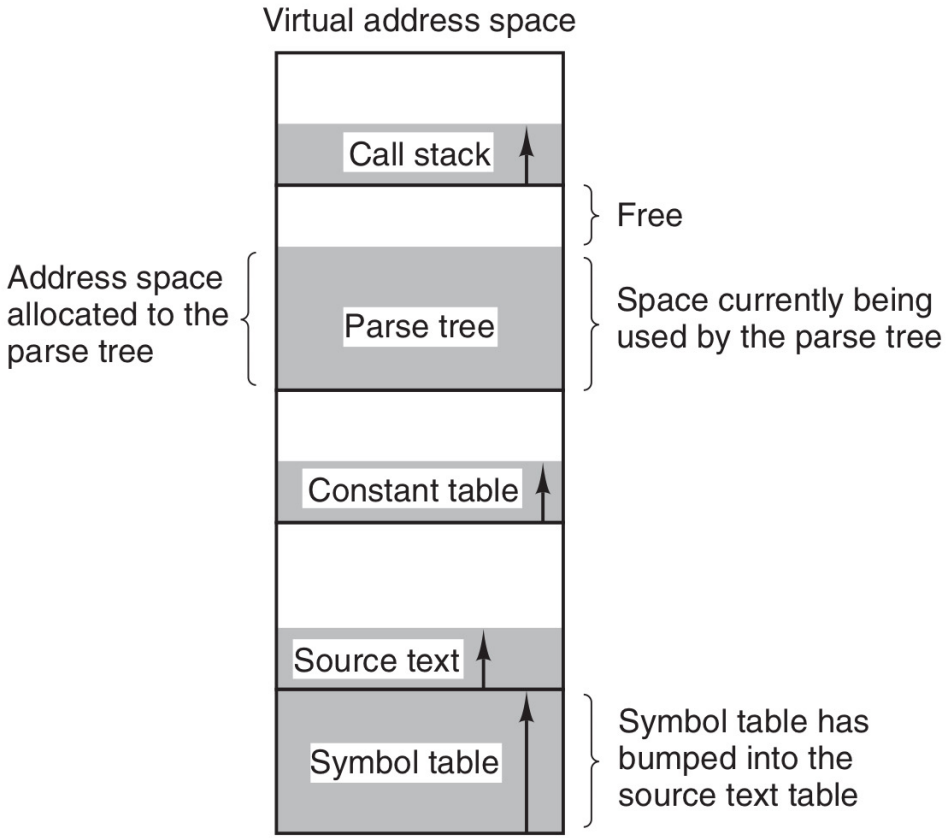
Backing Store



Paging to static swap area

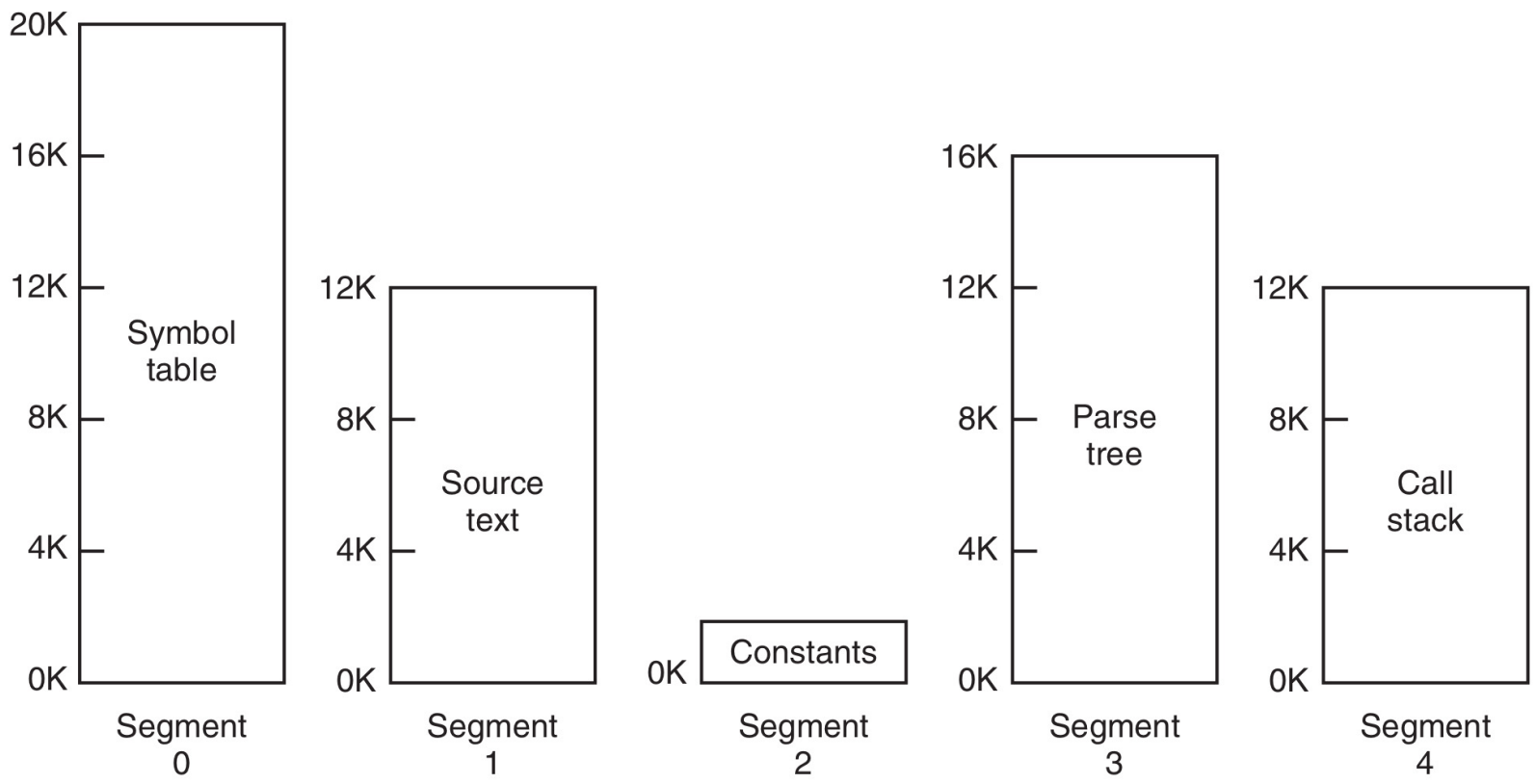
Backing up pages dynamically

Segmentation

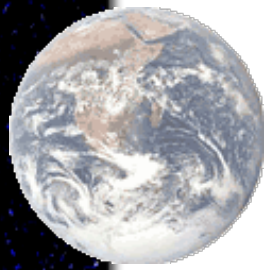


- One-dimensional address space with growing tables
- One table may bump into another

Segmentation



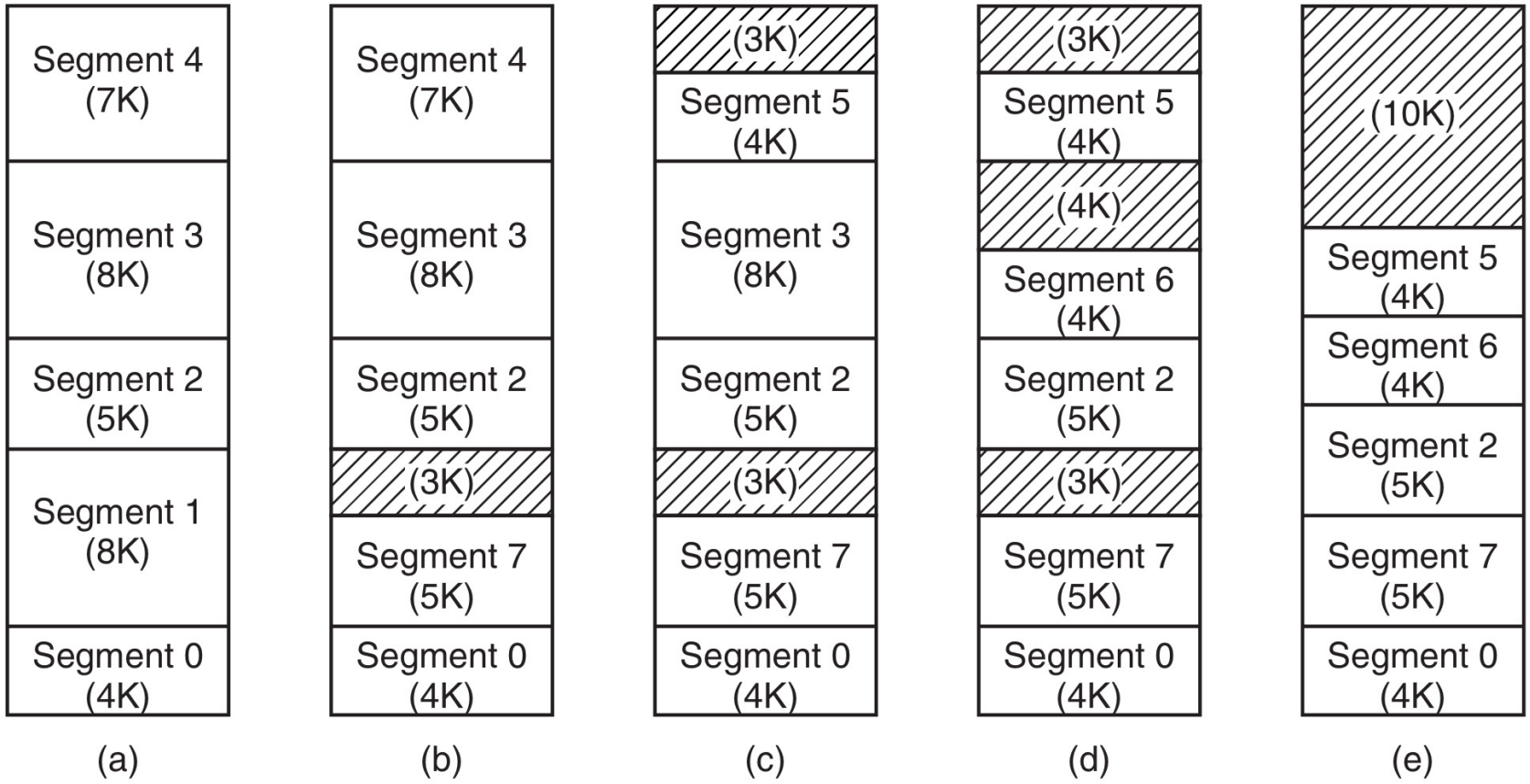
- Allows each table to grow or shrink, independently



Comparison of Paging and Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

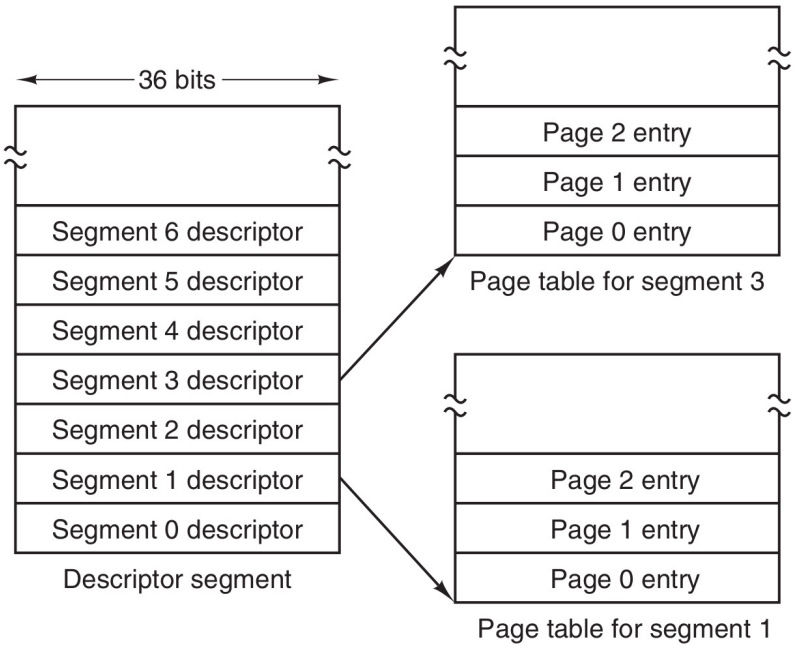
Implementation of Pure Segmentation

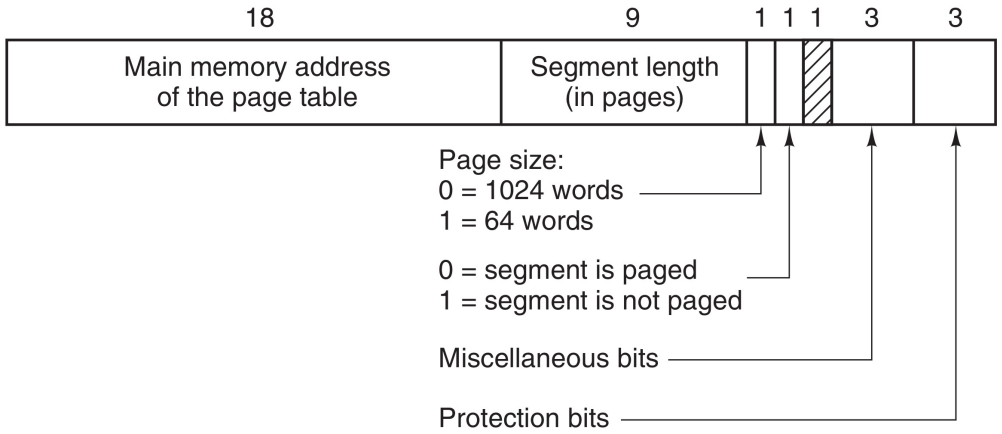


(a)-(d) Development of checkerboarding

(e) Removal of the checkerboarding by compaction

Segmentation with Paging: MULTICS

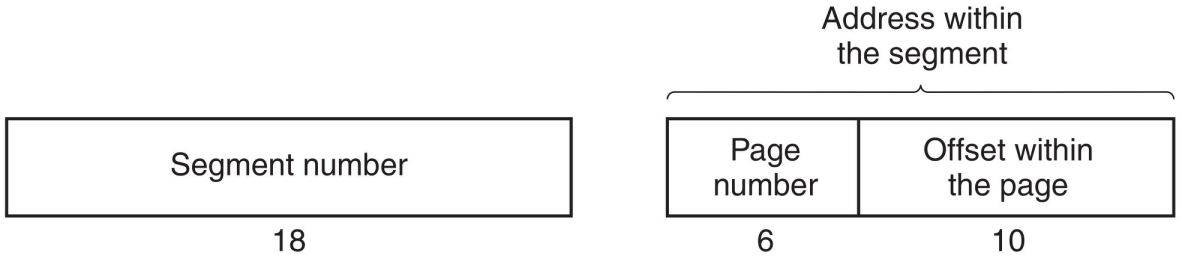




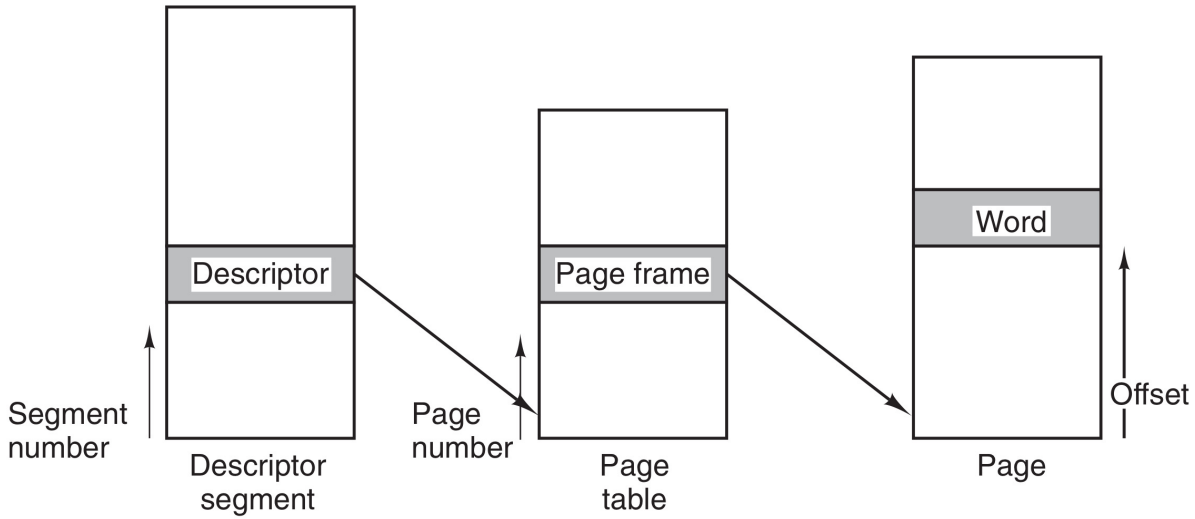
Descriptor segment points to page tables

Segment descriptor – numbers are field lengths

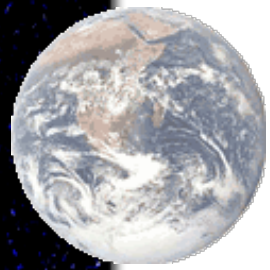
Segmentation with Paging: MULTICS



A 34-bit MULTICS virtual address



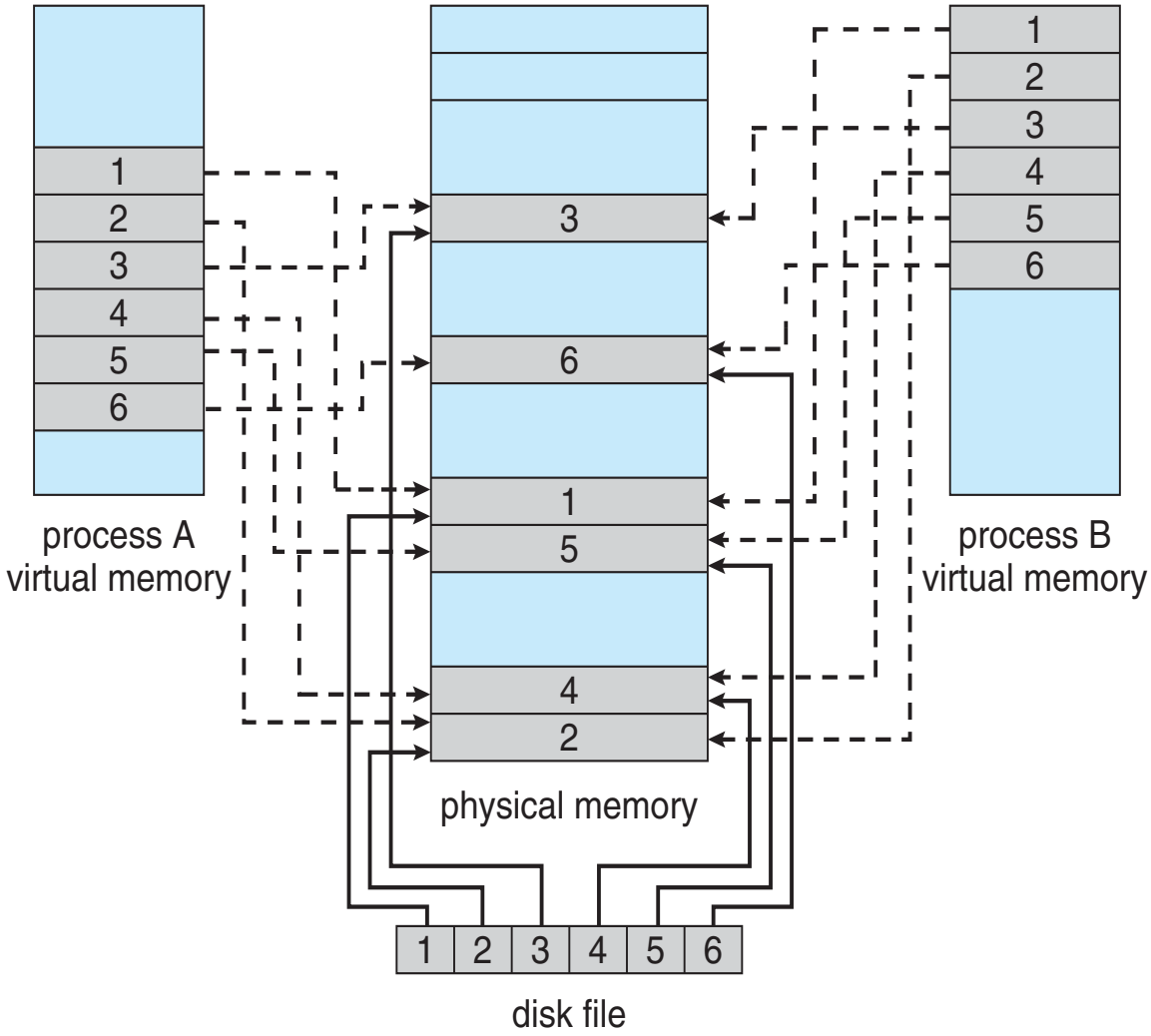
Translation of MULTICS virtual address to physical address

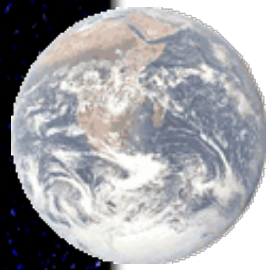


Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and/or at file `close()` time

Memory Mapped Files





References

1. Modern Operating Systems, 4th edition, Andrew S. Tanenbaum, Herbert Bos
2. Operating Systems, 3rd edition, H.M.Deitel, Pearson Education Limited: Longman.
3. Operating System Concepts, 10th edition, Abraham Silberschatz, Yale University, Peter Baer Galvin, Pluribus Networks, Greg Gagne, Westminster College, 10th edition. Wiley.
4. Operating Systems: Internals and Design Principles, 7th edition, William Stallings.