Virtual Memory

ICS332

Operating Systems

Virtual Memory

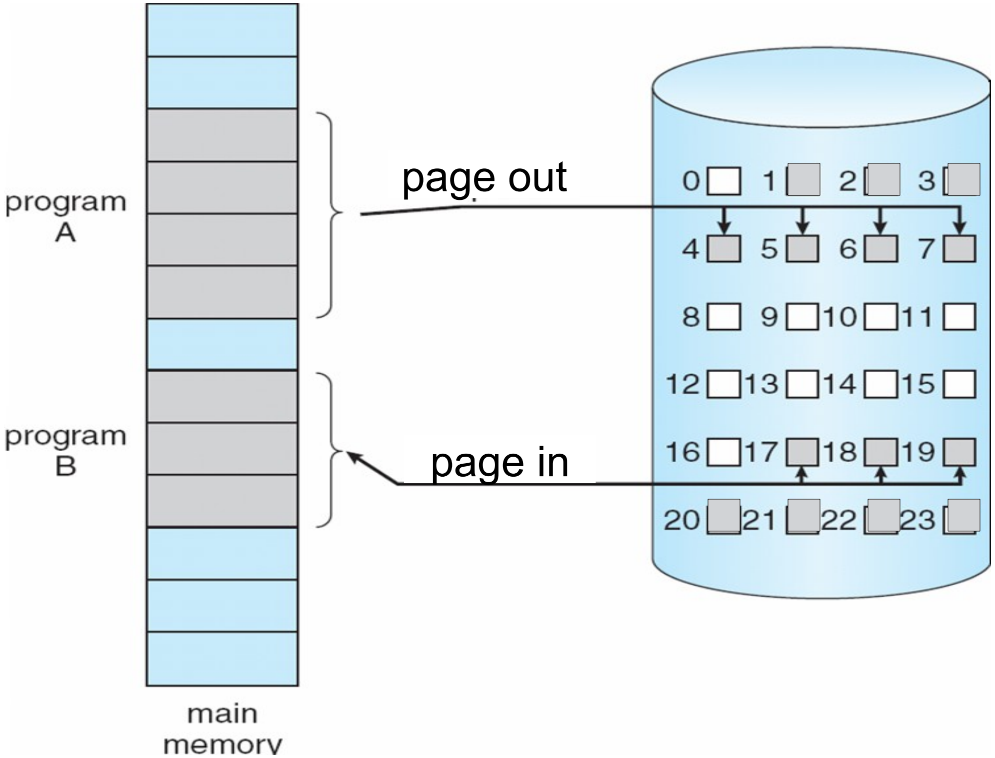
- Allow a process to execute while not completely in memory
  - Part of the address space is kept on disk
- So far, we have assumed that the full address space must be in memory for a process to execute
  - Although dynamic loading broke that assumption a little bit
- Requiring the full process in memory is overkill
  - Programs have code that’s not used often
  - Programs tend to declare more than they use
  - Not everything is needed at the same time!
- Perhaps the process’ address space is just too big
- But we want to conserve memory space anyway

Virtual Memory

- Advantages of partially in-memory processes
  - Easy of programming:
    - Users can write programs assuming a very large virtual address space
  - Better performance:
    - More processes in the ready queue at the same time
      - Better CPU utilization: good for the system
      - Lower wait times: good for users
    - Less I/O is needed to swap processes in/out when main memory is full
    - Programs can be started faster
      - Only a few pages are needed initially
    - Consider a program that fails right away: it would be really wasteful to load it entirely, then launch and abort right away

Demand Paging

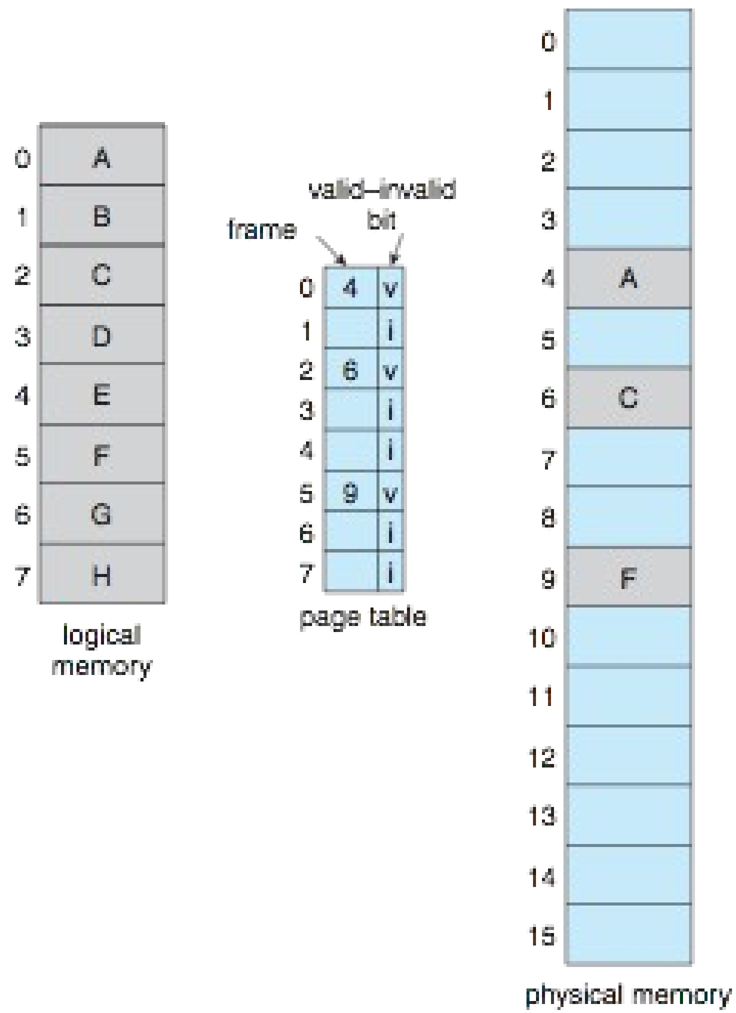
- Loading the whole process before starting it increases response time
- Demand paging: load a page only when it is needed (i.e., referenced)
  - Some pages may never be loaded!
- This is typically called a lazy scheme (as opposed to an eager scheme):



Valid/Invalid Bit

- For each process, the OS needs to keep track of which pages are in memory and which are on disk
- This is done with a valid bit in page table entries
  - a page is marked as valid if it is legal and in memory
  - a page is marked as invalid if it is illegal or on disk
- Initially the bit is set to invalid for all entries
- If the pager guesses right on which pages to bring in, the process will only reference pages with the bit set to valid
- During address translation, if the bit is invalid a trap is generated: a page fault

Valid Bit Example

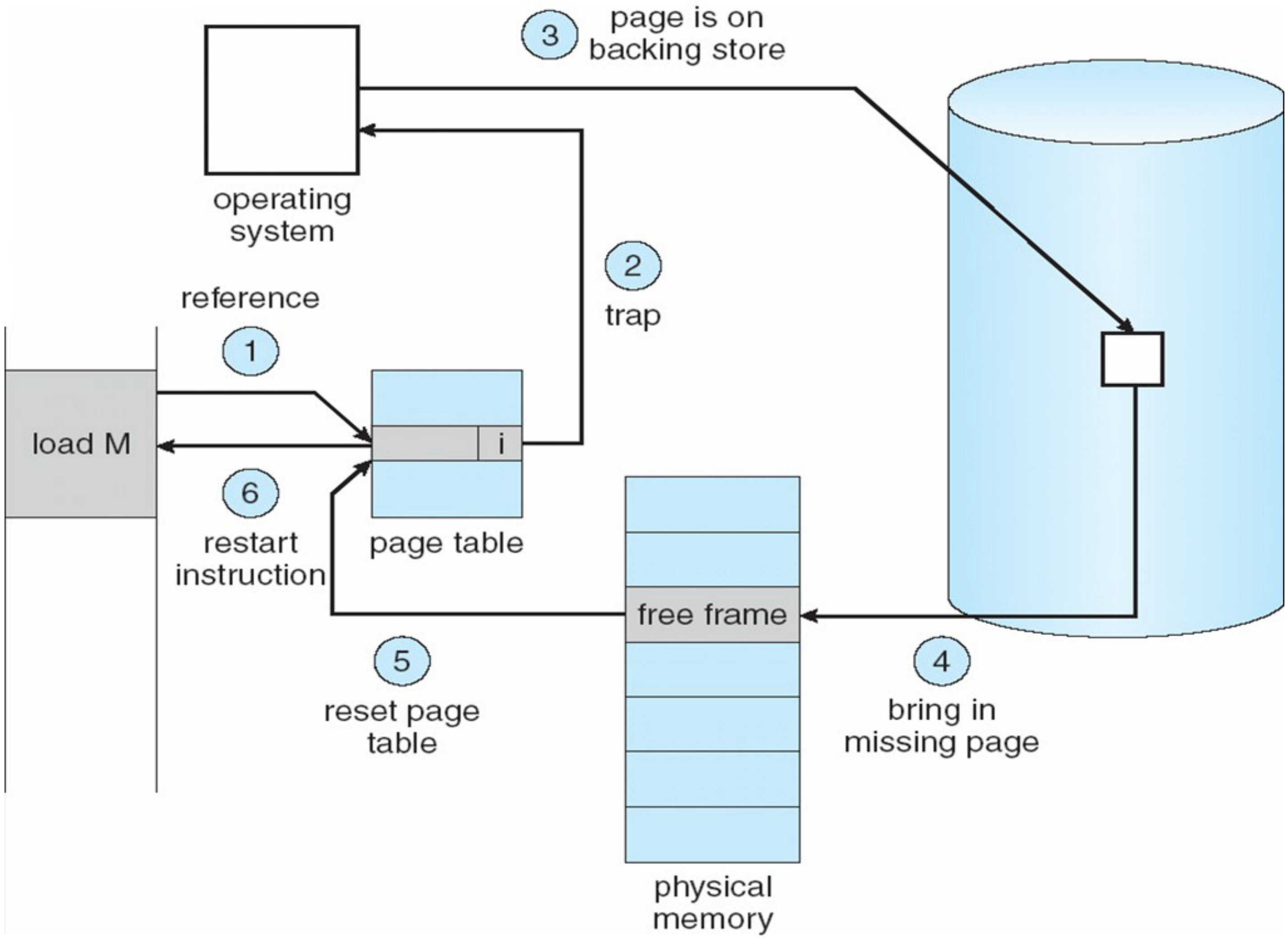


- Accessing logical page 3 (content D) would lead to a page fault
- Accessing page 2 (content C) wouldn't

Page Fault

- Upon receiving a page fault the kernel:
  - Checks whether the page is illegal or just on disk
    - The kernel keeps track of where a page is
    - If it’s illegal, then likely abort the process
  - Finds a free memory frame
    - Recall that there is a free frame list in the Kernel
  - Schedules a disk access to load the page into the frame
    - And put the process on the blocked queue of the paging manager, so that another process can run in the meantime
  - Once the disk access completes, updates the process’ page table with the new logical-physical mapping
  - Updates the valid bit of that entry
  - Restarts the process, restarting the instruction that was interrupted by the page fault in the first place

Page Fault



3
page is on backing store
operating system
2
trap
reference
1
load M
i
6
restart instruction
page table
free frame
5
reset page table
4
bring in missing page
physical memory

Restarting a Process

- Restarting a process that has page faulted can be easy
  - If the fault was on the instruction fetch, then just restart the fetch
    - Just decrement the Program Counter register by one
  - If the fault was on an operand fetch, then just restart the instruction in the same way
    - Operand will be fetched again, but oh well
  - If the fault was on result store, same idea
- Problem: instructions that modify multiple memory locations
  - e.g., an instruction that increments [eax] and decrements [ebx] and that page faults on the [ebx] access
    - Then we have to be careful not to increment [eax] twice
- Luckily we have come to love load/store architectures
  - Only two instructions access memory: load and store
  - Explicit in the ISA or in (hidden) microinstructions

Virtual Memory Performance

- Let *p* be the probability that a memory access causes a page fault
- Let *ma* be the *memory access time if no fault occurs*
  - Say 200 ns (a bit pessimistic)
- Let *penalty* be the time to resolve a page fault
- *Then we have:*
  - *from book: effective access time = (1-p)*ma + p * penalty*
  - *better as effective access time = ma + p * penalty*
- How bad is the penalty?
- The bulk of the penalty is the disk access time
  - The book makes a case for 8ms
  - Could be better due to use of swap partitions
- With these numbers: eff. access time ~ 200 + 8,000,000p
- To get performance degradation of 10%, we need p=0.0000025!!!
- Message: non-very low page fault rate = death

Fork() and Exec()

- We've seen that fork() does a copy of the address space of the parent process to create an identical child process
- Most of the time we use exec() right after fork() to run another program
  - Example: if (!fork()) { exec("/bin/ls",...); }
- Why is this a horrible waste?

Fork() and Exec()

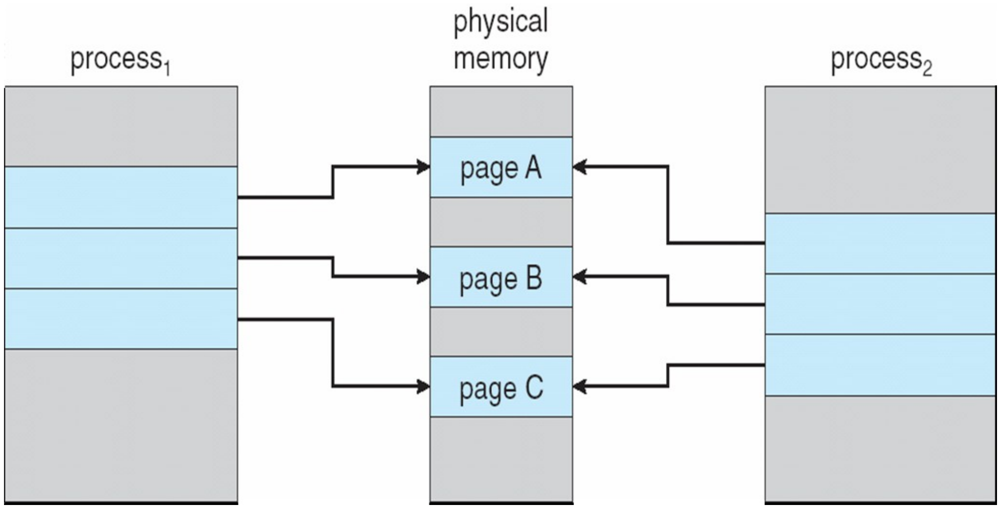
- We’ve seen that fork() does a copy of the address space of the parent process to create an identical child process
- Most of the time we use exec() right after fork() to run another program
  - Example: if (!fork()) { exec(“/bin/ls”,...); }
- Why is this a horrible waste?

- Why copy an address space to immediately overwrite it with another?? (that of “/bin/ls”)

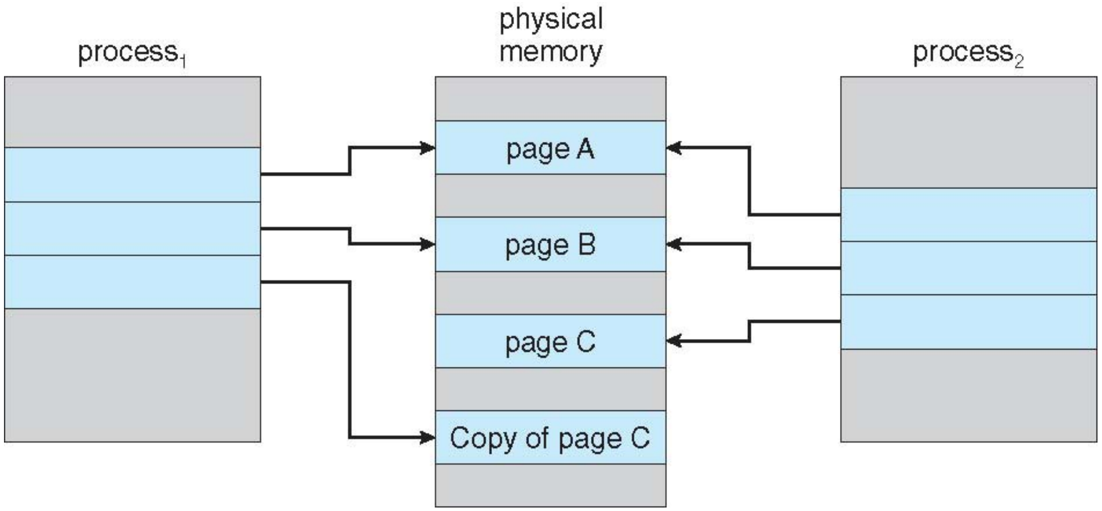
Copy-on-Write

- Process creation, i.e., fork(), can be sped up by page sharing
  - Minimize the number of new pages for the new process
- Since fork() is often followed by exec(), no need for full address space copy
- Copy-on-write
  - Parent and Child share all pages
  - All writable pages are marked as “copy-on-write”
    - e.g., the code isn’t marked as copy-on-write
  - If either process modifies a copy-on-write page, then a copy is made
- Used by WinXP, Linux, Solaris, etc.
  - Linux vfork(): parent is suspended
    - Used right before exec()

Copy-on-Write



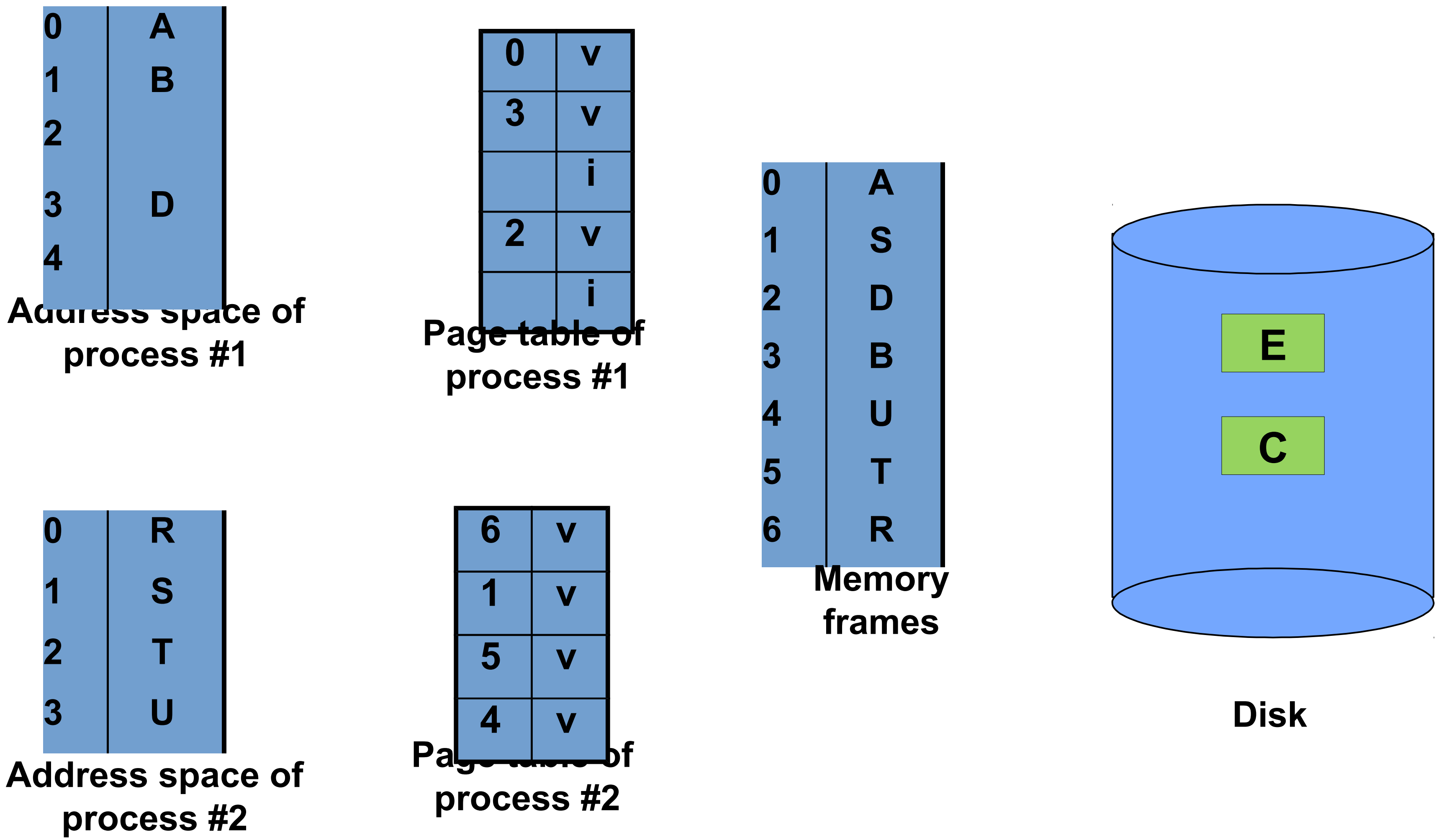
One process writes to page C



Page Replacement

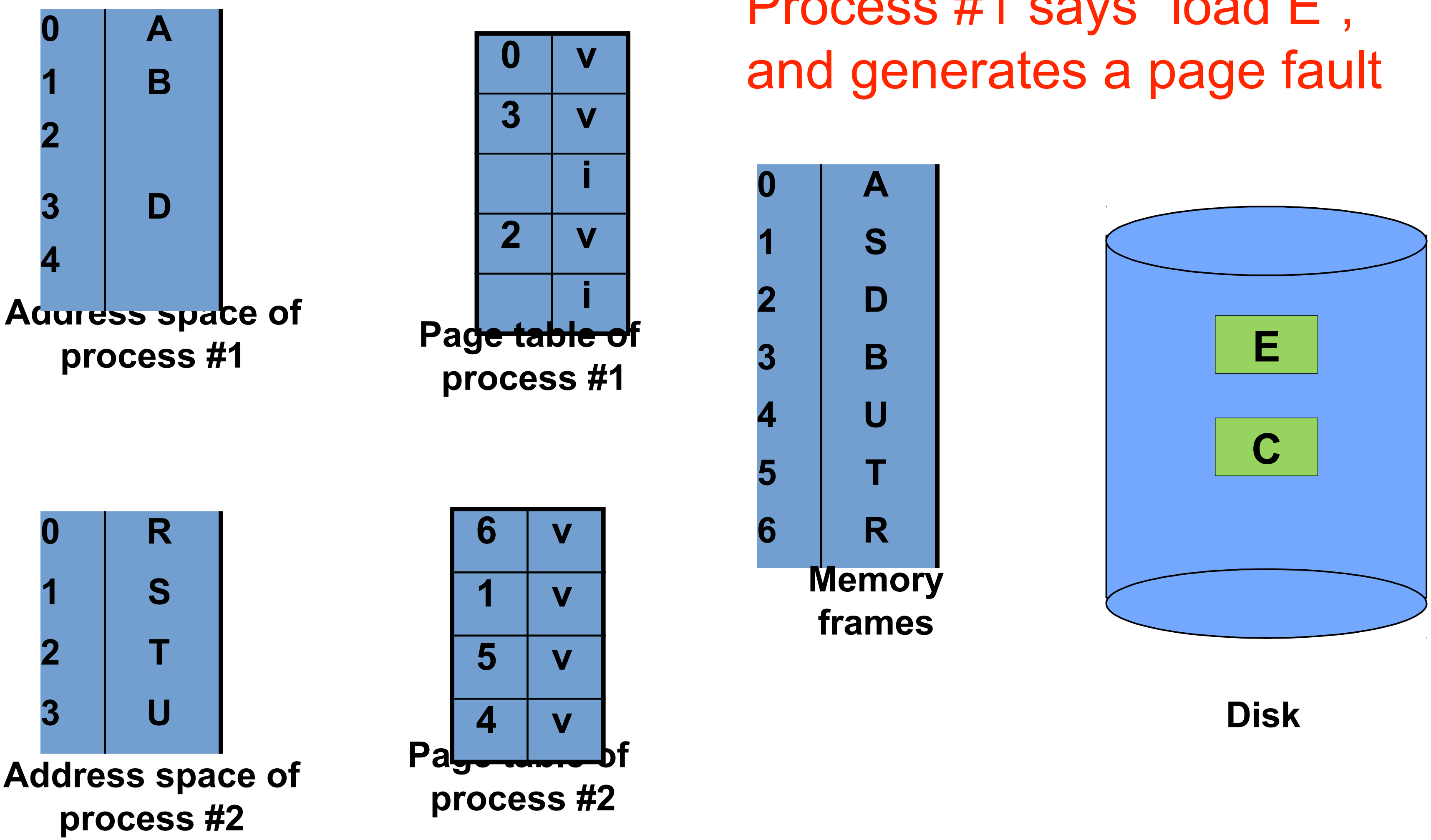
- Virtual Memory increases multi-programming and provides the illusion of large address spaces
- But it may run out of memory:
  - A page fault occurs
  - The free frame list is empty
- There is a need for page replacement
  - Evict a page from a frame (victim frame)
    - Possibly write it back to disk
  - Put the newly needed page in its place
- Page replacement may thus require two page transfers
  - When your main memory is full, and all processes are trying to access memory, things just get slow

Page Replacement



Page Replacement

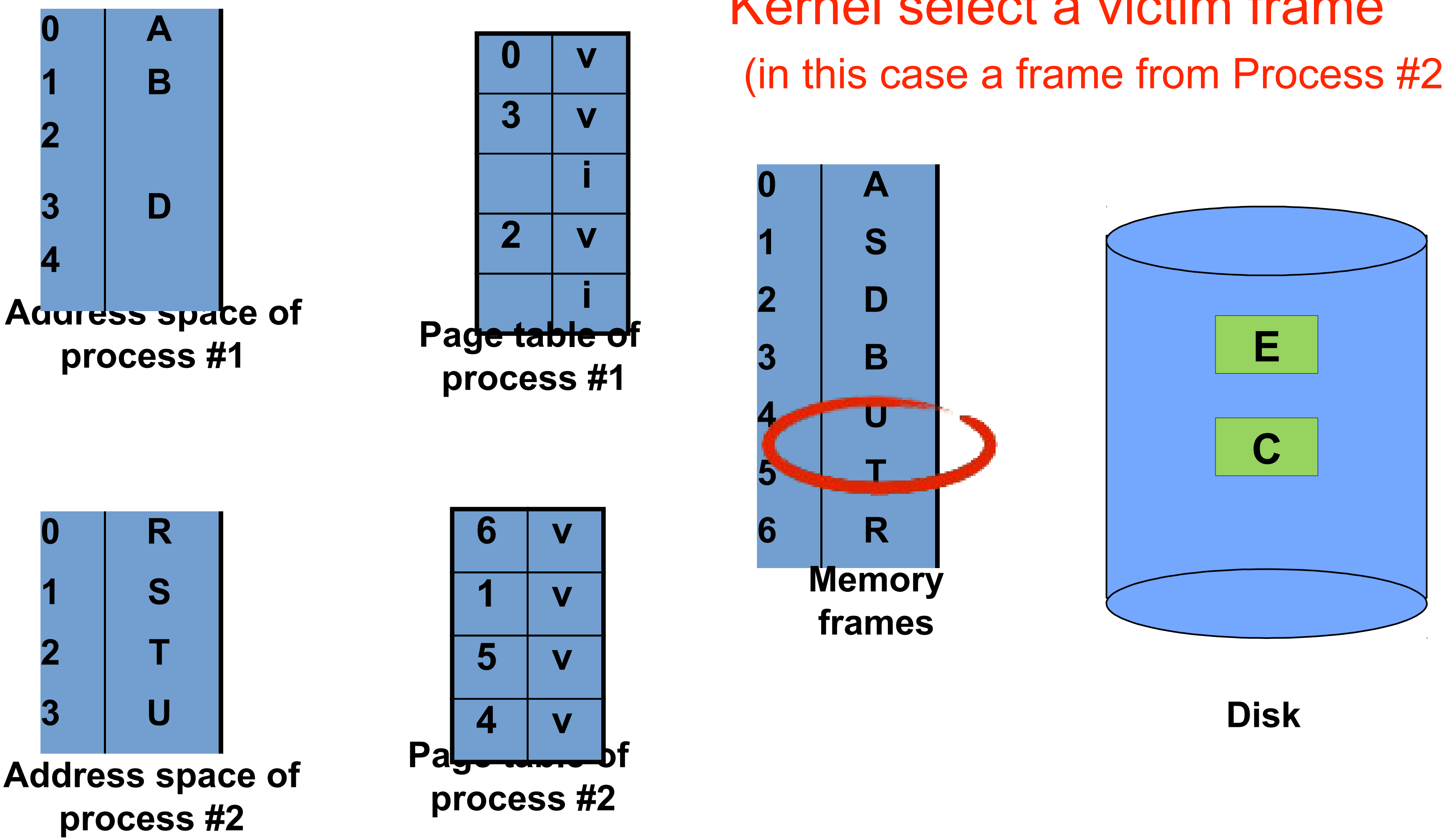
Process #1 says “load E”, and generates a page fault



Page Replacement

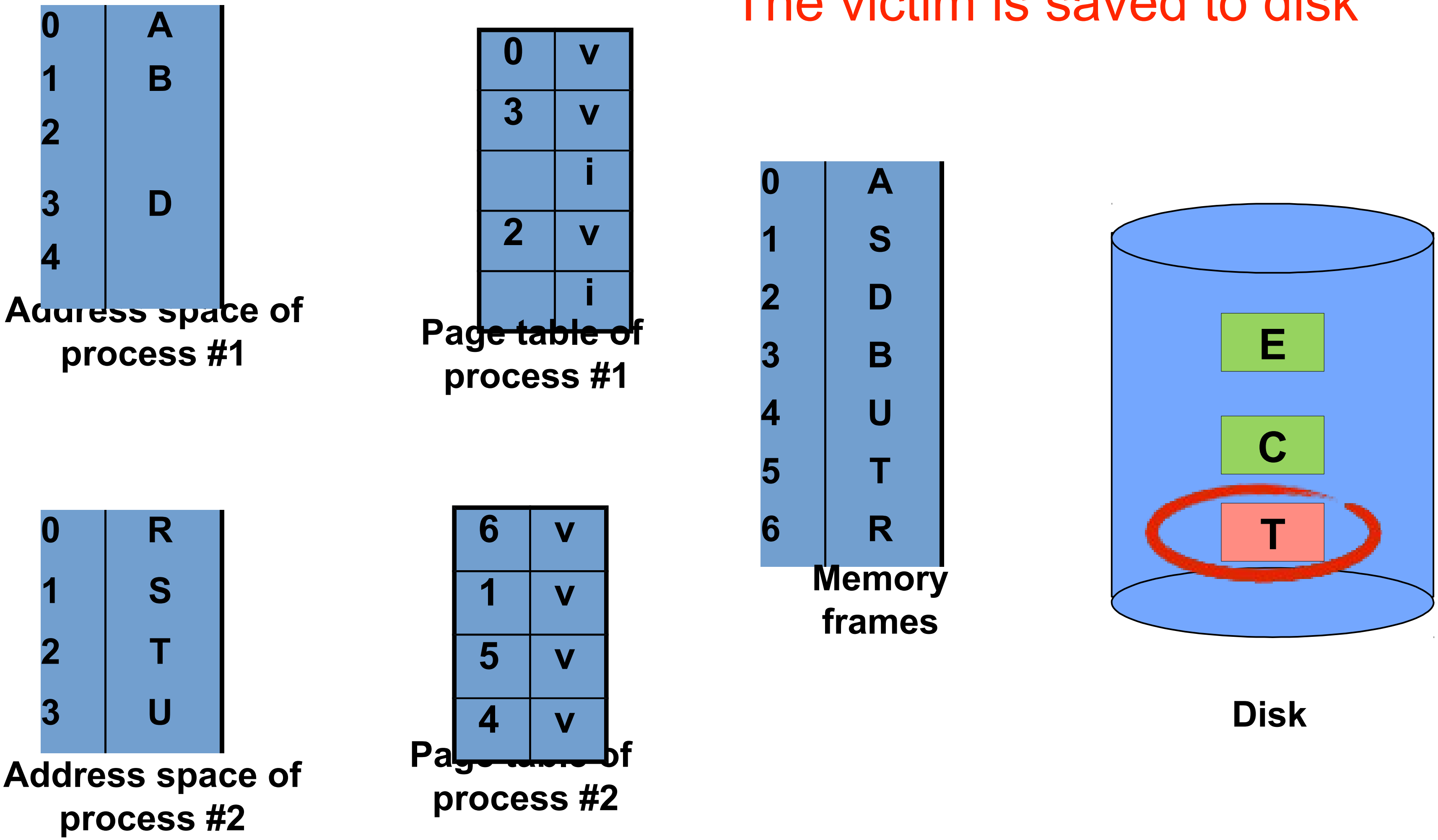
Kernel select a victim frame

(in this case a frame from Process #2)



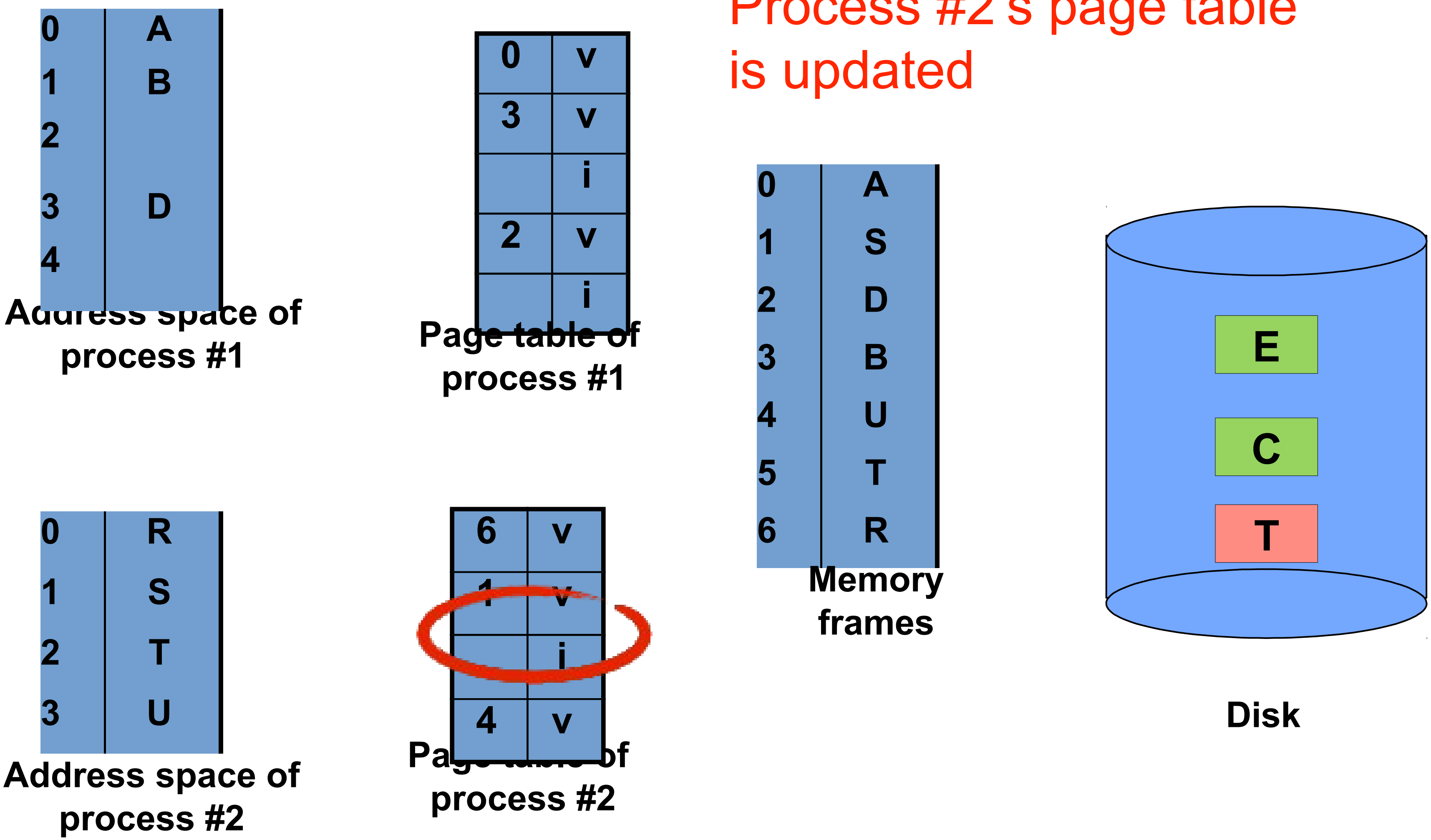
Page Replacement

The victim is saved to disk



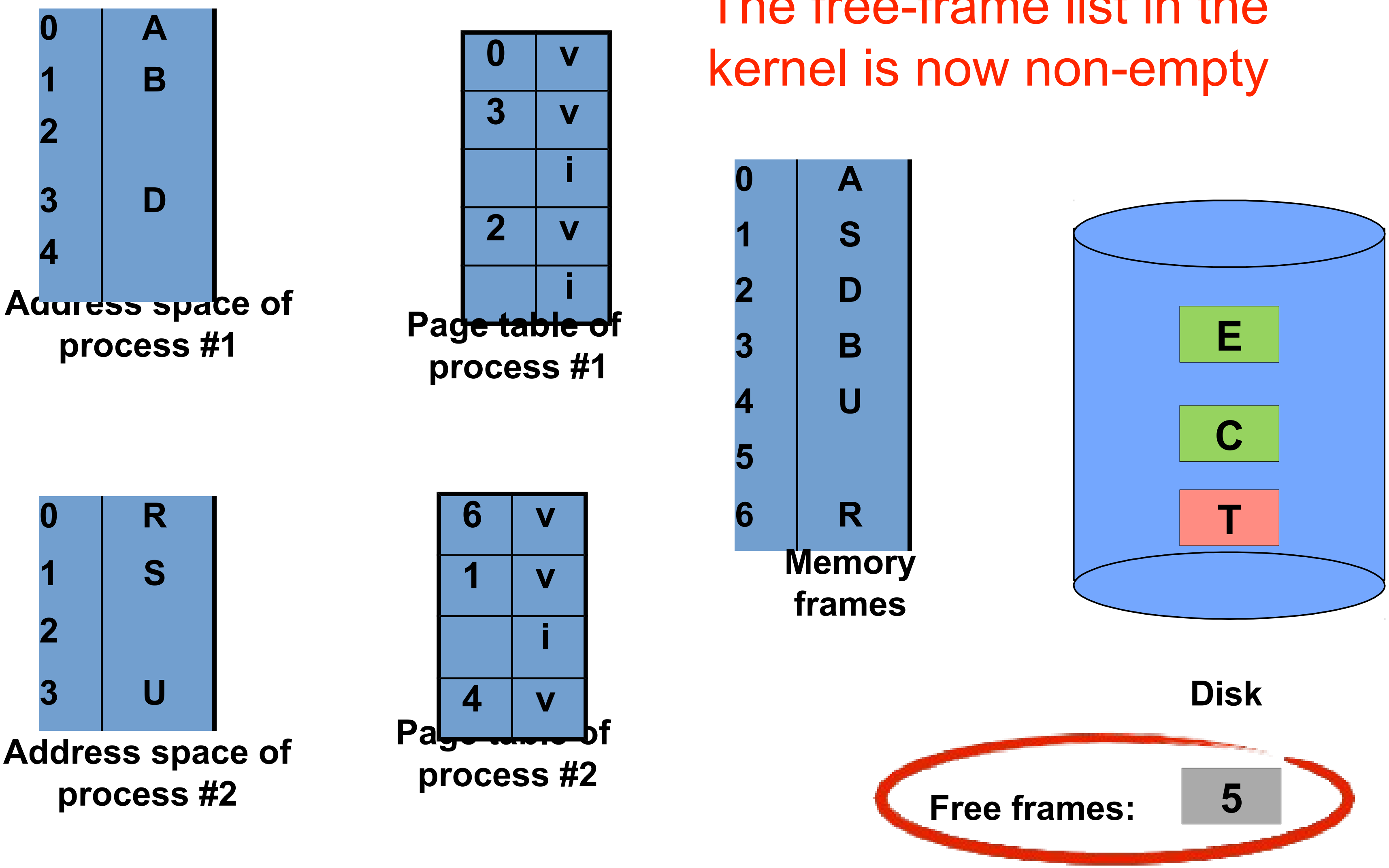
Page Replacement

Process #2's page table is updated

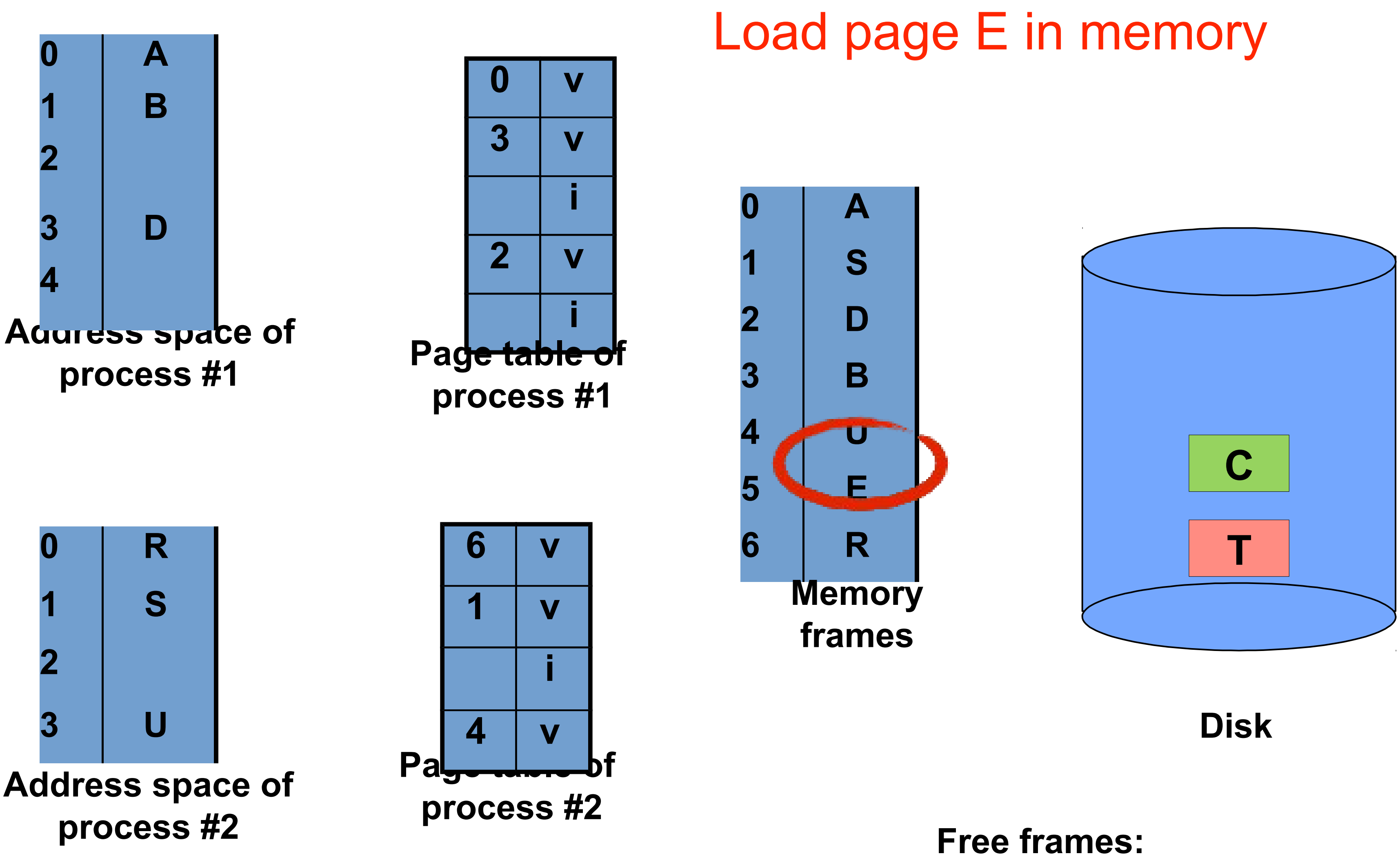


Page Replacement

The free-frame list in the kernel is now non-empty

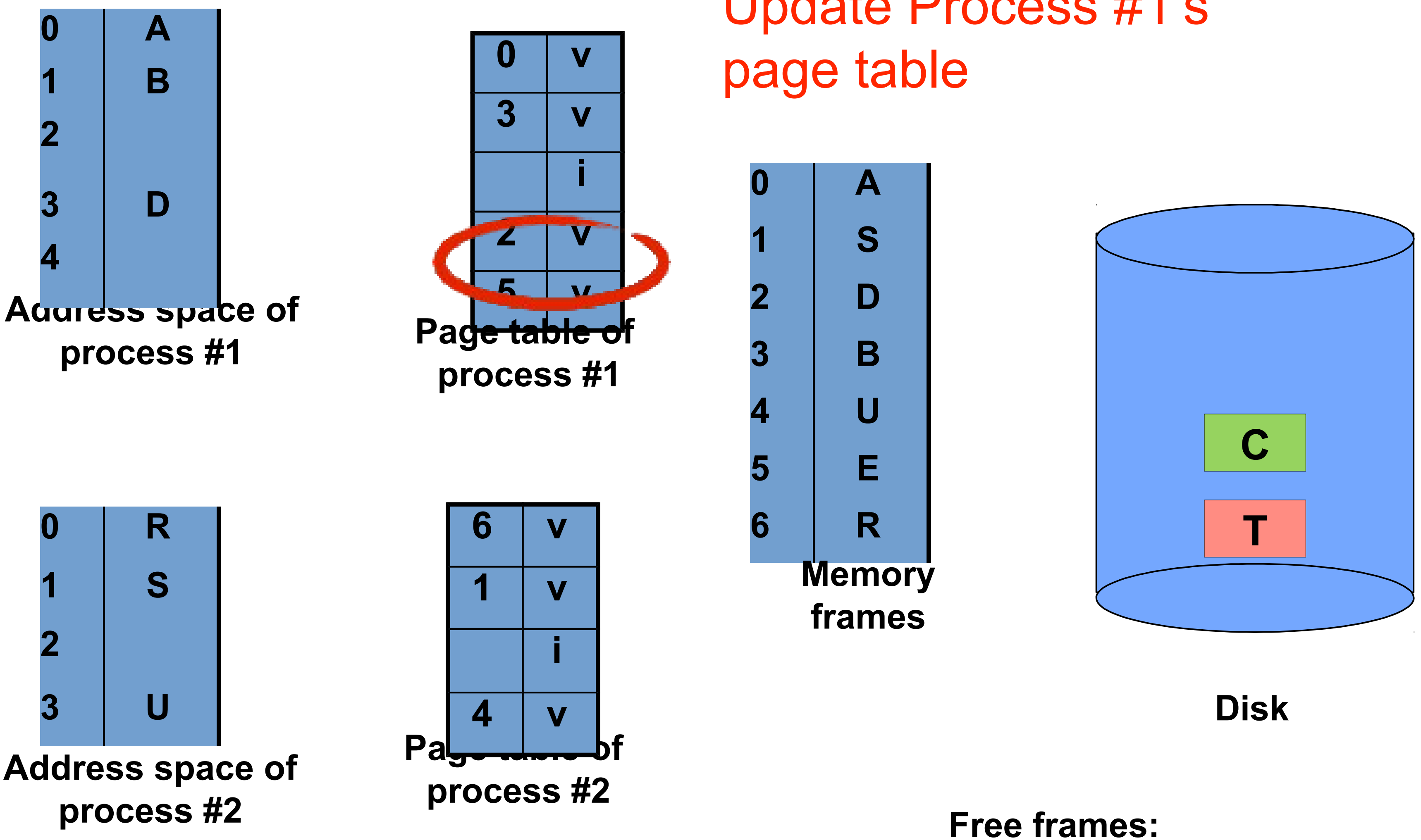


Page Replacement



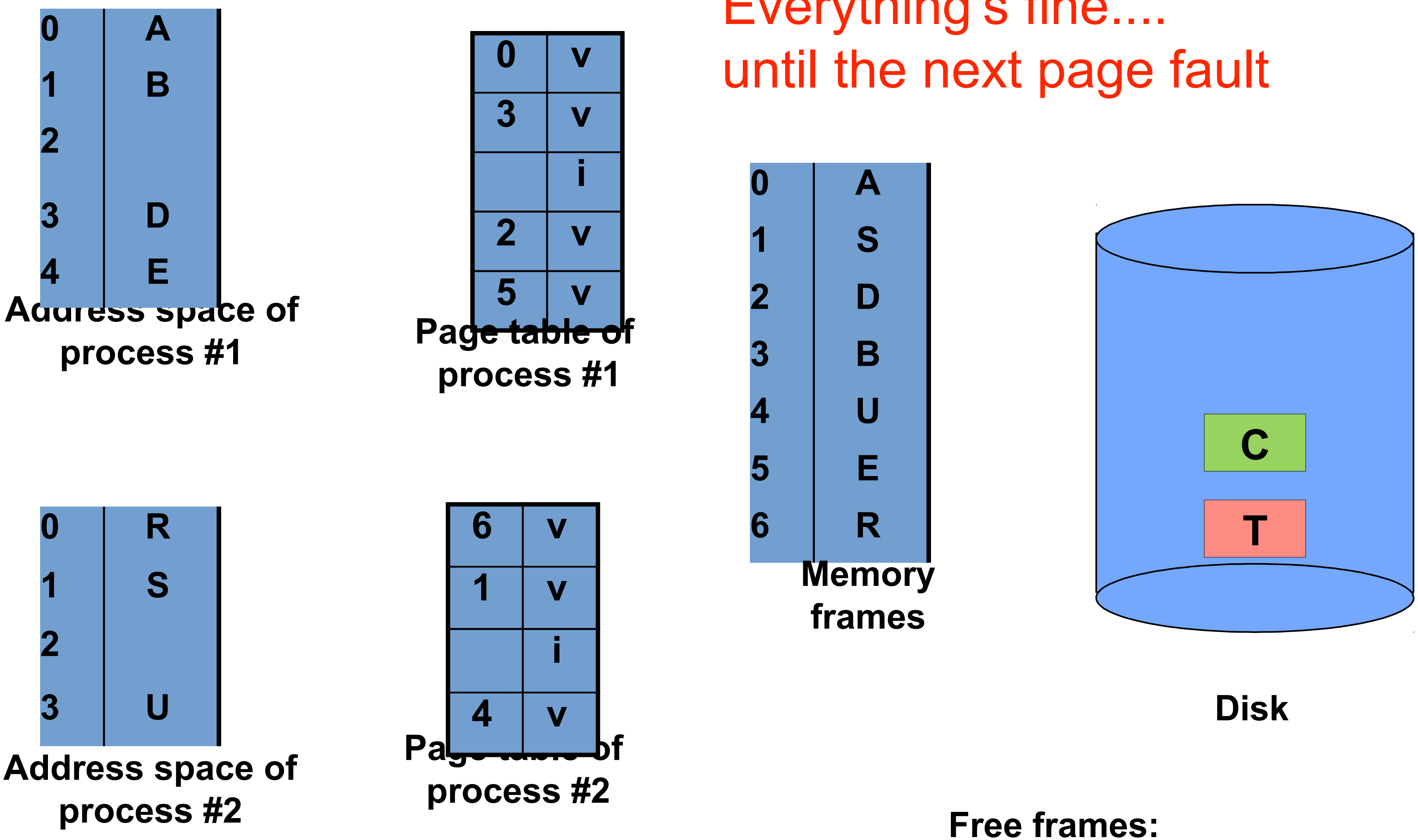
Page Replacement

Update Process #1's page table



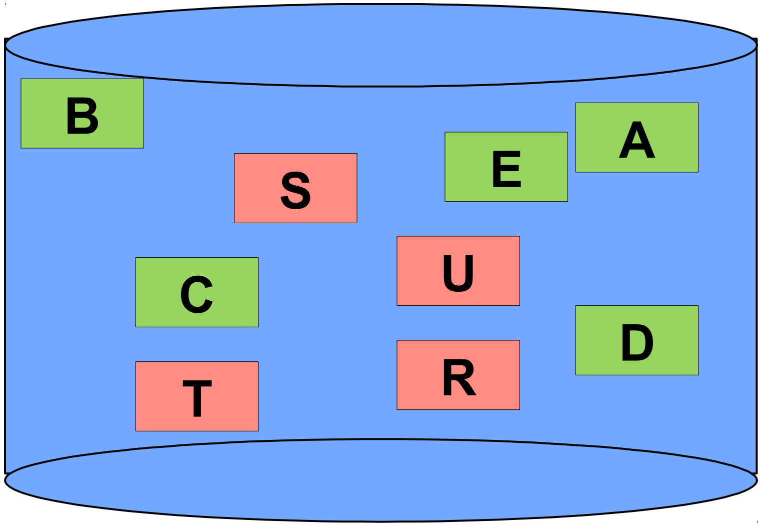
Page Replacement

Everything's fine....
until the next page fault



All pages are kept on disk

- In the previous pictures, it seems that pages are either in memory or on disk
- But pages are always on disk
  - If the system crashes, we don’t want to lose data and text segments of our executable!
- So, the disk picture should have always been:



Dirty Bit

- When writing an evicted page back to disk, it is possible that no change was ever made to that page
  - If it’s a read-only page, e.g., code or input
  - If it’s simply not been written to because the process that owns that page hasn’t gotten around to writing to it yet
- So when evicting a victim page, if it hasn’t been modified, no need to write it back to disk!
- Each frame (or page) is accompanied with a dirty bit
  - If the bit is set, the page in the frame has been modified and must be saved back to disk when evicted

Policies

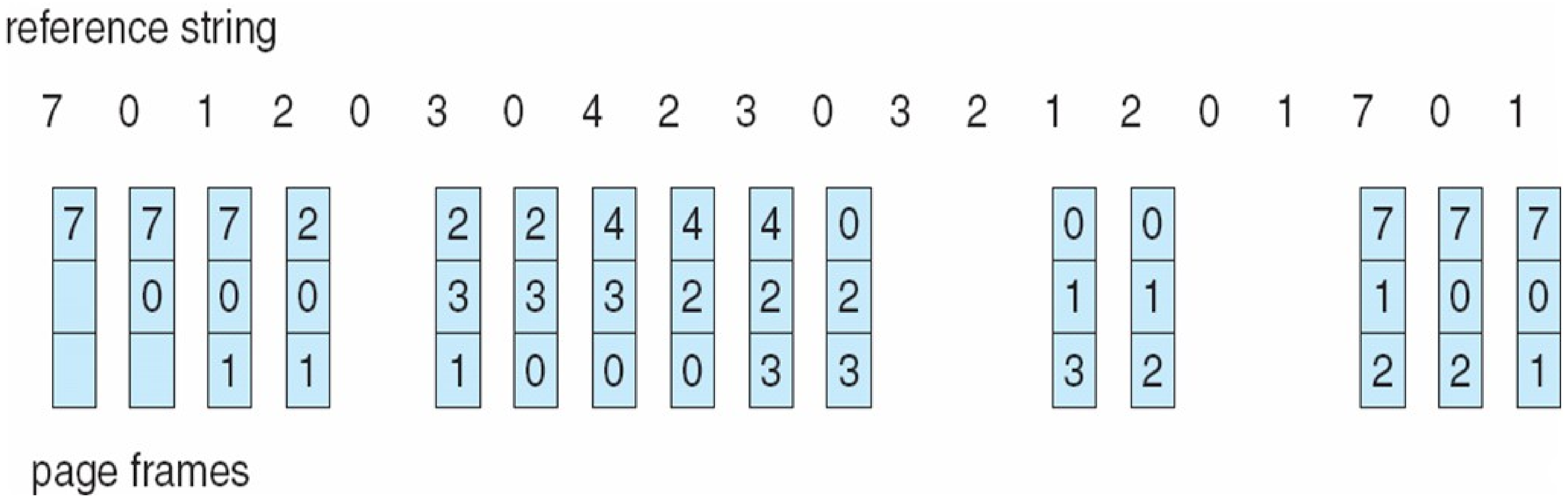
- We now have all the mechanisms, but we need to define the policies:
  - Page replacement algorithm
  - Frame allocation algorithm
- Goal: minimize the number of page faults
- Note the contrast
  - Scheduling the CPU
    - The CPU is so fast that we have to make decisions very quickly
    - We use simple algorithms that do OK, hopefully
  - Scheduling memory frames
    - The disk is so slow, that it's ok to spend some time making a decision
    - Saving only a very small fraction of the page faults leads to huge improvements
    - We can afford to use more sophisticated algorithms
    - But as usual, we work with imperfect information

Evaluating Page Replacement Algs

- Like for CPU scheduling, it's hard to tell which algorithm is good
- So we just try a bunch of cases
- A case is defined as:
  - Some number of memory frames
  - A string of page references
    - Either synthetic
    - Or collected on a real system
  - Output: count of the page faults
- Algorithms in the book are presented for 3 memory frames and the following string:
  - 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

FIFO Page Replacement

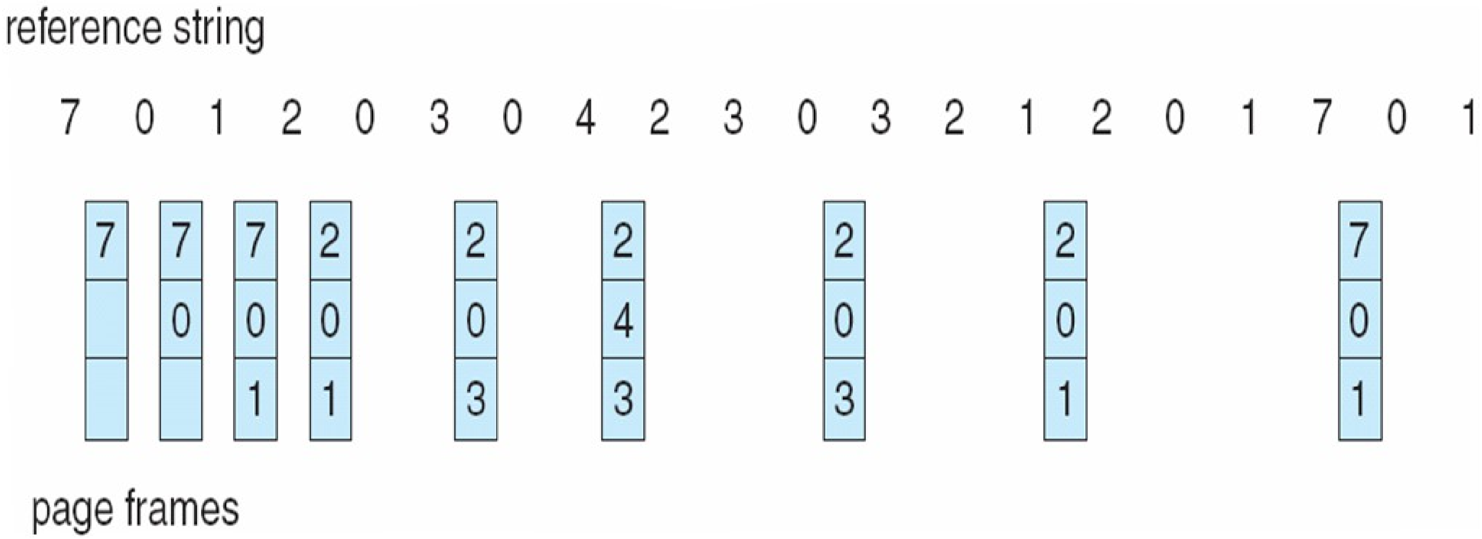
- Simplest algorithm: always evict the oldest page
  - Implemented via a FIFO queue



15 page faults

Optimal Page Replacement

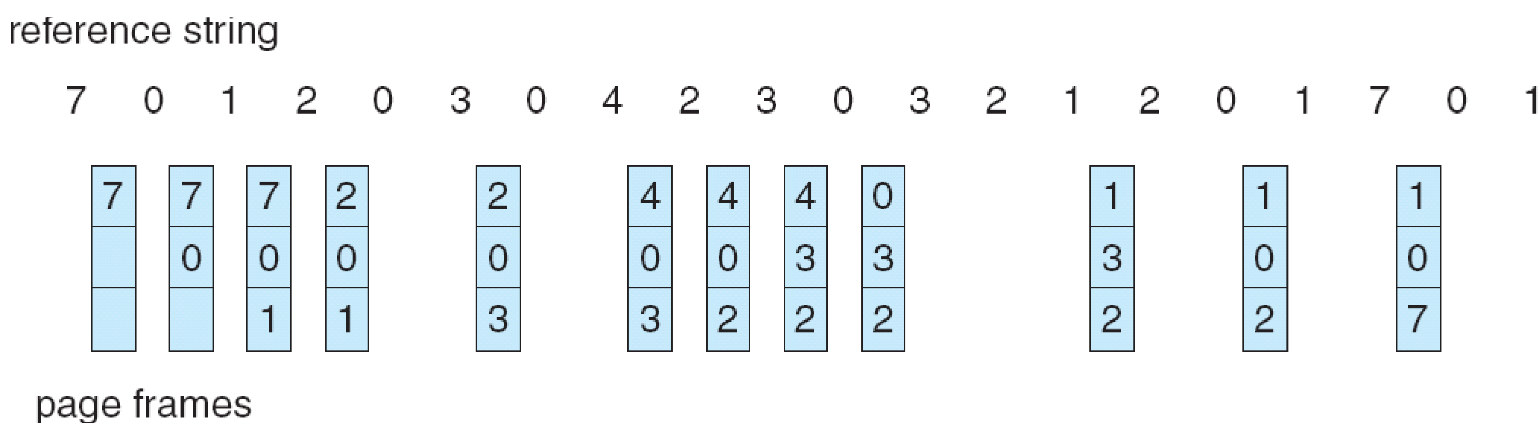
- Assuming we know the future, the best choice: evict the page that will not come in use for the longest time
  - Not possible to implement in practice
  - But good to evaluate other algorithms in absolute terms



9 page faults

LRU Page Replacement

- Least Recently Used
  - The problem with FIFO is that an old page may be used all the time
  - So it's likely better to keep track of when a page was last used



12 page faults

LRU Implementation

- LRU is considered a “good” algorithm
- Question: How to keep track of last time of use for each page?
- Answer #1: Counters
  - Augment each page table entry with a “time of use” field
  - Update that field for each memory access
  - Upon eviction search for the minimum field across the entries
  - High-overhead
- Answer #2: Stack
  - A page is moved to the top after each use
  - Requires a bunch of pointer shuffling
  - But no search for the victim (always at the bottom of the stack)
- In both cases, hardware help is needed to achieve speed

Help from the Hardware

- If the hardware doesn't provide any help, forget doing anything other than FIFO
- And the hardware doesn't typically provide enough help to implement full-fledge LRU
- Most hardware provides a reference bit
  - An additional bit to each page table entry
    - And therefore to the TLB
  - Set to 1 by the hardware when a page is accessed
- The reference bit can be used to make some (somewhat) enlightened decisions

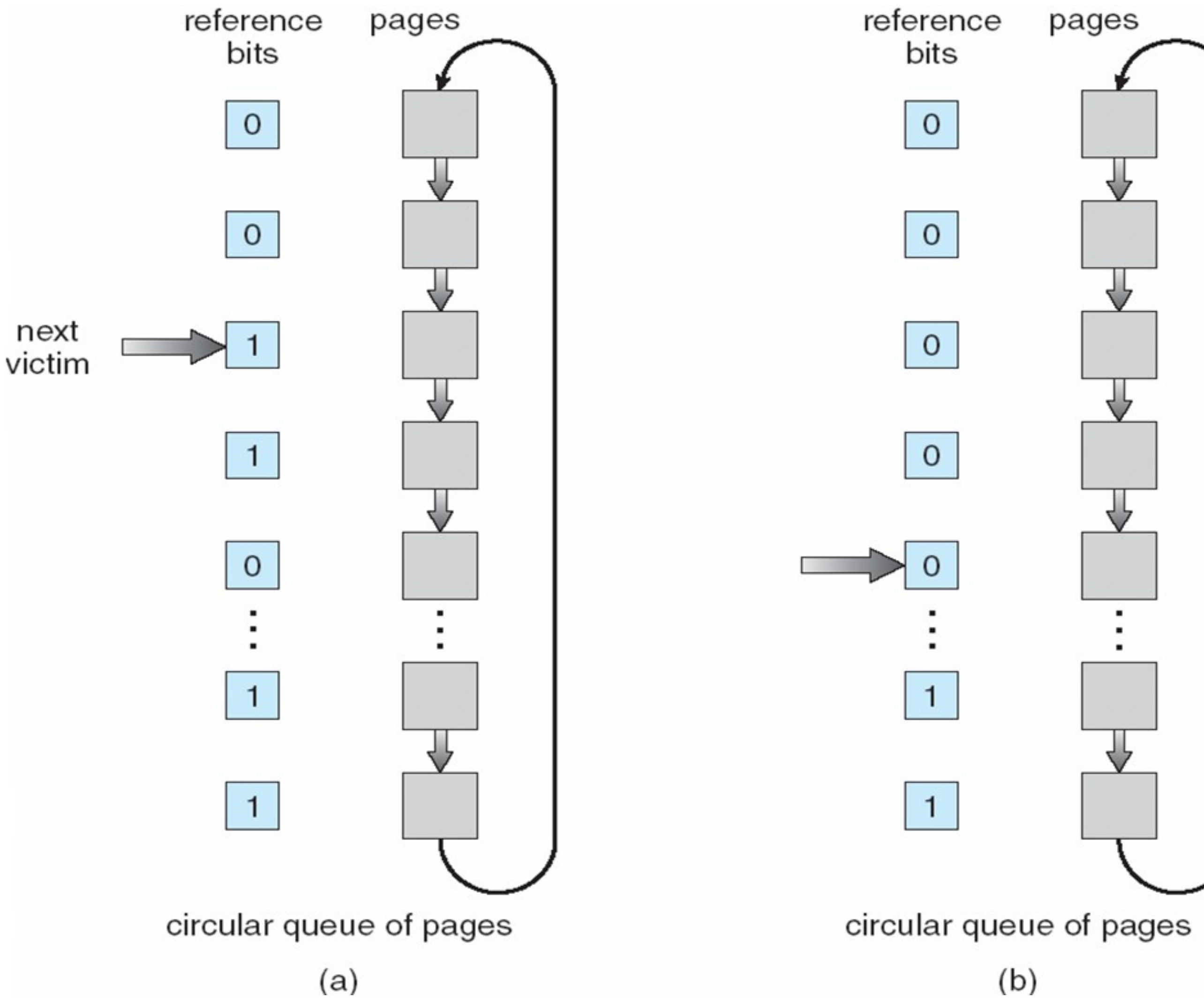
Approximate LRU

- Keep a limited history of the reference bit for each page
  - e.g., an extra N bits attached to every entry
- Update this history periodically (e.g., every 100ms) by right-shifting the reference bit into the bits of the N-bit history
- The page with the smallest history is the approximate least recently used page
- Example:
  - Page #4: 01101110
  - Page #12: 00001101 (LRU)
  - Page #13: 10100000
- Many pages can have the same history
  - Especially if N is small
- So this scheme can be used in combination with a FIFO

Second Chance

- FIFO that relies on the reference bit for history in addition to page age
- When considering the oldest page for eviction
  - If the reference bit is set to 0, evict the page
  - If the reference bit is set to 1, set it to 0, and reset the page's arrival time (i.e., age = 0)
- Result: A page that keeps getting referenced is never evicted
- Implementation technique: a circular queue

Second Chance: Figure from Book



Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms

0	0	0	0	0	0	0	0

initially no frame has been referenced

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0

After 120ms, 3 frames have been referenced

But no page fault has occurred

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0

t = 150ms page fault!

At time t=150ms we need to pick a victim

We slide the pointer to the right, zeroing out

reference bits along the way until a zero is found

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	0	1	0	0	0	0

The victim's evicted and a new page arrives (and is referenced)

The pointer advances

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0
t = 320ms	1	0	1	1	1	0	1	1

By time 320ms, no page fault has occurred and a few more frames have been references

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0
t = 320ms	1	0	1	1	1	0	1	1

Question: if we now have 2 page faults in a row, which page will be evicted?

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0
t = 320ms	1	0	1	1	1	0	1	1

Question: if we now have 2 page faults in a row, which frames are have a page fault now, which page will be evicted?

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0
t = 320ms	1	0	1	1	1	0	1	1

Question: What will the circular look like once the two page faults are resolved are where is the pointer?

Second Chance Example

- Example for 8 frames, assuming all frames are always occupied by some page

t = 0ms	0	0	0	0	0	0	0	0
t = 120ms	1	1	0	1	0	0	0	0
t = 150ms	0	0	1	1	0	0	0	0
t = 320ms	0	1	1	0	0	1	0	0

Question: What will the circular look like once the two page faults are resolved?

Performance Optimizations

- Keeping a pool of free frames
  - Load the new page into a free frame before waiting for having evicted the victim page
- Remembering ghosts of evictions past
  - Assume a pool of free frames is kept
  - These free framed are *marked* free, not wiped out
  - So if an evicted page is needed again, it may already be in a frame marked free and can be retrieved with zero cost
- Opportunistic un-dirtying
  - Whenever the disk's idle, pick a dirty page, write it out to disk, and clear its dirty bit
    - We like clean pages because we can evict them "for free"

Frame Allocation Algorithms

- Question: how many frames to give to which processes?
  - e.g., we have 47 free frames in total, we have 2 new processes, how many do we give to each?
- Minimum number of frames (to execute any instruction)
  - Depends on the architecture
  - If an instruction is longer than a word, then it may straddle two frames
  - If an instruction allows both memory access and memory indirection, then we need at least three frames
    - One for the instruction
    - One for the address access
    - One for the data access
  - If the degree of indirection is unbounded, then in the worst case one needs the whole address space in frames
    - e.g., mov eax, ((((((((((ebx))))))))))
    - Unlikely in a real-world ISA
  - For a load/store architecture with word-size instructions: 2 frames
- Maximum number of frames: size of physical memory

Frame Allocation

- Equal allocation
  - m frames, n processes
  - each process gets m/n frames
- Proportional allocation
  - if s_i is the memory size of process p_i
  - if S is the sum of all process sizes
  - each process gets $(s_i/S)*m$ frames
- Priority allocation
  - bias the above to include process priority

NUMA Systems

- Non Uniform Memory Access
  - A multi-CPU system can have multiple boards, each with a CPU and memory
  - A CPU can access the memory on its board faster than that on other boards
- The paging system for a NUMA machine should try to keep pages close to processors
- Things at that point get pretty complicated
  - Especially throwing in threads

Global/Local Page Replacement

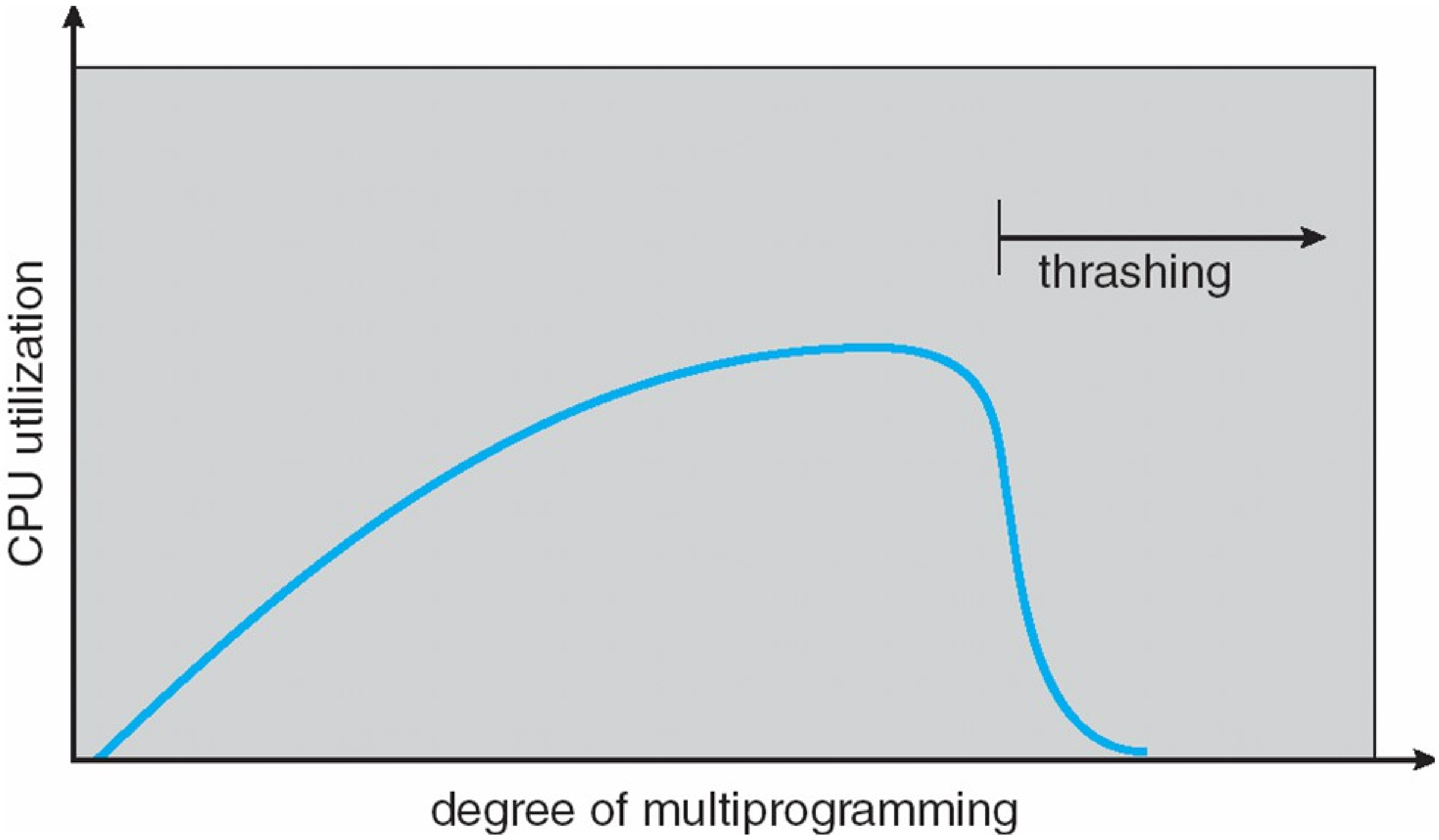
- Local replacement: victim among the page-faulting process' pages
  - Number of frames per process is kept constant
- Global replacement: Any victim can be selected
  - Could be good for high-priority processes
  - But then the page-fault performance of a process depends on other processes and may change from one run to the next
- Global replacement is typically used because it increases system throughput
  - Let processes grab frames when they need them where they can find them as opposed to everybody in their own space
- Our example a few slides ago assumed global replacement

Thrashing

- Let’s consider a system with a global page replacement algorithm
- A process needs more frames and increases its page-fault rate
- It takes frames away from other processes
- These processes now do more page-faults
- As a result the ready queue empties out
- CPU utilization decreases as processes are waiting for the disk
- The CPU scheduler starts a new process to increase utilization
- This process needs frames and joins the “waiting for pages” group
- Another process gets brought in to increase utilization
- No work gets done: everybody’s waiting for pages
- This is called thrashing

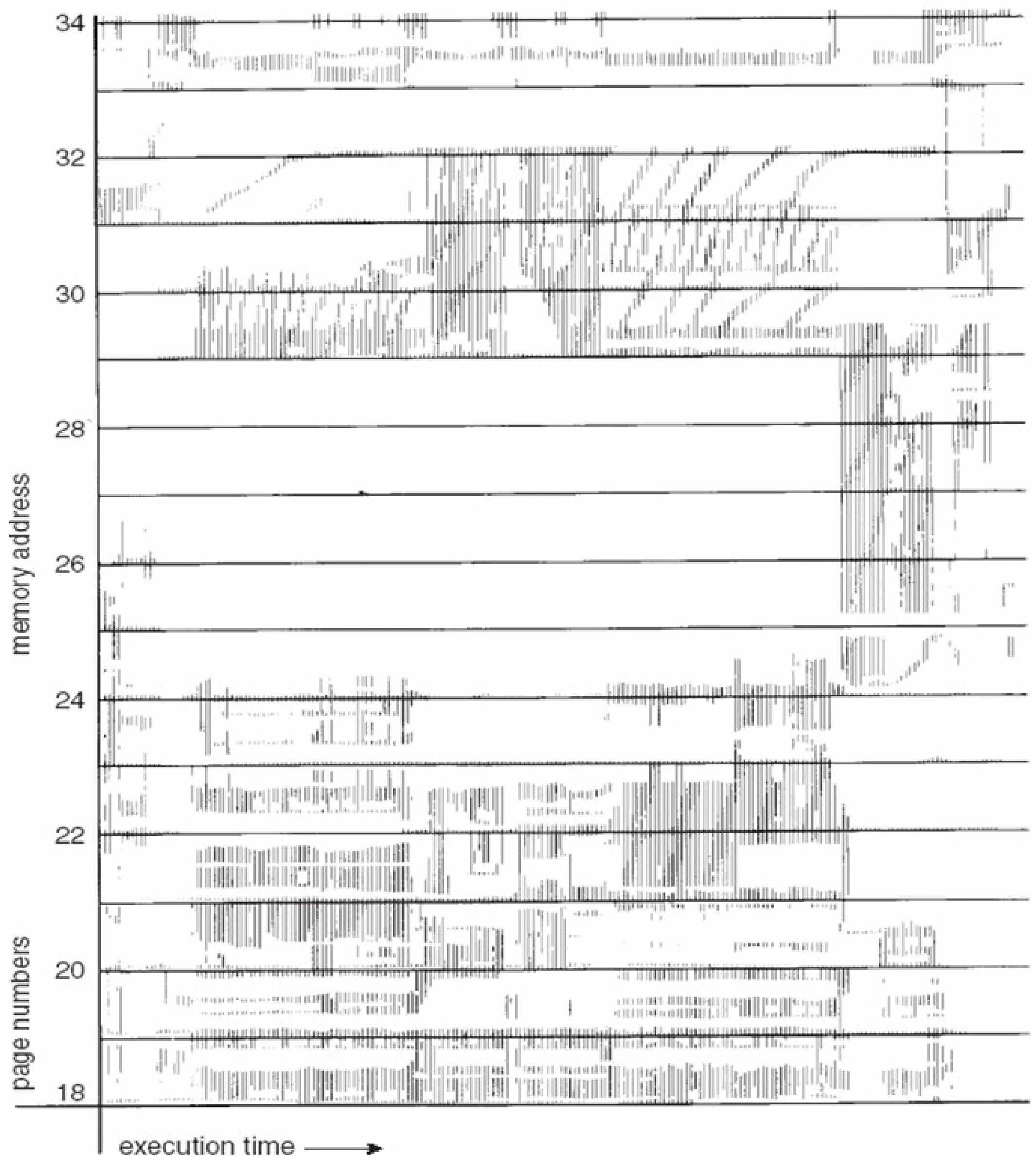
- Paradox: to increase CPU utilization we must reduce the multiprogramming level

Thrashing



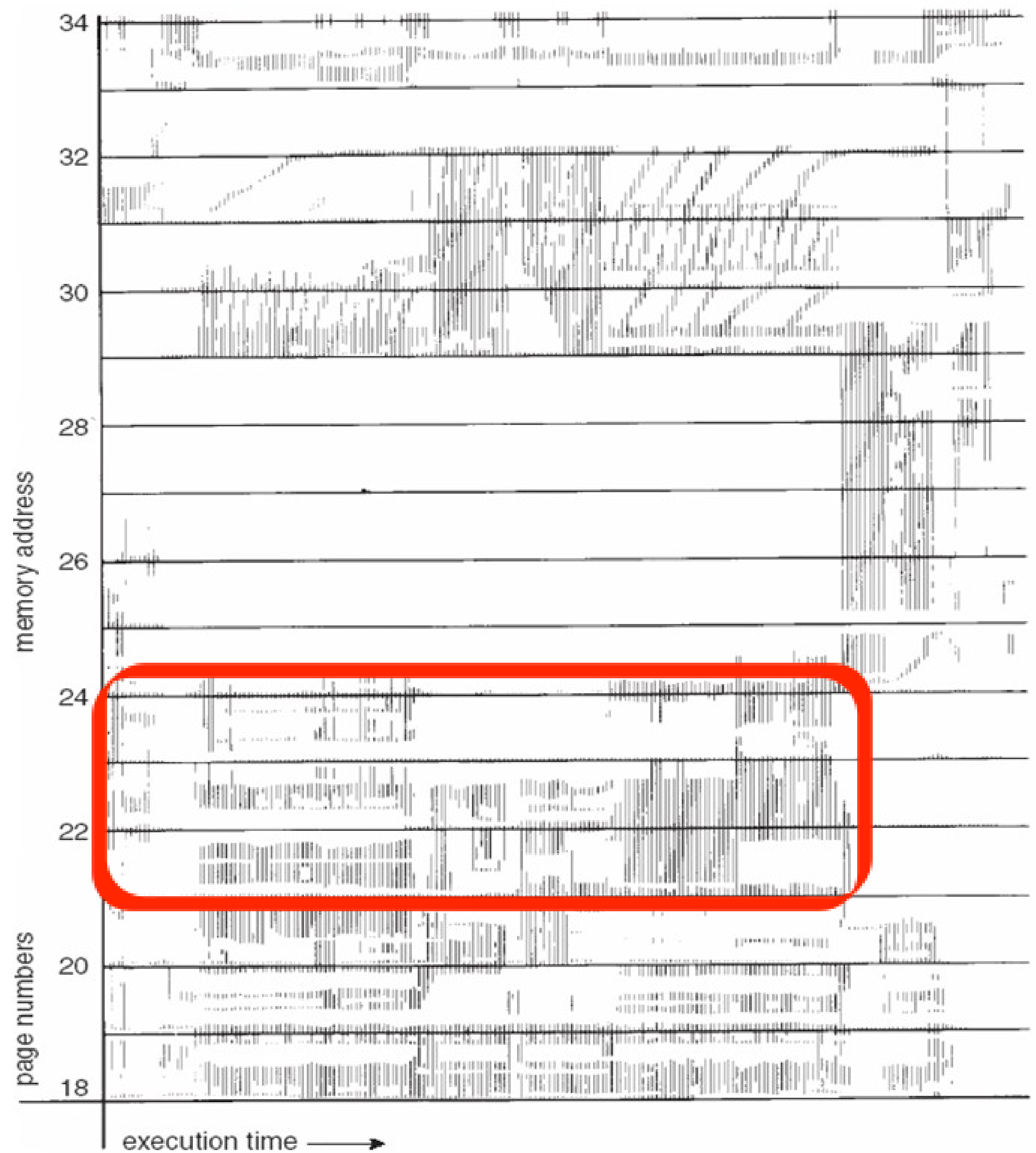
Locality

- The way to prevent thrashing is to provide each process with the pages it needs
  - easy, right?
- Problem: how do we know how many pages a process needs?
- Locality: a process tends to access pages in the same area of the address space for a while before moving to another area



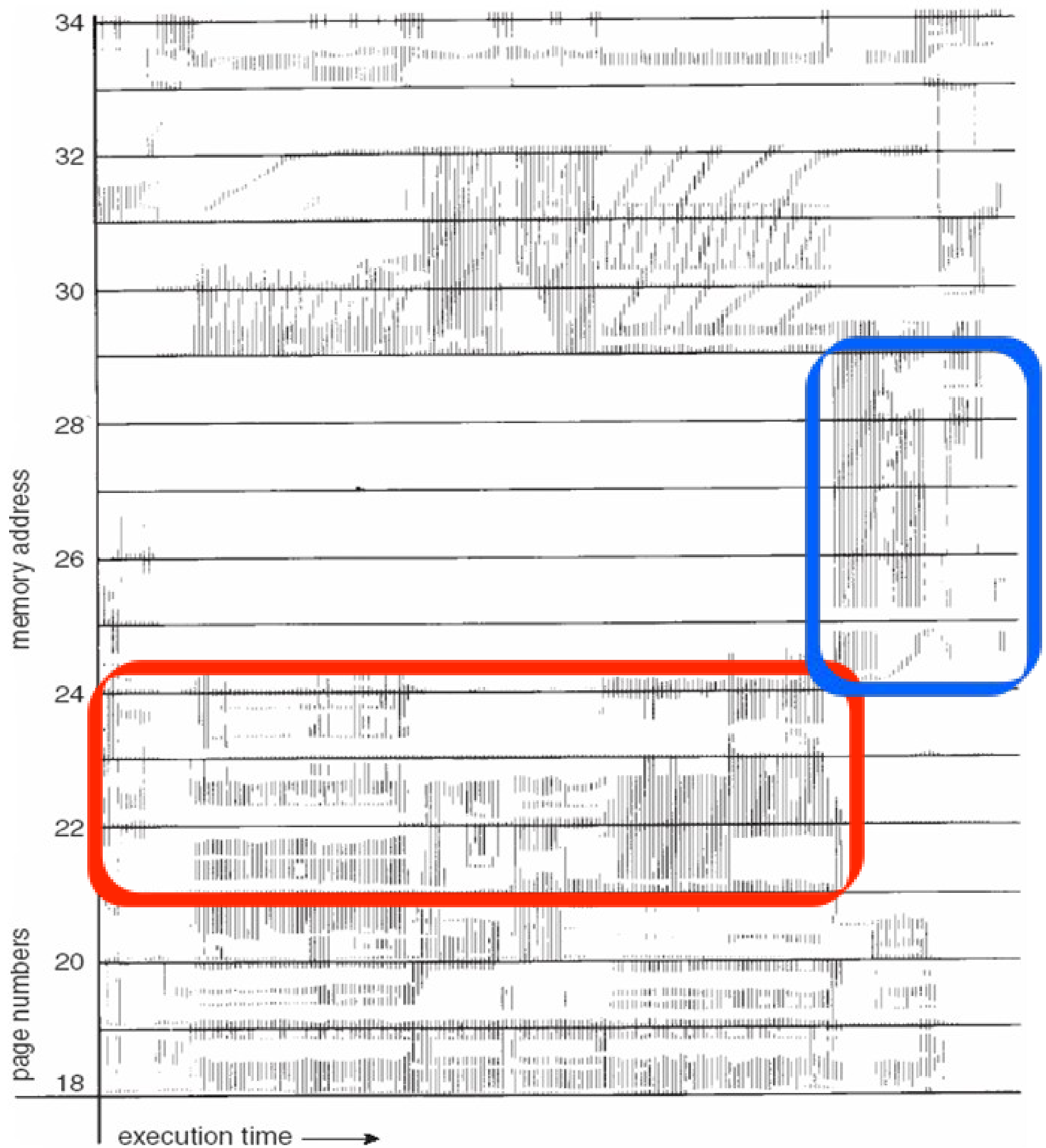
Locality

- The way to prevent thrashing is to provide each process with the pages it needs
  - easy, right?
- Problem: how do we know how many pages a process needs?
- Locality: a process tends to access pages in the same area of the address space for a while before moving to another area



Locality

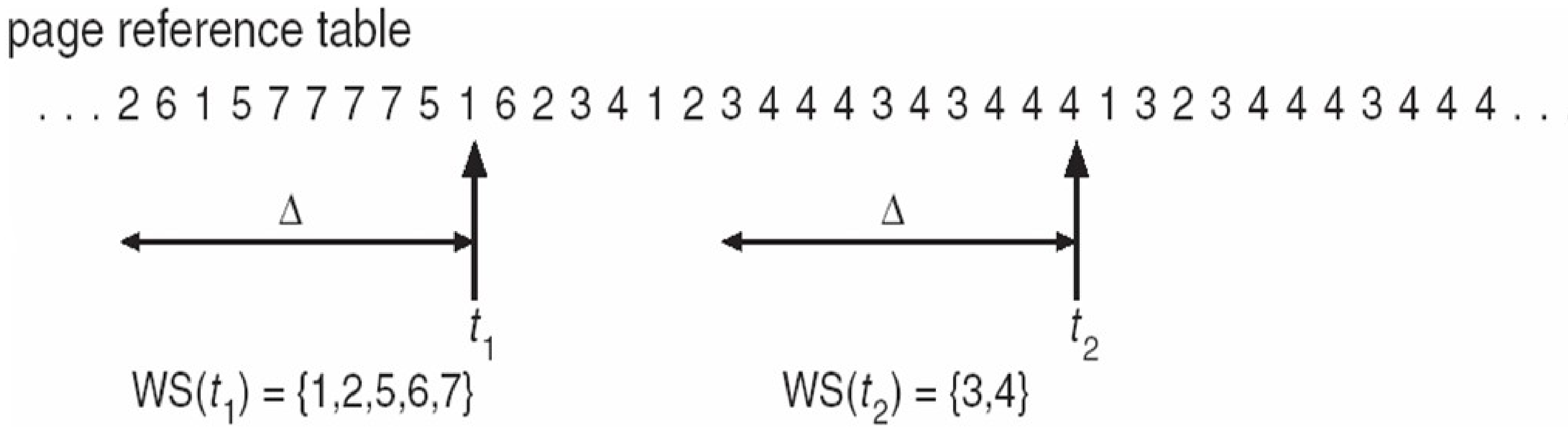
- The way to prevent thrashing is to provide each process with the pages it needs
  - easy, right?
- Problem: how do we know how many pages a process needs?
- Locality: a process tends to access pages in the same area of the address space for a while before moving to another area



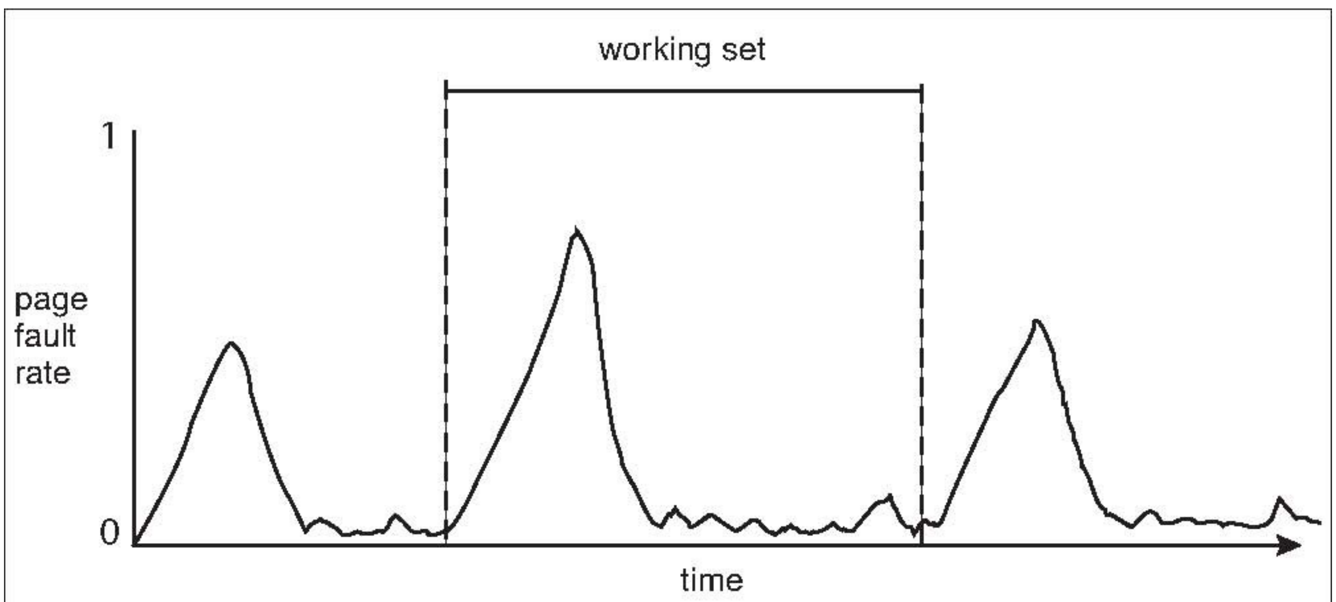
Working Set Strategy

- We can keep track of all the pages referenced by each process during a window of the last Δ memory references
- We call this the working set of the process
- The system keeps track of D, the sum of the sizes of the working set of running processes
- The system swaps out an entire process when D is larger than the number of available memory frames
- As a result, no thrashing happens

Working Set

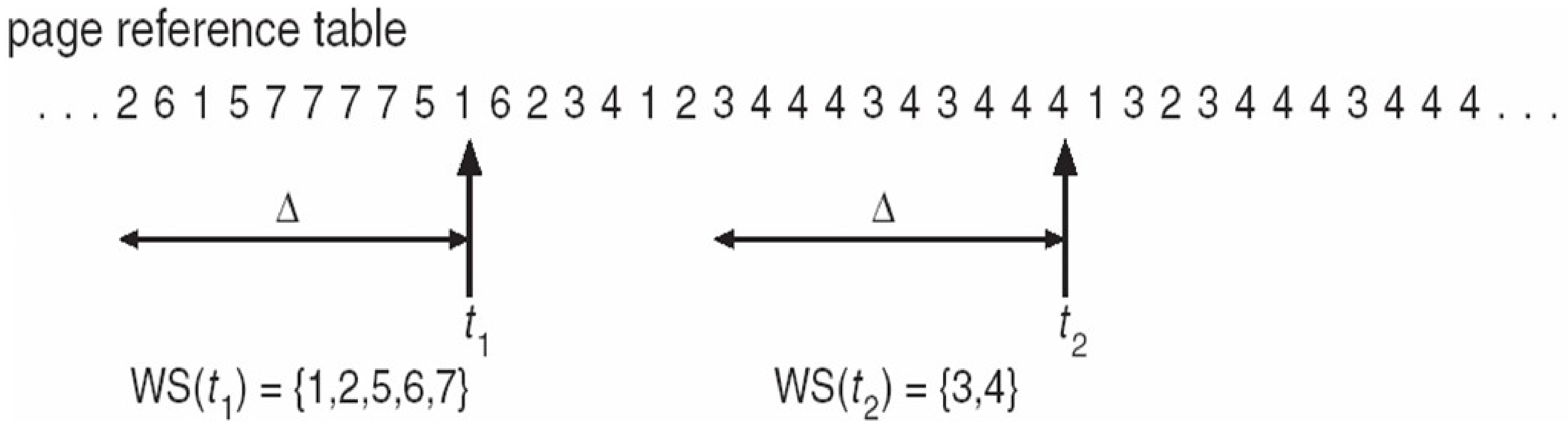


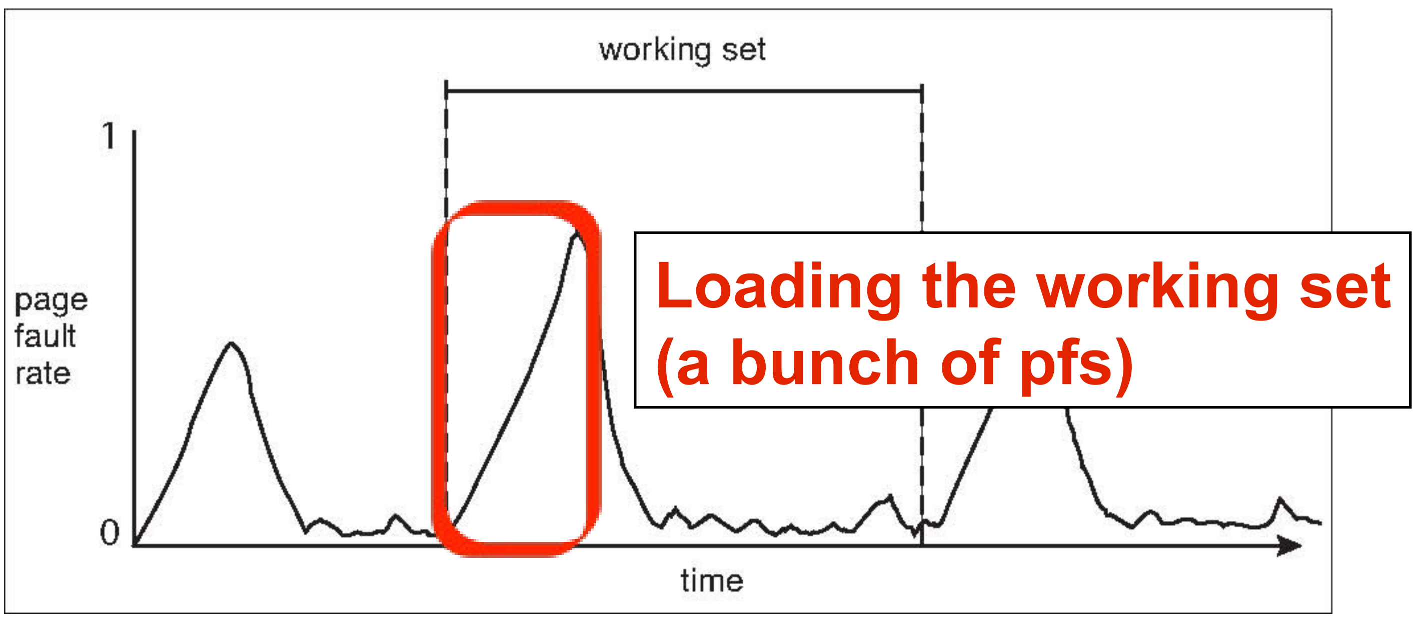
page reference table
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .
Δ
Δ
t_1
t_2
$WS(t_1) = \{1,2,5,6,7\}$
$WS(t_2) = \{3,4\}$



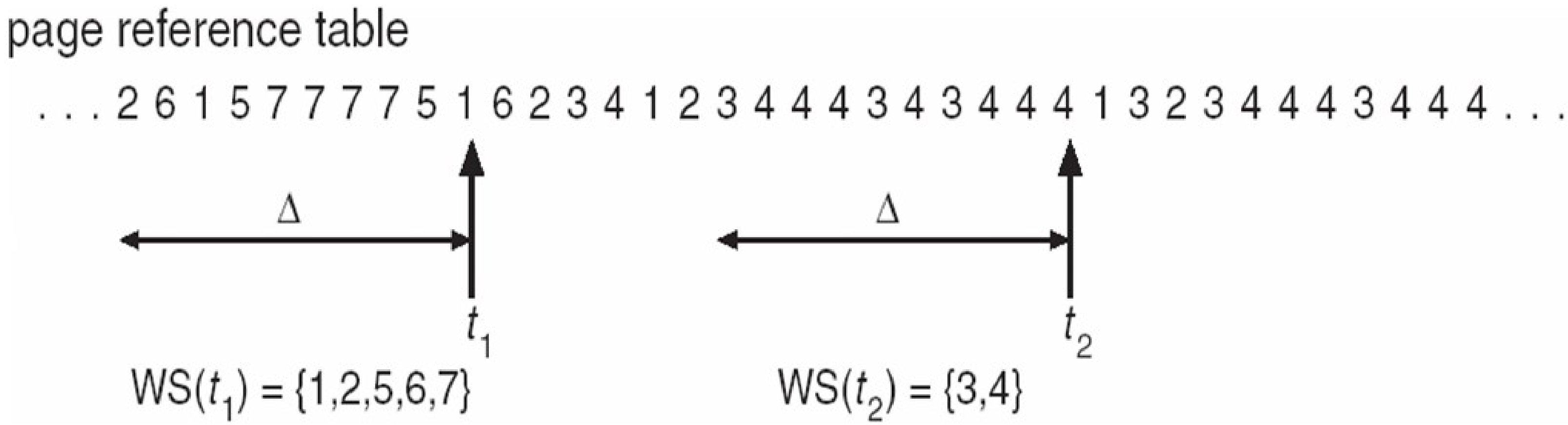
working set
1
0
page
fault
rate
time

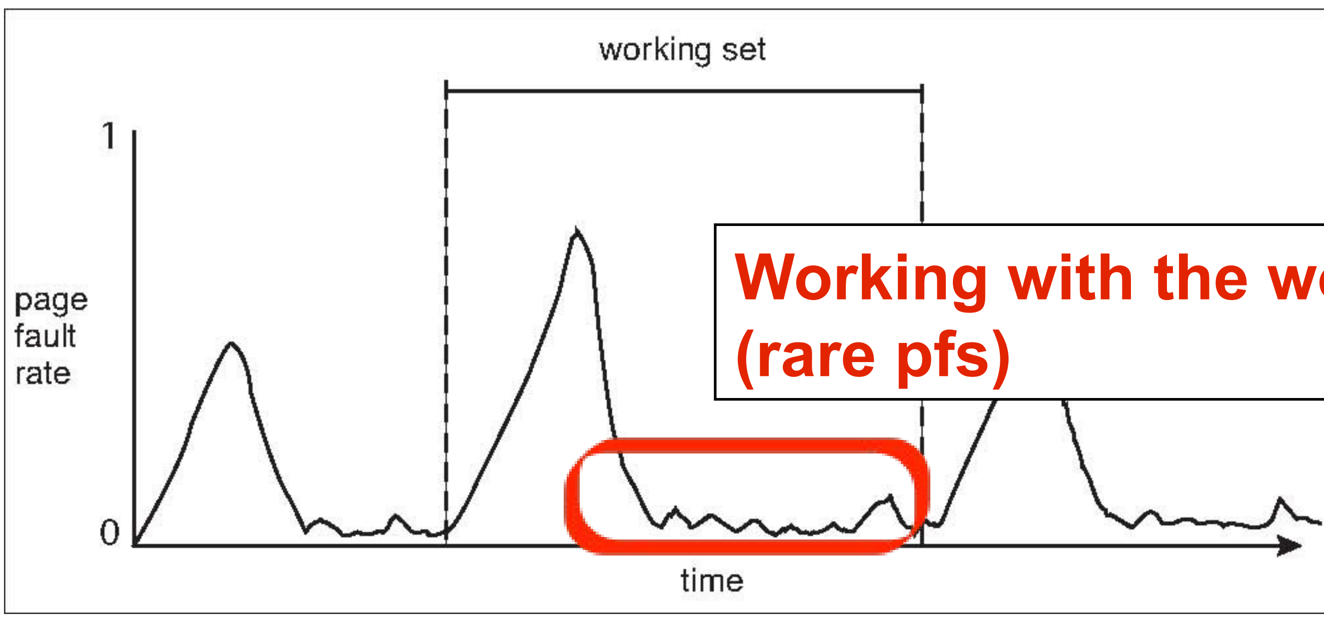
Working Set


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 . . .
Δ
Δ
t_1
t_2
$WS(t_1) = \{1,2,5,6,7\}$
$WS(t_2) = \{3,4\}$


working set
1
0
page fault rate
time
Loading the working set
(a bunch of pfs)

Working Set





Working with the working set (rare pfs)

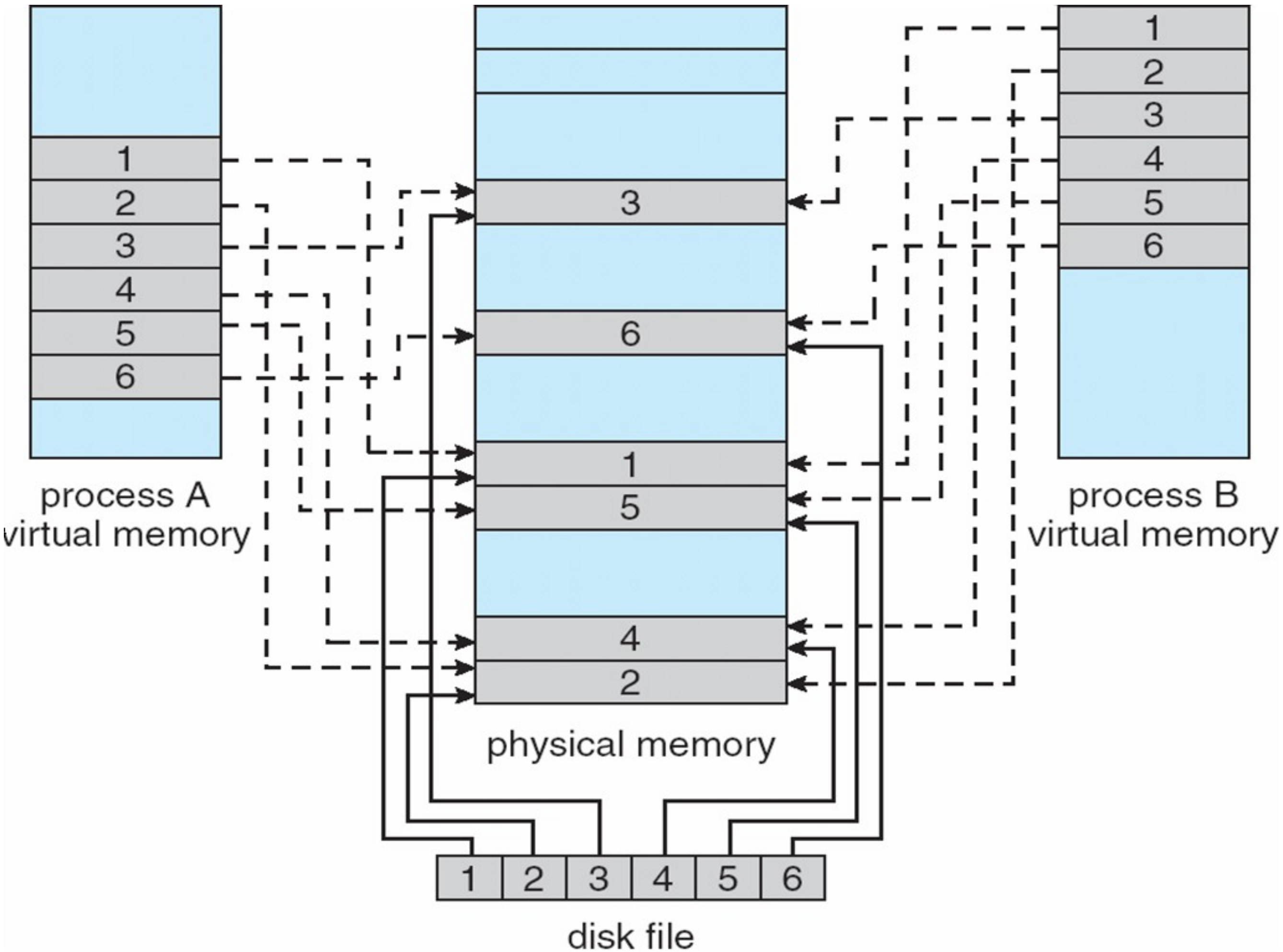
Page-Fault Frequency Strategy

- A much simpler approach than working set estimation is to simply monitor the page fault rate
- We set upper and lower bounds on the page fault rate of each process
  - If the rate is above the upper bound, we give the process another frame
  - If the rate is below the lower bound, we take a page away from the process
- If no new frame can be given to a process, we simply suspend it and swap it out entirely
  - Just like with the working set strategy

Memory-Mapped Files

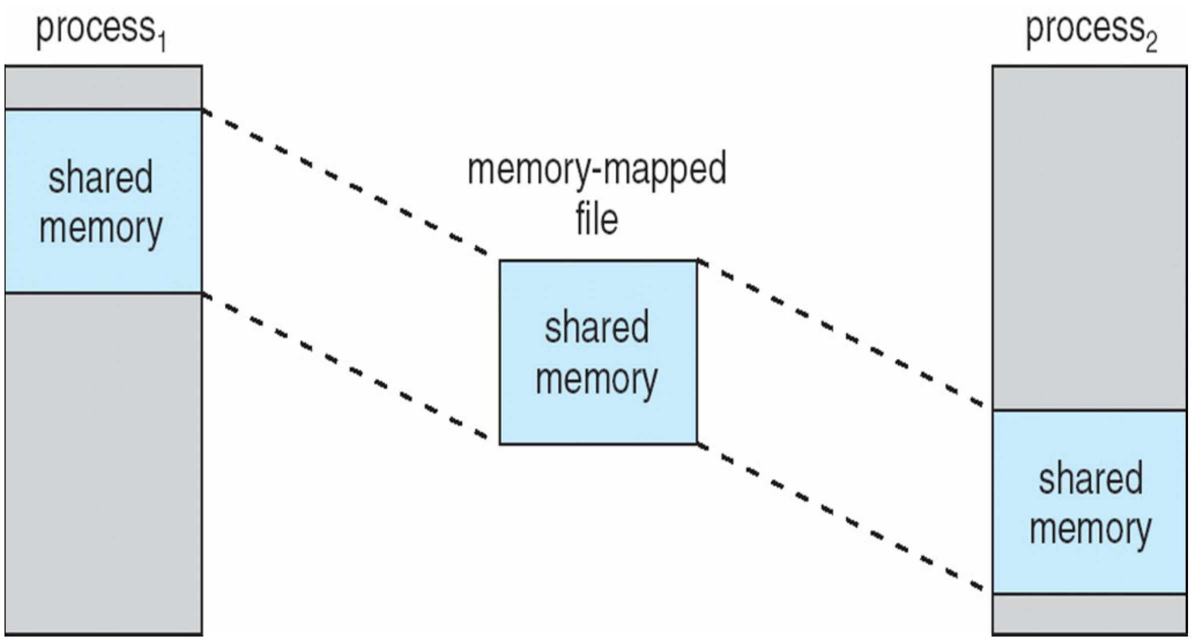
- I/O is known to be very expensive
  - Each access to the file requires disk access
  - Disk seek and access times are very high
- With virtual memory, on-disk address space pages are brought into RAM and written to disk later
- Why not do the same for files?
- Memory mapping: mapping a disk block to a memory frame
  - Initial access to the file generates a page fault
  - Subsequent accesses are in memory
    - read() and write() are “tricked” into going to memory rather than the disk
  - The on-disk file may be updated later, upon closing, etc.
- Memory mapping is via special system calls or by default
  - e.g., Solaris memory maps all files (in user or kernel space)
- Multiple processes may map the same file concurrently

Memory Mapping and Sharing



Memory Mapping and Shared Memory

- Memory mapping can be used to implement shared memory



- In Linux, there are separate mechanisms for memory mapping and shared memory
  - mmap() vs. shmget(), etc.
- In Windows shared memory is implemented with memory mapping as in the diagram above

Memory-Mapped I/O

- To access I/O devices, one can set aside ranges of memory addresses
- Loads/Stores to these addresses cause interaction with the device
- Convenient because then all memory-mapped I/O devices look similar

Conclusion

- Virtual Memory:
  - A process can be partially in memory
- Two key issues:
  - Page replacement
  - Frame allocation
- The thrashing problem and its solutions
- Memory-mapping for files or I/O