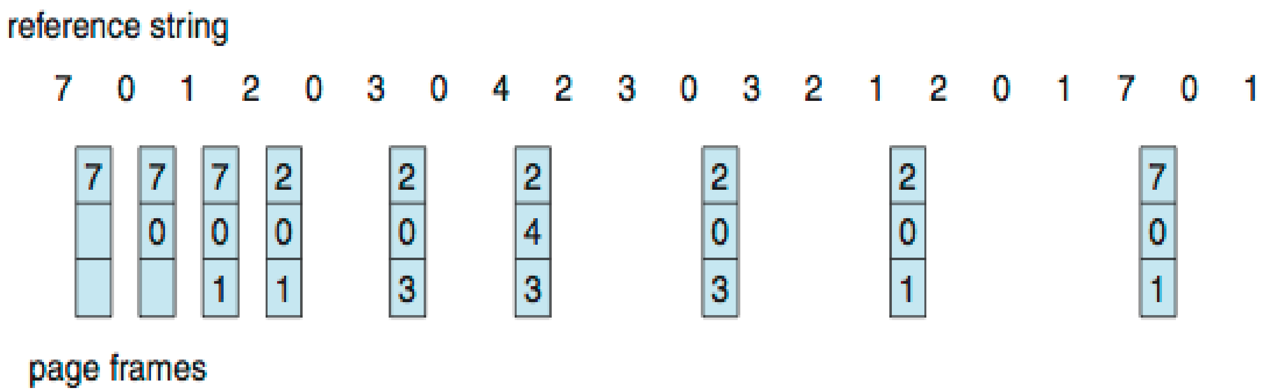
9.4.3 Optimal Algorithm

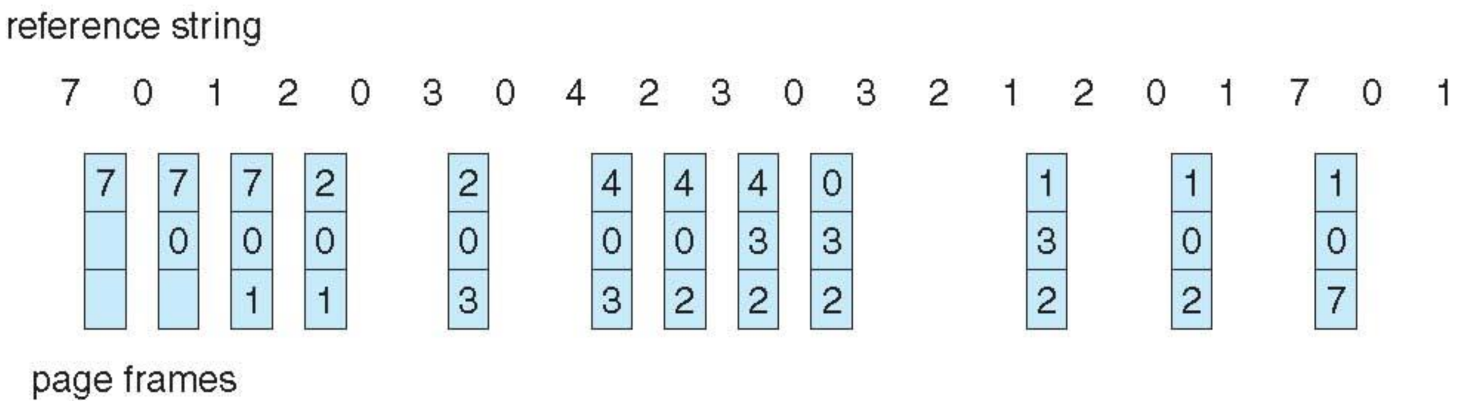
- Replace page that will not be used for longest period of time (i.e. measured by the **maximum forward distance** on page trace)
  - The optimal algorithm results in 9 page faults for the example page trace.
- How do you know this “not used for longest period of time”?
  - We don’t, we can’t read the **future**
    - But we can try to predict (as we shall see later)
  - It is a **hypothetical system** used for measuring how well your algorithm performs



9 page faults

9.3.4 Least Recently Used (LRU) Algorithm

- Use **past knowledge** rather than future
- Replace page that has not been used in the most amount of time (i.e. **maximum backward distance** on page trace)
- Associate time of last use with each page



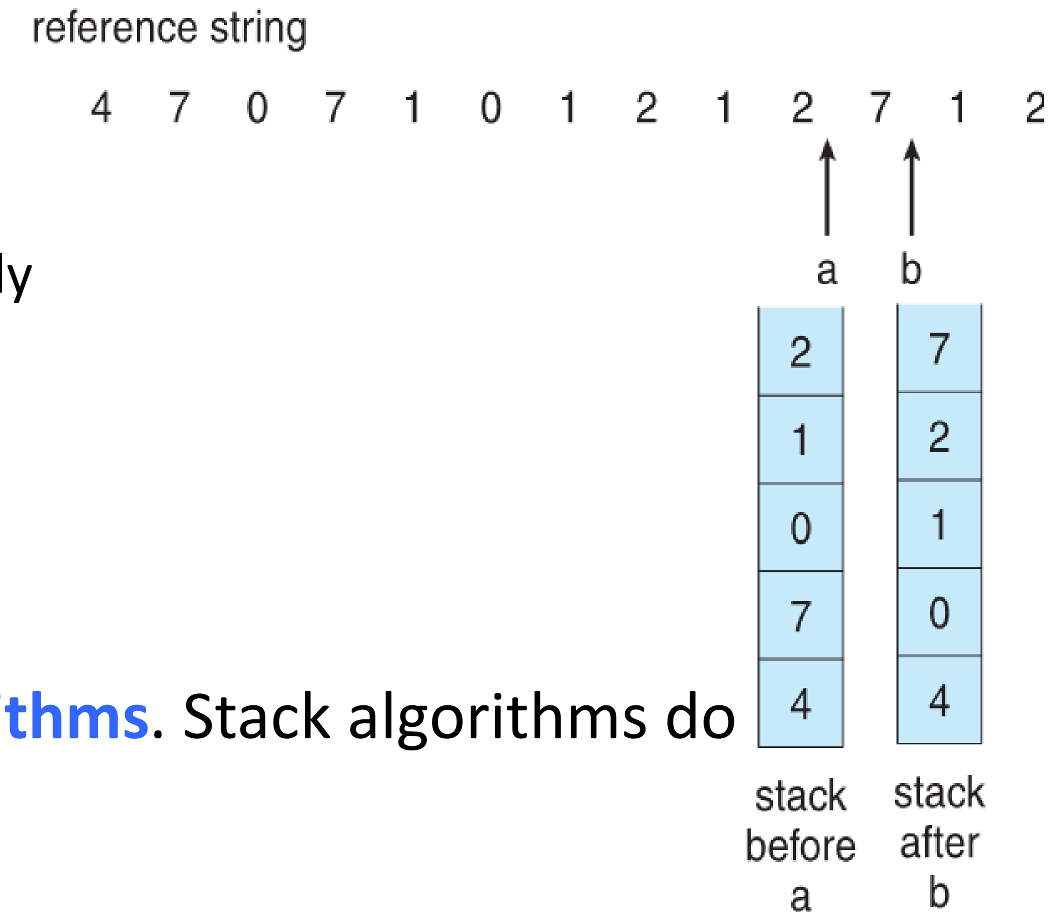
12 page faults

- 12 faults – better than FIFO but worse than the optimal algorithm
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

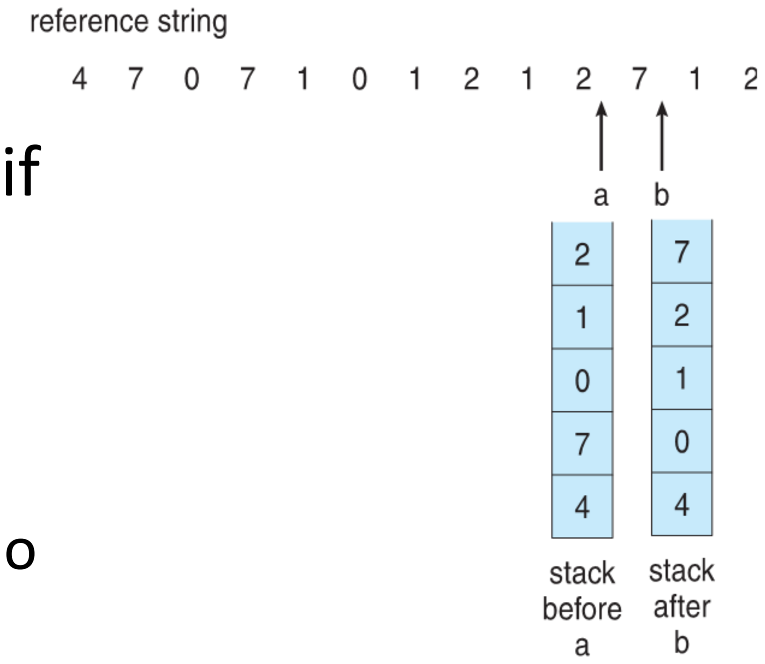
- Counter implementation
  - Every page entry has a counter variable; every time the page is referenced, the H/W copies the clock (Process' clock, not the CPU clock) into that page's counter
  - When a page needs to be changed, the OS kernel looks at the counters to find smallest time value

  (i.e. oldest reference, or LRU)
    - Search through page table needed
- Stack implementation
  - Keep a stack of page numbers in a doubly linked list form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement is needed
  - But each update is more expensive
- LRU and OPT are cases of **stack algorithms**. Stack algorithms do not exhibit the Belady Anomaly



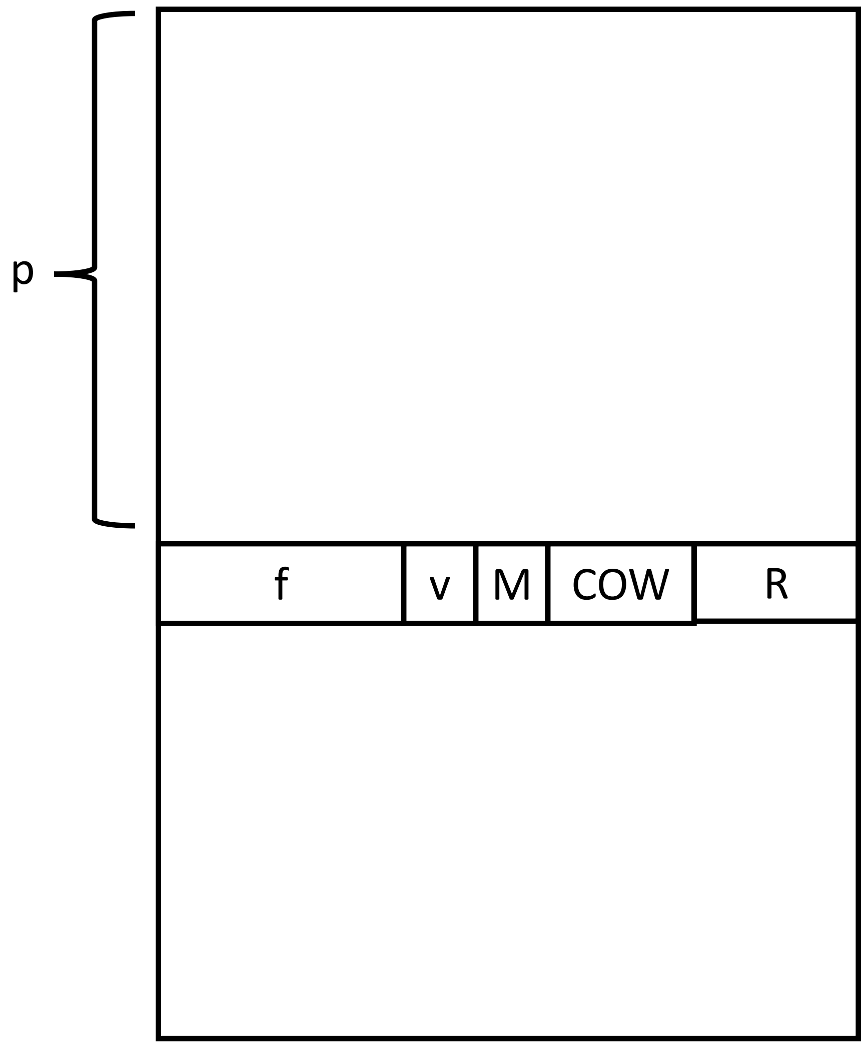
Least Recently Used – cont.

- In stack algorithms, a set of pages in memory for *N* frames is always a subset of the set of pages that would be in memory if *N + 1* frames were used.
  - If the number of frames increases from N to N+1, then the N *stacked* pages will still be the most recently referenced and remain in memory + one more --> predictable and has no randomness.
- This statement is not true for FIFO.



9.4.5 LRU Approximation Algorithms

- LRU needs special hardware and still slow
  - The OS cannot be invoked in every page access to update the timestamp → hardware must update the reference timestamp.
- **Reference bit**
  - With each page associate a reference bit:
    - All initialized to 0 (I.e. for all pages)
  - When page is referenced, H/W sets reference bit to 1
  - Replace a page whose reference bit = 0 (if one exists).
    - However, we may have more than one page with a reference bit of 0 and we do not know which one is older



Page Table

9.4.5 LRU Approximation Algorithms – cont.

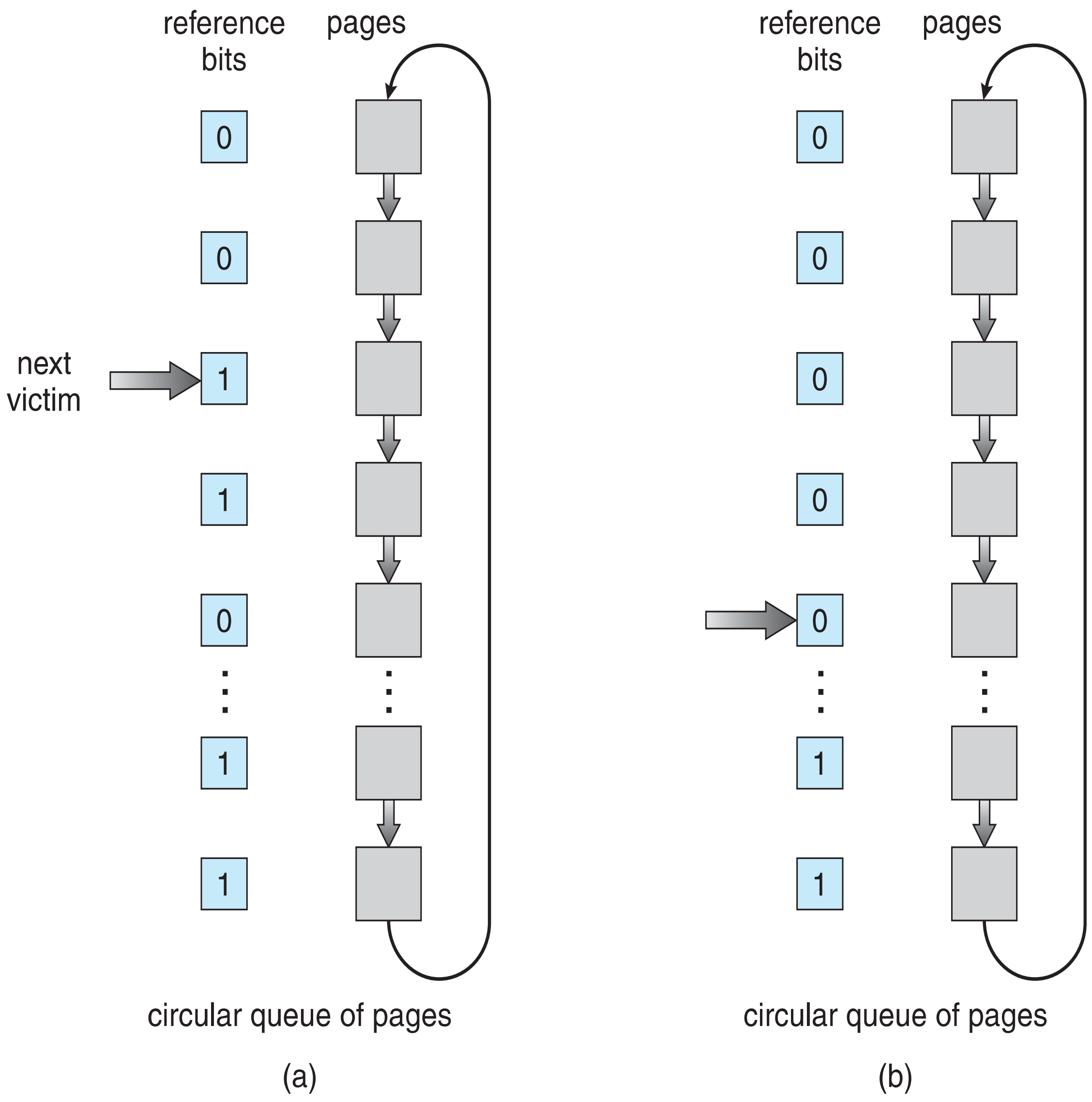
Additional-Reference-bits algorithm

- We can keep an 8-bit shift register for each page in a table in memory.
  - At regular intervals (e.g. 100 milliseconds), a **timer interrupt** transfers control to the operating system.
  - The OS right-shifts the register by one and then inserts the new reference bit (written by H/W on each page access, cleared by OS at end of the timer interrupt) on bit 7 → The 8-bit shift registers contain the history of page use for the last eight time periods.
  - A shift register containing **00000000** indicates that the page has not been used for eight time periods.
  - A page that is used at least once in each period has a shift register value of **11111111**.
  - A page with a history register value of 11000100 (value = 0xC4) has been used more recently than one with a value of 01110111 (value = 0x77).
- On page faults, the OS find the page with the lowest value is the LRU page → it can be replaced
- When multiple pages have the same value → use the FIFO method to choose among them.

9.4.5 LRU Approximation Algorithms – cont.

Second-chance algorithm

- No shift registers involved, but still requires the hardware reference bit.
- Similar to the FIFO algorithm but modifications – On a page fault:
  - A circular buffer implements the FIFO (looks like a **clock** whose hands point to the next page to examine)
  - On page fault, if currently examined page has:
    - Reference bit = 0 -> replace it.
    - Reference bit = 1, then:
      - set reference bit 0, leave page in memory → give it a ***second chance.***
      - Repeat for next page, subject to same rules



Second-Chance (clock) Page-Replacement Algorithm

LRU Approximation Algorithms – cont.

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used nor modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must swap the page out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement is called for, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

9.4.6 Counting Algorithms

- Keep a counter of the number of references that have been made to each page (since start of the reference string)
  - Since it is required to update the counter in each reference, this is done by the hardware → too complex → Makes this class of algorithms uncommon

- **Least Frequently Used (LFU) Algorithm**: replaces page with smallest count

- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  - thus eject page with MFU count assuming it was brought in too long ago and less likely to be needed again.

9.4.7 Page-Buffering Algorithms

- These are optimization algorithms.
- Always **keep a pool of free frames**, thus a frame is readily available when needed instead of being searched for when the page fault takes place:
  - Read page into free frame (swap in)
  - Select victim to evict (using some replacement algorithm)
  - When convenient, evict victim, i.e. **swap out on the background** and add its frame to free pool.
- Possible optimization:
  - keep list of modified pages
  - When backing store otherwise idle, write modified pages there and set to non-dirty

  i.e. try to keep as many pages as possible clean (**clean the pages on the background**)
  → clean pages do not need to be swapped out.
- Another possible optimization:
  - keep free frame content intact and note which pages reside on them (i.e. **don't swap out**)
  - If one of those pages is referenced again before the frame got a new resident, then there is no need to load contents again from disk.
  - Generally useful in reducing penalty if the wrong victim frame was selected.

9.4.8 Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
  - However, some applications have better knowledge – e.g. databases
  - Additionally, in database systems, the same memory may be buffered twice:
    - OS keeps copy of page in memory as disk I/O buffer
    - Database application keeps page in memory for its own work.
- Operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode allows a database application for example to have full access to a raw disk partition.
  - Bypasses buffering, filename search, locking, etc

9.5 Allocation of Frames

- Each process needs ***minimum*** number of frames
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - Dynamic allocation
- Many variations on these two schemes.

9.5.2 Fixed (static) vs variable (dynamic) Allocations

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - **Static** (fixed) allocation.
  - Keep some as free frame buffer pool

- **Proportional allocation** – Allocate according to the size of process
  - **Dynamic**, reacts to changes in the degree of multiprogramming or changes in process sizes.

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

– a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 62$

$s_1 = 10$

$s_2 = 127$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- Alternatively, allocate based on priority + size

9.5.3 Global vs. Local 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 (depending on other processes running in the system)
  - Greater throughput so more common

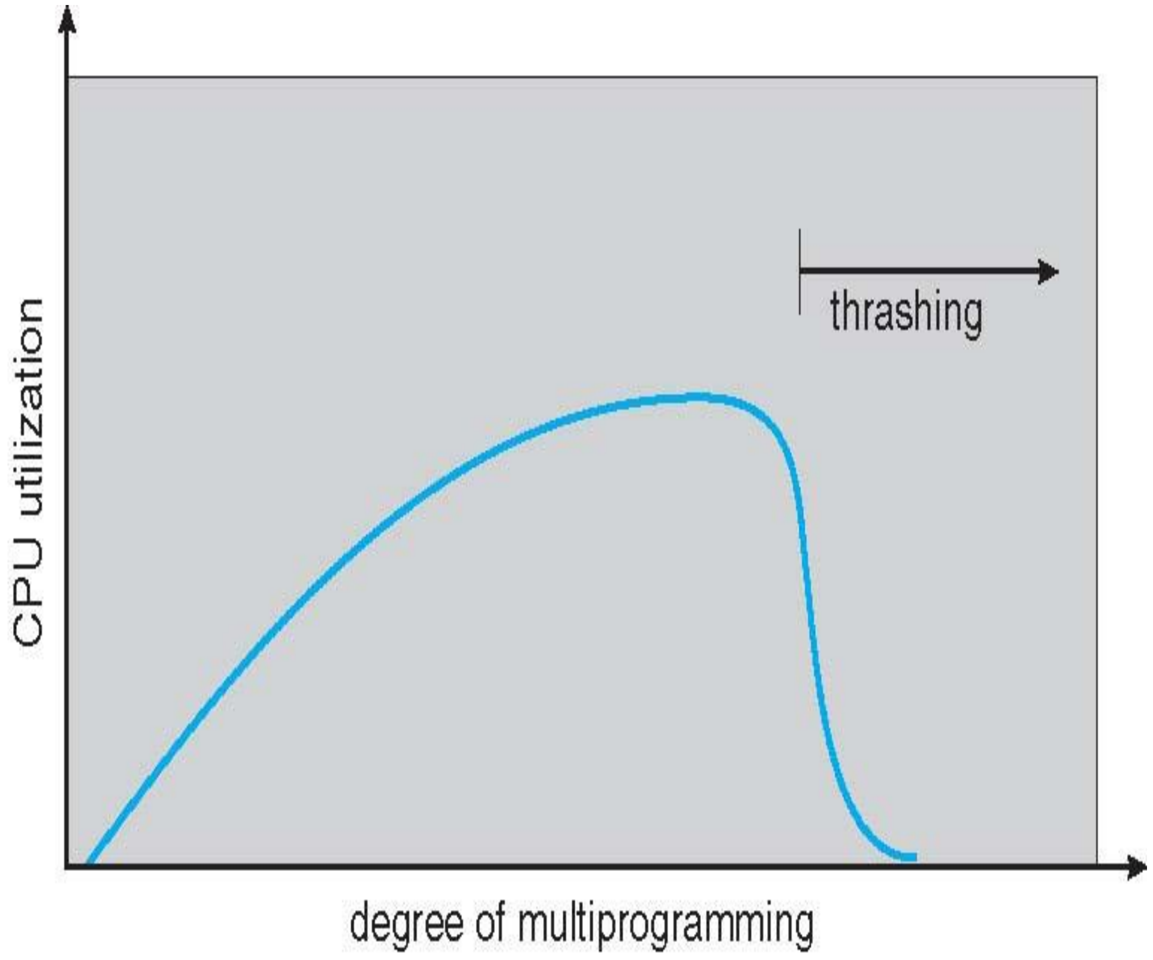
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

9.5.4 Non-Uniform Memory Access

- So far, all memory accessed equally, and we allocated numbers of frames to processes, without specifying their locations.
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule all threads of a process on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory **l**ow latency groups
    - Used by scheduler and pager
    - When possible schedule all threads of a process and allocate all memory for that process **within the same lgroup**

9.6 Thrashing

- If a process does not have “enough” frames allocated, the page-fault rate is very high
  - Page fault to get page
  - Replace a victim page
  - But quickly need evicted page back
  - This leads to:
    - Low CPU utilization
    - Operating system may think it needs to increase the degree of multiprogramming
    - Another process added to the system

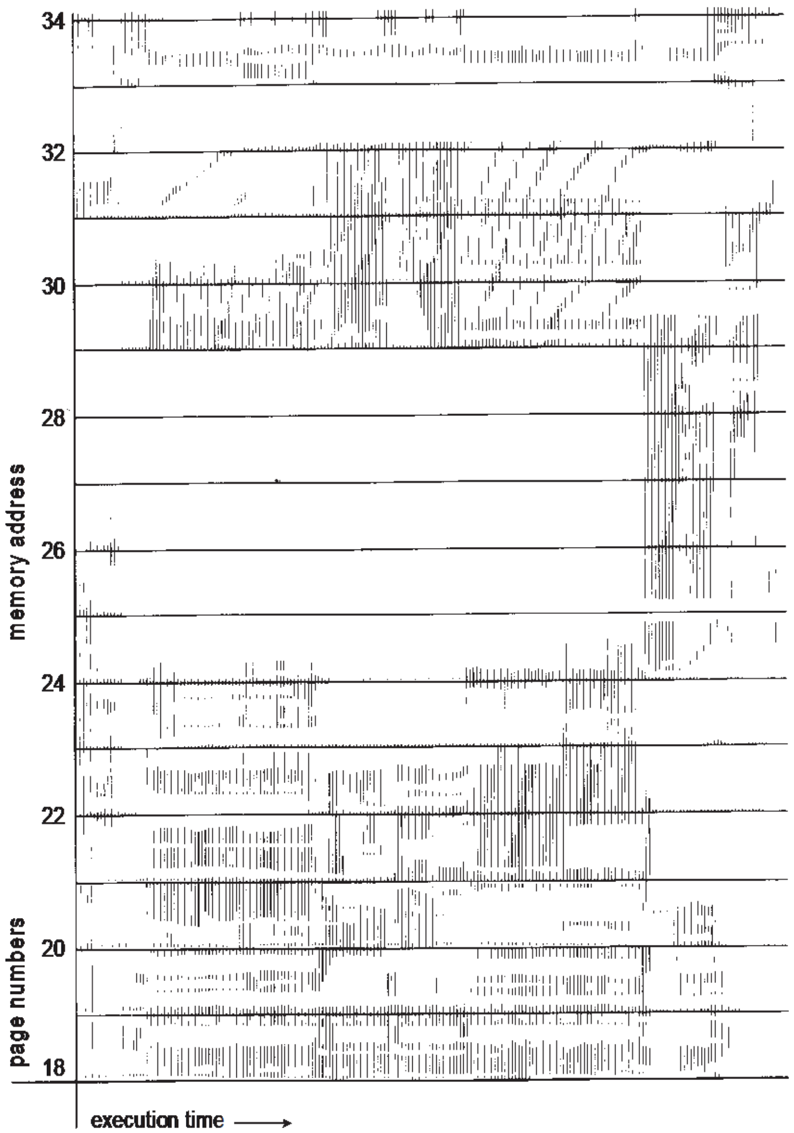


- **Thrashing** ≡ a process is busy swapping pages in and out

Demand Paging and Thrashing

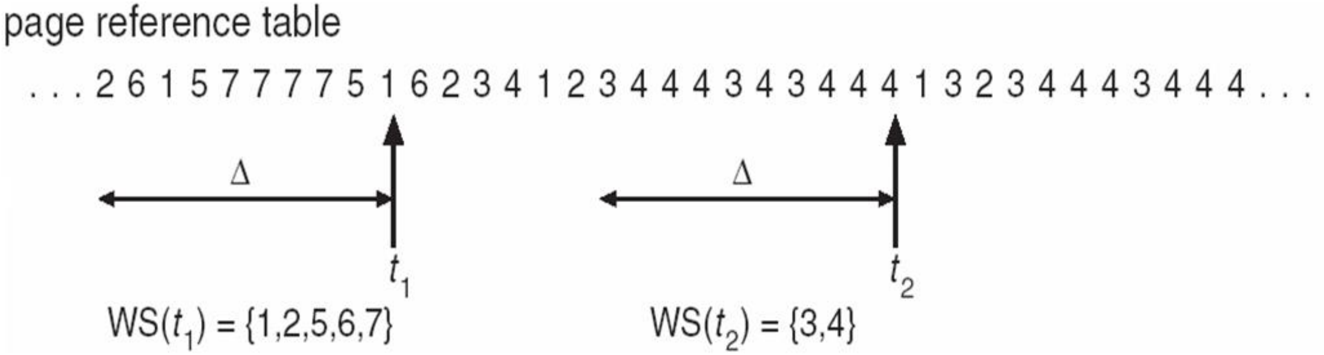
- Why does demand paging work?
  Locality model
  - As a process executes, it moves from locality to another. **A locality is a set of pages that are actively used together**.
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size allocated
  - Thus allocate the process the frames it needs to avoid thrashing.
  - Limit effects by using dynamic frame allocation
    - In reality, what matters is the size of the locality of a process, not the total size or priority of the process --> how do we determine the locality size? (next slides)



Working-Set Model

- **$\Delta \equiv$ working-set window** $\equiv$ a time window specified by a fixed number of page (or memory) references. Note that this **is a sliding window**.
  Example: 10,000 instructions (or 10,000 cycles $\equiv$ 10,000 page references)
- ***WSS$_i$* (working set size of Process *P$_i$*)** =
  total number of pages referenced in the most recent Δ (varies over 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



- $D = \Sigma\ WSS_i \equiv$ total demand frames for all processes
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes
  - **Which process scheduler** does that (short term, medium term or long term scheduler?)

Keeping Track of the Working Set

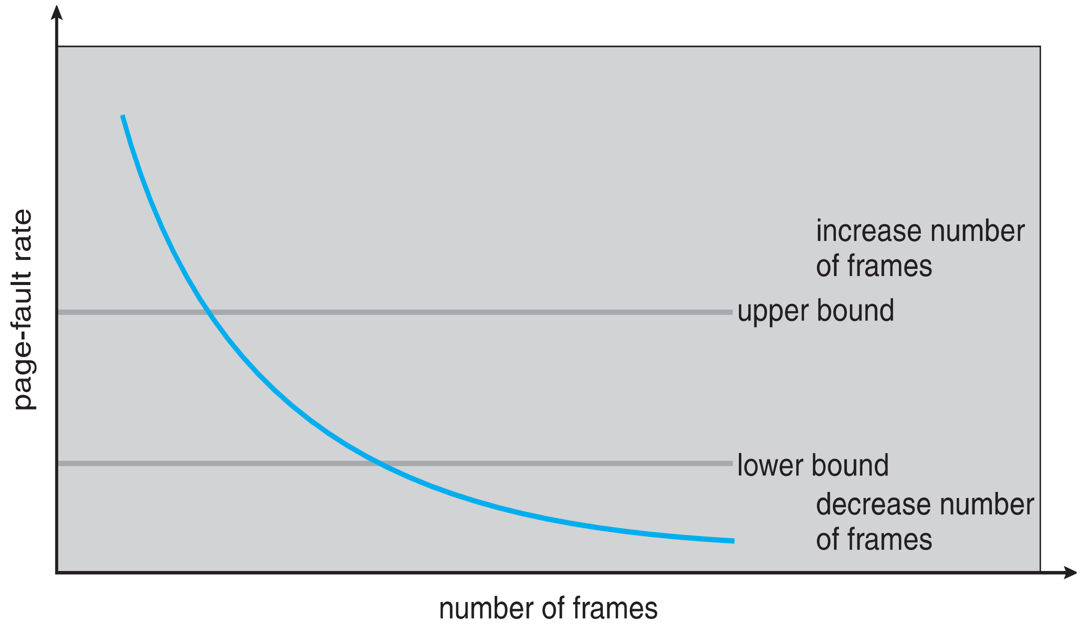
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bit that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 8 bits and interrupt every 1250 time units

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units (hence 2 interrupts)
  - Keep in memory 2 bits for each page (one per interrupt).
  - This is in addition to the reference bits that exists on the page table (one for each page, which is set by the H/W whenever the page is accessed).
  - Whenever a timer interrupt occurs, copy the reference bits on the page table to the corresponding memory-stored reference bits (i.e. one of the 2 bits per page) and set values of all page table reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Working set slides by ${}^{\Delta}/_{2}$
- Improvement = 8 bits and interrupt every 1250 time units
  - Working set slides by ${}^{\Delta}/_{8}$

Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use dynamic allocation policies
  - If actual rate too low, process loses frames
  - If actual rate too high, process gains frames



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
  - Page faults peaks are when a process is changing locality (or working set)
- Peaks and valleys over time