Virtual Memory

the role of the operating system

resident set

- the resident set of a process at a given time is the set of pages that are in main memory at that time
  - its content chages over time
  - its size may change depending on the OS policies

Fetch Policy

- 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 a location on the page
    - Many page faults when process first started
  - Prepaging brings in more pages than needed
    - More efficient to bring in pages that reside contiguously on the disk

Placement Policy

- Determines where in real memory a process piece (segment or page) is to reside
- Important in a segmentation system
- Paging or combined paging with segmentation hardware performs address translation

Replacement Policy

- Replacement Policy
  - Which page is replaced?
  - Page removed should be the page least likely to be referenced in the near future
  - Most policies predict the future behavior on the basis of past behavior

Replacement Policy

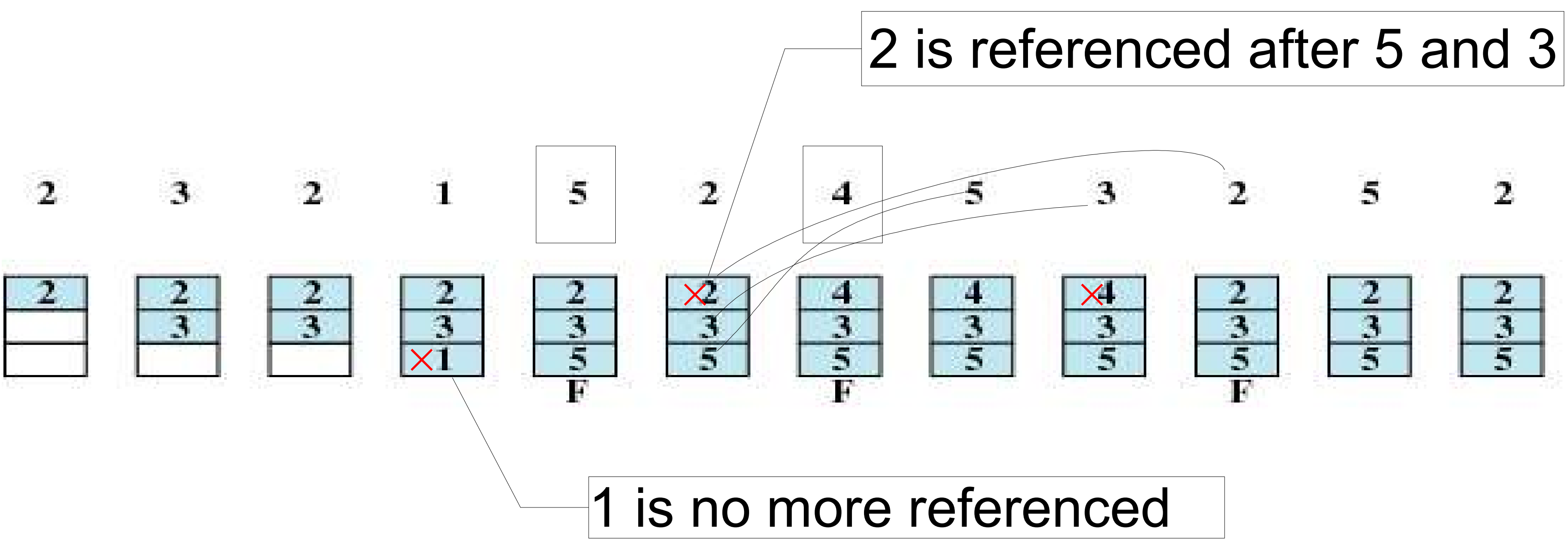
- Frame Locking
  - If frame is locked, it may not be replaced
  - Kernel of the operating system
  - Control structures
  - I/O buffers
  - Associate a lock bit with each frame

Basic Replacement Algorithms

- **Optimal policy**
  - Selects for replacement that page for which the time to the next reference is the longest
  - **results in the fewest number of page faults**
  - no other policy is better than this
  - Impossible to implement
    - it needs to have perfect knowledge of future events

optimal policy example

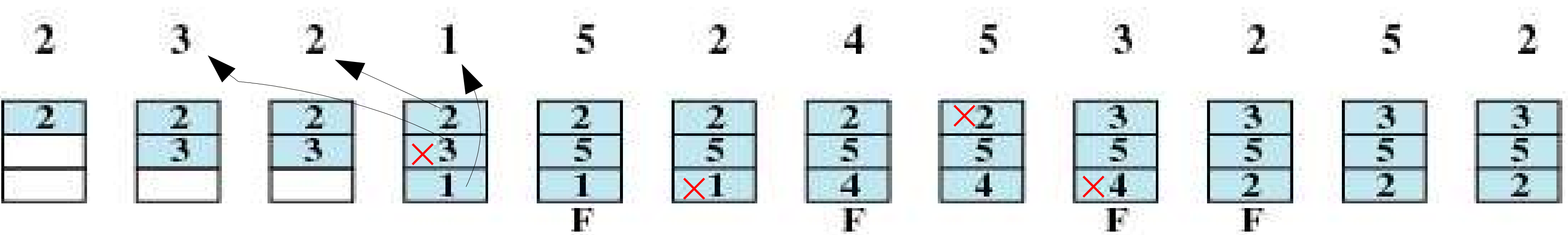
- page address stream:
  2 3 2 1 5 2 4 5 3 2 5 2
- 3 frames are available



Basic Replacement Algorithms

- **Least Recently Used (LRU)**
  - Replaces the page that has not been referenced for the longest time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - Each page could be tagged with the time of last reference. This would require a great deal of overhead.
    - timestamp update for each reference in memory!

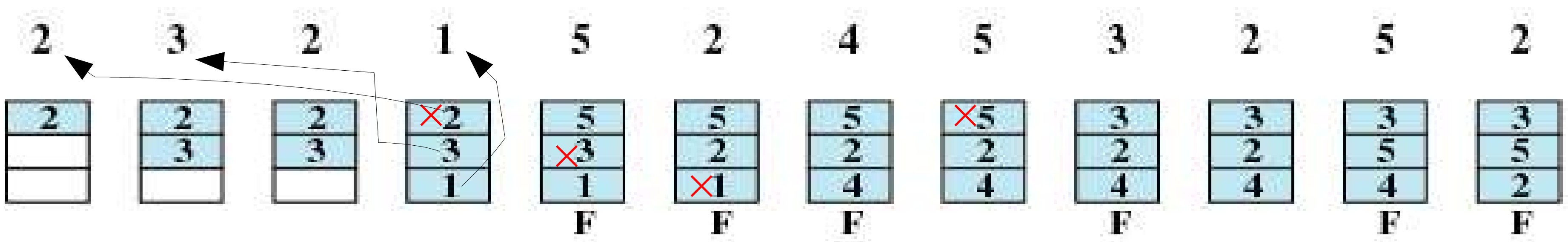
LRU policy example



Basic Replacement Algorithms

- **First-in, first-out (FIFO)**
  - Treats page frames allocated to a process as a circular buffer (queue)
  - Pages are removed in round-robin style
  - Simplest replacement policy to implement
  - Page that has been in memory the longest is replaced
  - These pages may be needed again very soon

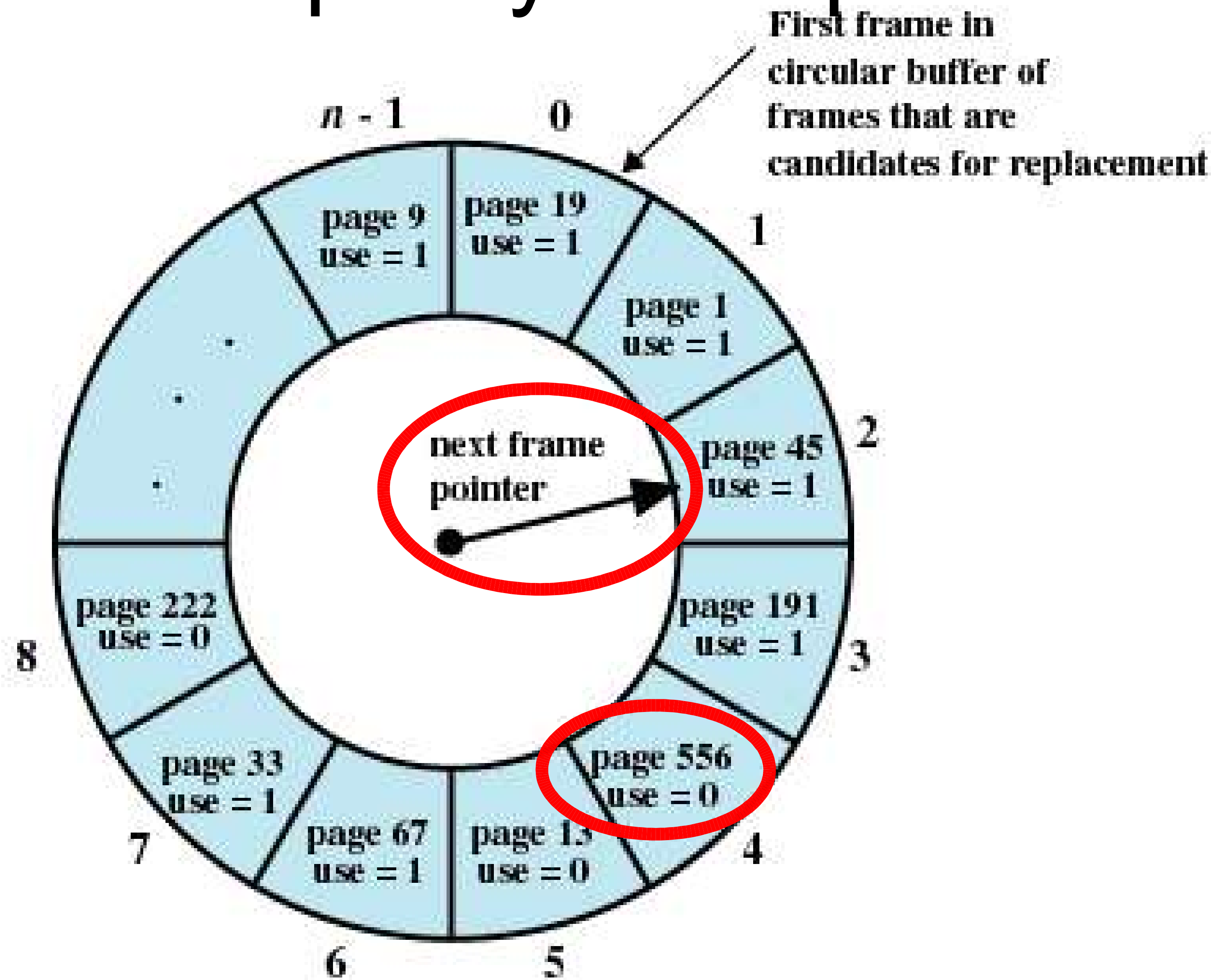
FIFO policy example



Basic Replacement Algorithms

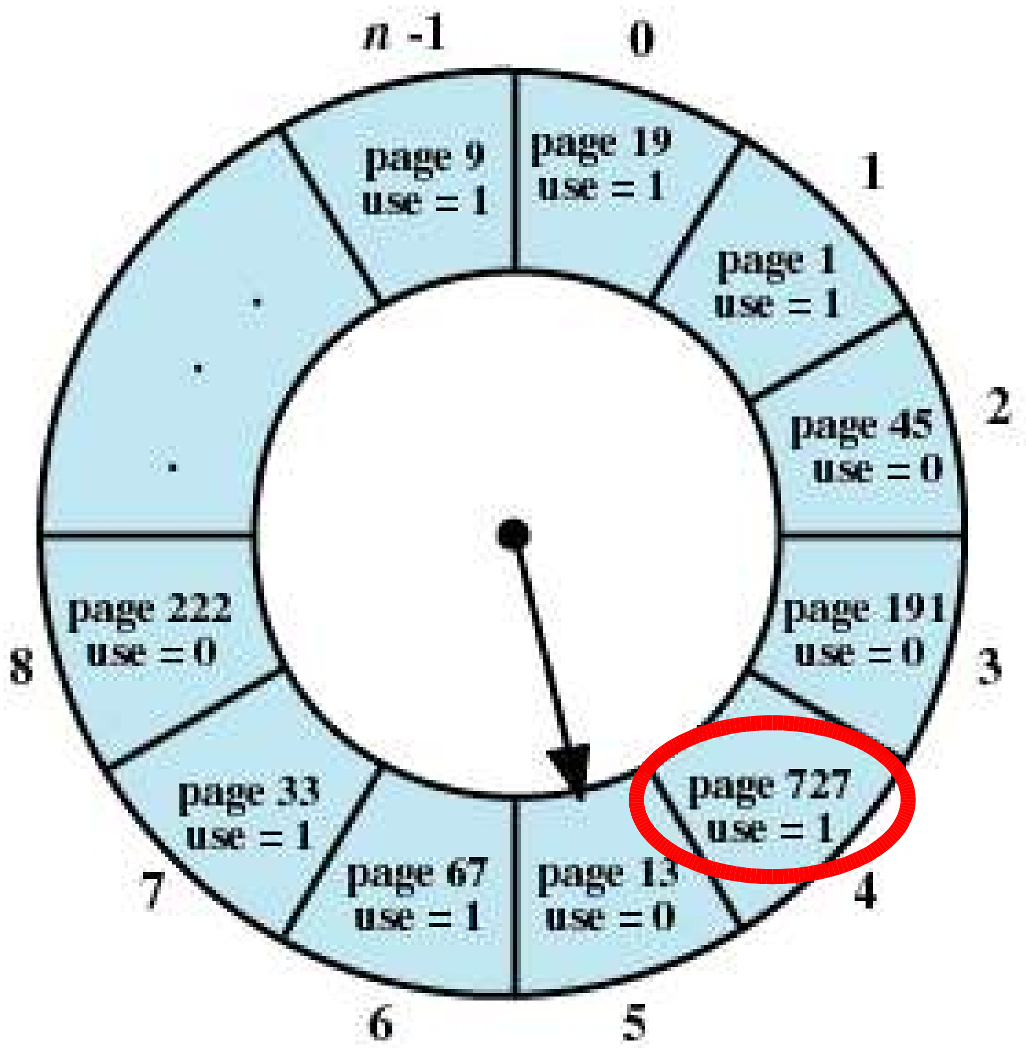
- **Clock Policy (second chance)**
  - one additional for each page bit called a use bit
  - set use=1
    - when a page is first loaded in memory
    - each time a page is referenced
  - when it is time to replace a page scan the frames...
    - the first frame encountered with use=0 is replaced
    - while scanning if a frame has use=1, set use=0

clock policy example



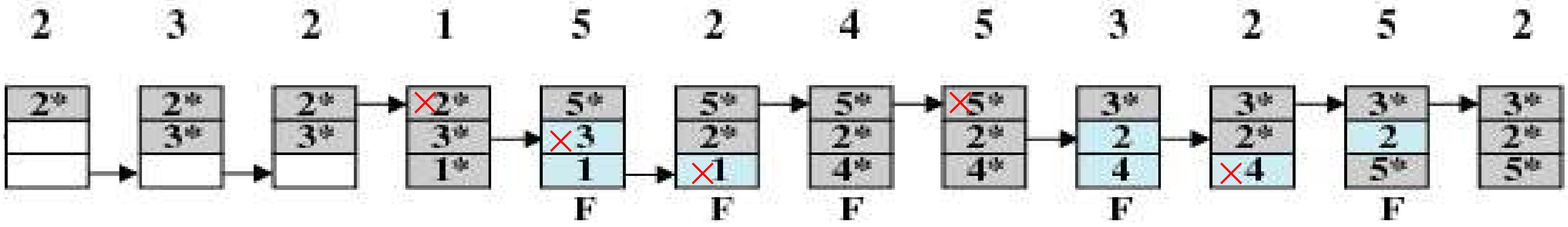
(a) State of buffer just prior to a page replacement

clock policy example

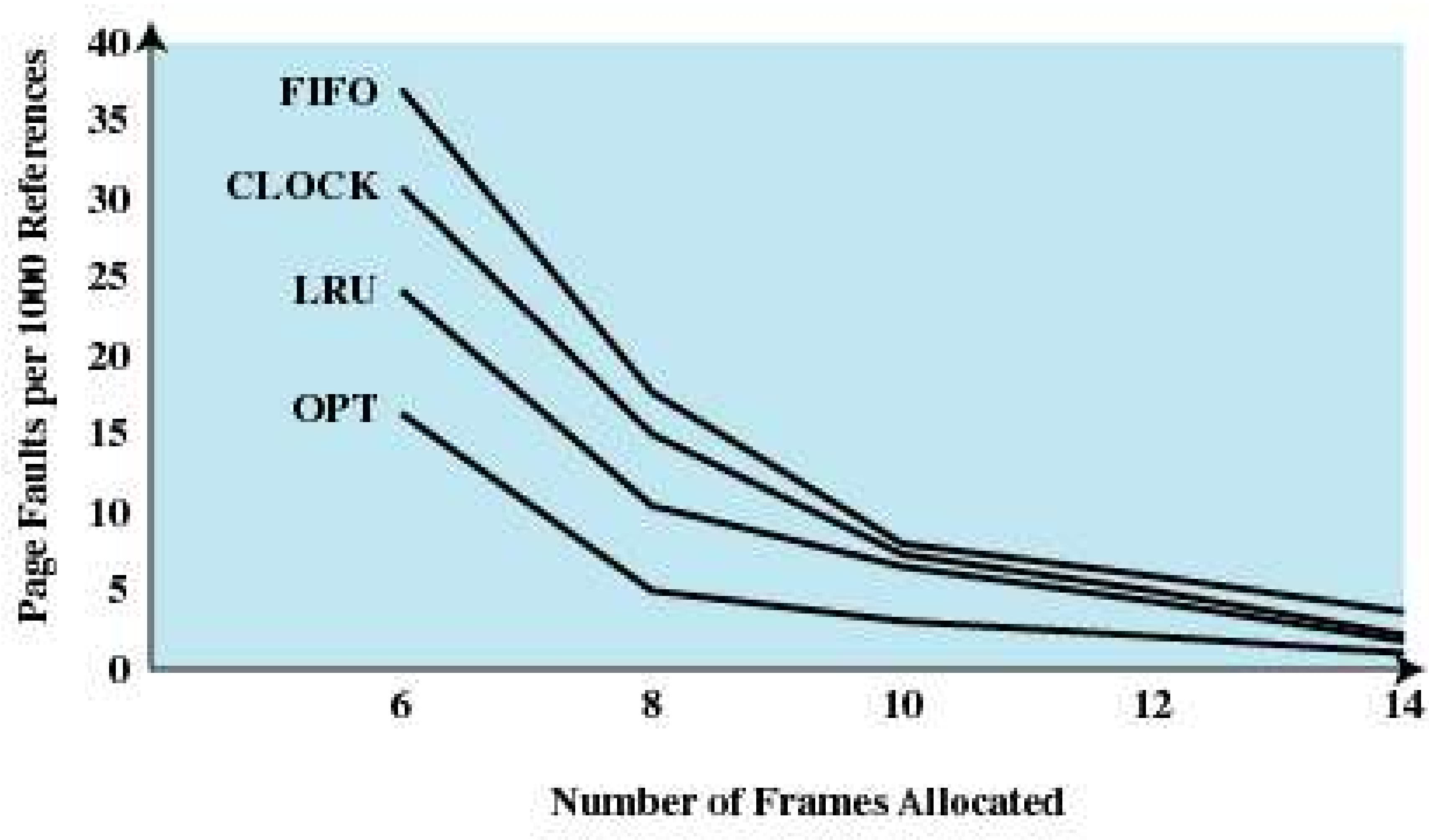


(b) State of buffer just after the next page replacement

clock policy example

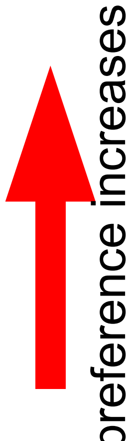


comparison of replacement algorithms



clock policy variation

- we prefer to replace frames that have not been modified
  - since they need not to be written back to disk
- two bits are used (updated by the hardware)
  - use bit
  - modified bit
- frames may be in four states
  - not accessed recently, not modified
  - not accessed recently, modified
  - accessed recently, not modified
  - accessed recently, modified



clock policy variation

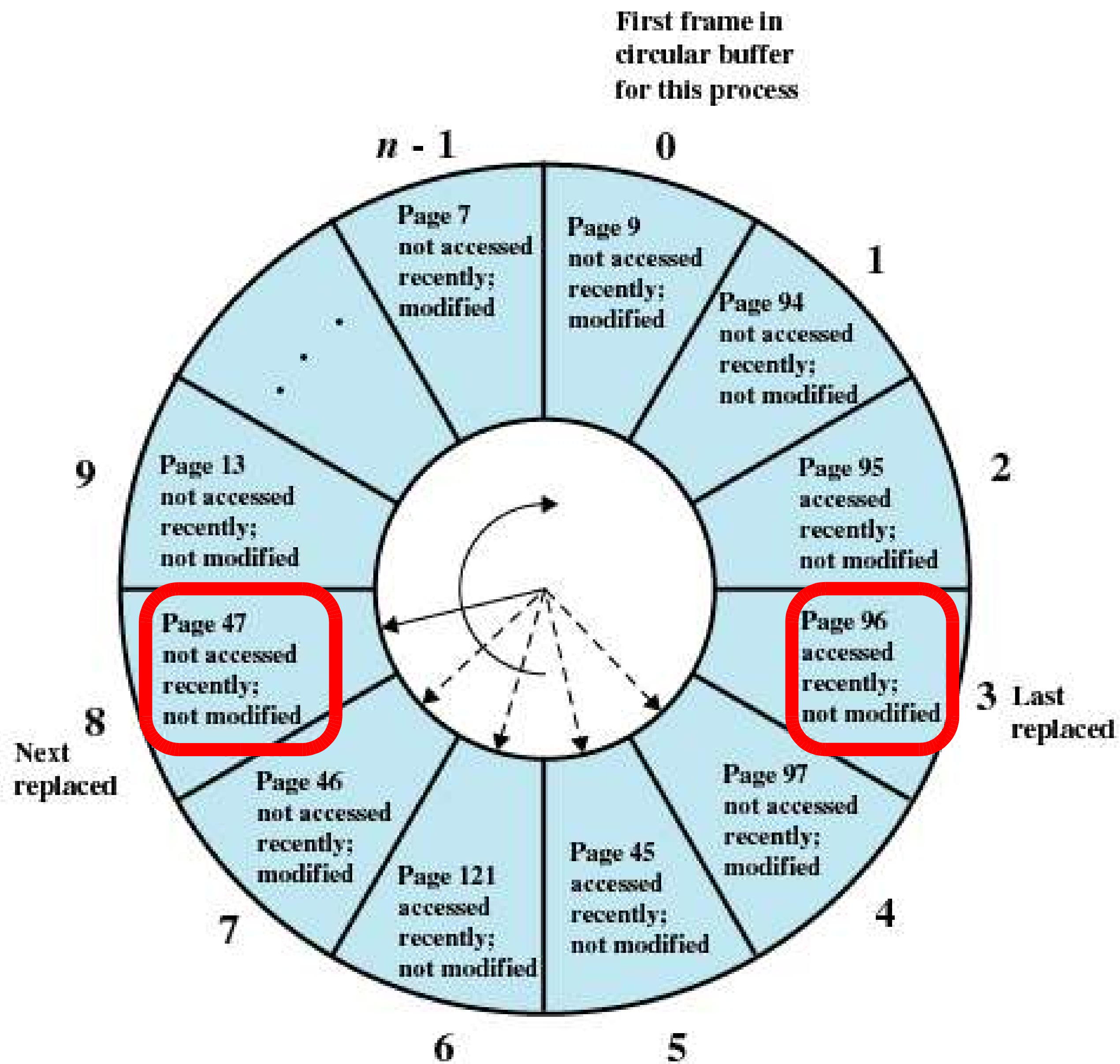
1 look for frames not accessed recently and not modified (use=0, mod=0)

2 if unsuccessful, look for frames not accessed recently and modified (use=0, mod=1)

- ... while setting use=0 as in regular clock.

3 if unsuccessful, go to step 1

clock policy variation



aging policy

(da Tannenbaum)

- for each age keeps an age “estimator”
  - the less is the value the older is the page
- periodically sweep all pages...
  - scan use bits and modify estimator for each page, two possible alternatives:
    - shift right (that is divide by two) all estimators and insert the value of use bit as leftmost bit
    - decrement all estimators for unused pages and increment those belonging to used pages
  - clear use bits
- evict pages starting from older ones
  - lower estimator

aging policy

right shift

- R is the use bit

	(a)	(b)	(c)	(d)	(e)
	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0

Page	(a)	(b)	(c)	(d)	(e)
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	**00010000**	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

aging policy

- it is a form of LRU approximation
- ages are quantized in time
  - many references between two sweeps are counted once
- very old references are forgotten
  - when an estimator reach zero it remains unchanged

Page Buffering

- system always keeps a small amount of free pages
- pages replaced are added to one of two lists
  - Free page list, if page has not been modified
  - Modified page list, otherwise
- pages of the free list are physically overwritten only if the page is really re-assigned
- if the page is claimed again it may be given to the process without any access to secondary memory

Page Buffering

- when a modified page is written out it is put into the free page list
- modified pages can be written out on secondary memory in clusters reducing the number of I/O
- page buffering has been adopted to “correct” simple policies like FIFO

resident set management

- resident set size
  - how many pages are in memory for each process?
- replacement scope
  - what is the set of pages that are considered for replacement?

Resident Set Size (RSS)

- Fixed-allocation
  - Gives a process a fixed number of pages within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
  - Number of pages allocated to a process varies over the lifetime of the process

Replacement Scope

a process A generated a page fault

- that is, a page of A must be loaded in memory
- it will take the place of another page, which one?

- local policy
  - the page to be replaced is chosen among the pages of A
- global policy
  - the page to be replaced is chosen among all the pages in memory regardless of the process they belong to.

fixed allocation, global scope

- not possible

Fixed Allocation, Local Scope

- Decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate
- If allocation is too large there will be too few programs in main memory
  - bad usage of main memory

Variable Allocation, Global Scope

- Easiest to implement
- Adopted by many operating systems
- Operating system keeps list of free frames
- A free frame is added to resident set of a process when a page fault occurs
- If no free frame, replaces one from another process

Variable Allocation, Local Scope

- When a new process is added, allocate a number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time
  - see **“working set”**

(memory) virtual time

- consider a sequence of memory references generated by a process *P*
  r(1), r(2),...
- r(i) is the page that contains the i-th address referenced by *P*
- t=1,2,3,... is called (memory) **virtual time** for *P*

it can be approximated by “process” virtual time

  – memory references are uniformly distributed in time

working set

- defined for a process at an instant (in virtual time) t and with a parameter Δ (window)
  - denoted by $W(t, \Delta)$
- $W(t, \Delta)$ for a process P is the set of pages referenced by P in the virtual time interval $[t - \Delta + 1, t]$
  - the last Δ virtual time instants starting from t

working set properties

the larger the window size, the larger the working set.

$$W(t, \Delta+1) \supseteq W(t, \Delta)$$

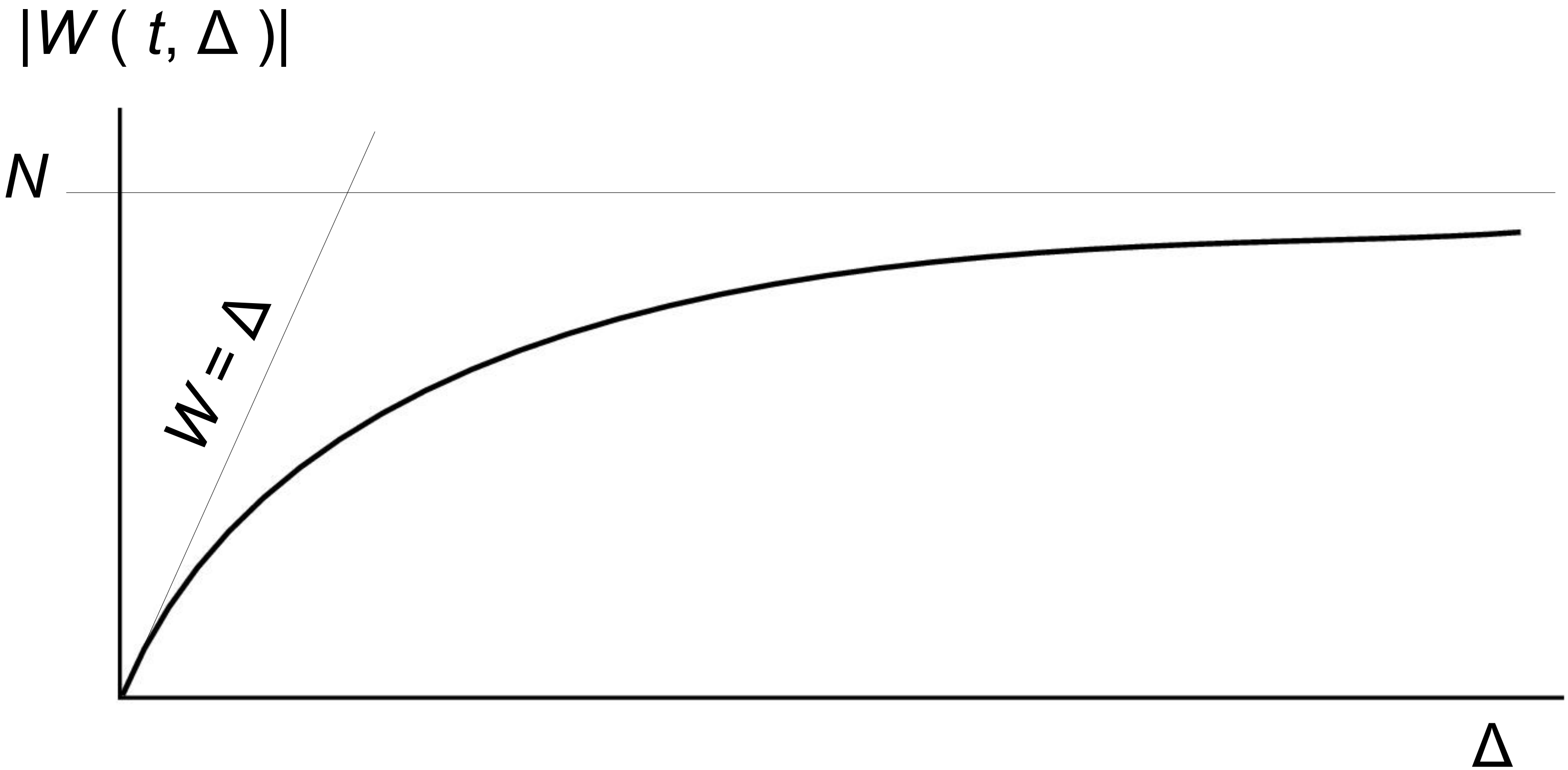
upper bound for the size of *W*

$$1 \leqslant |W(t, \Delta)| \leqslant \min(\Delta, N)$$

N number of pages in the process image

working set

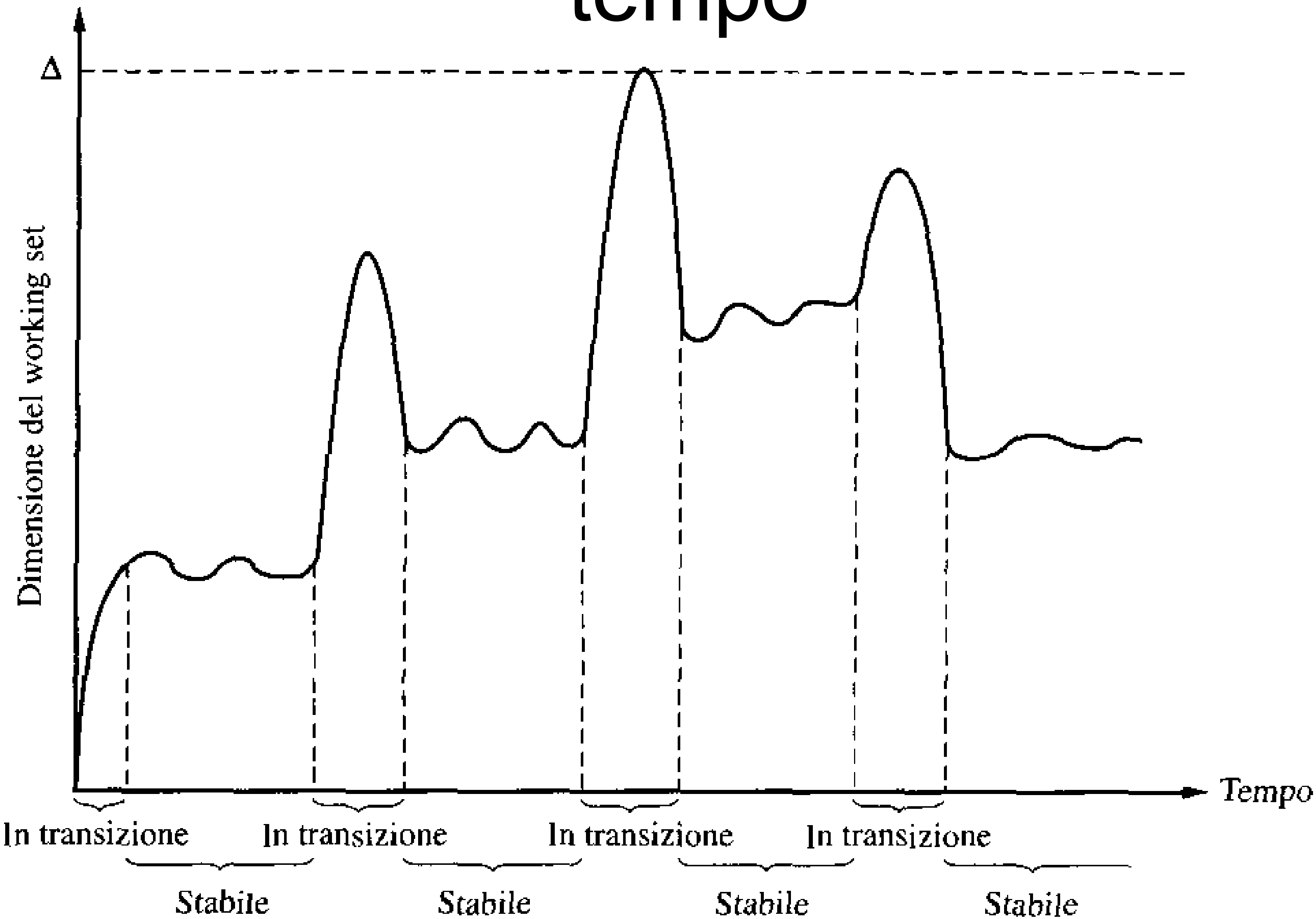
andamento al variare di Δ per *t* fissato *t*>>*N*



working set: esempio

Sequenza di riferimenti a pagina	Dimensione della finestra, Δ 2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	18 24	17 24 18	24 17 18	15 24 17 18

working set: andamento tipico nel tempo

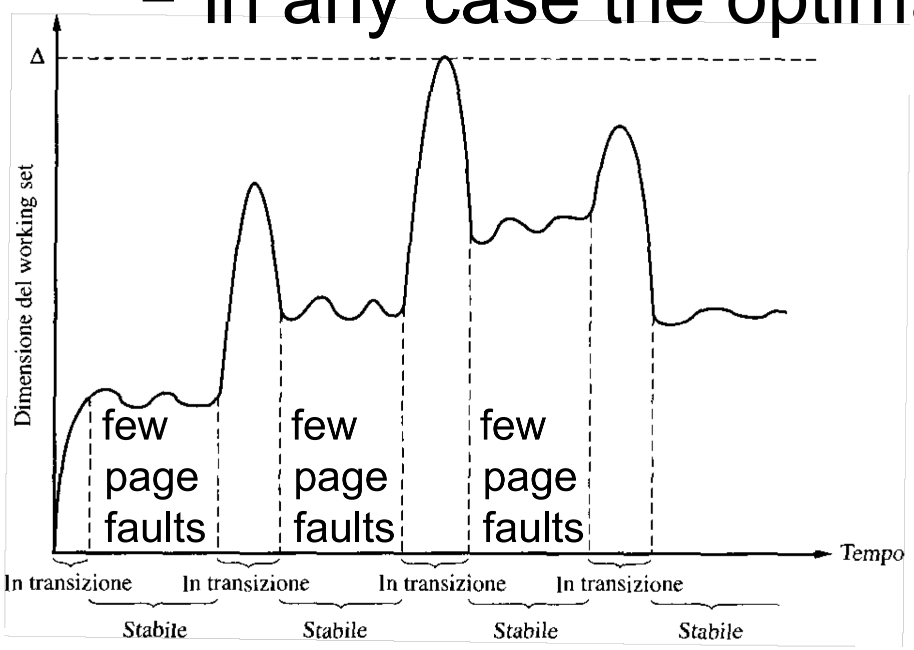


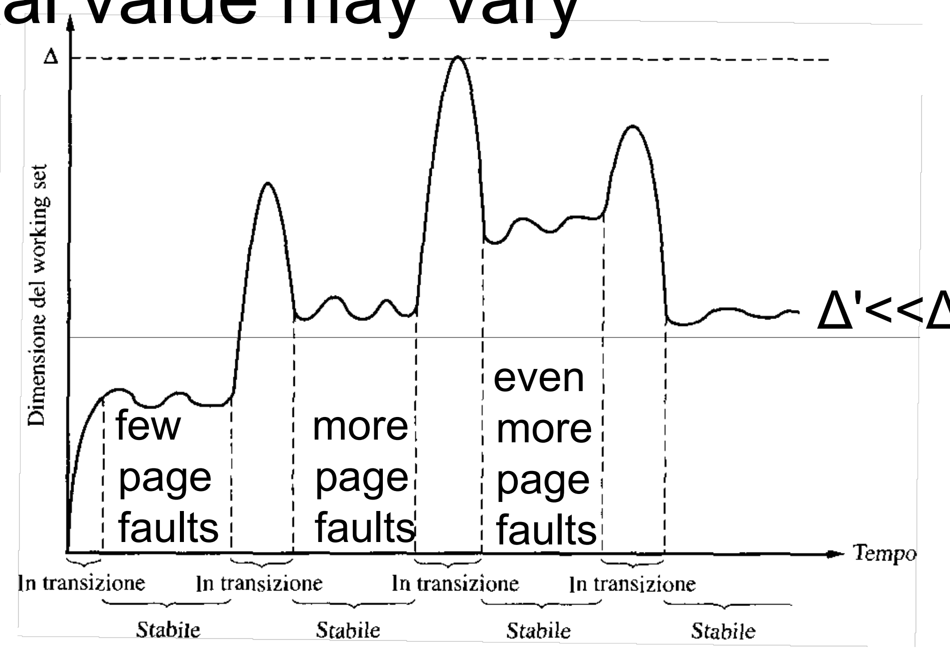
our goal

- ideally we would like to have alaways the working set in memory
- WS strategy
  - monitor the WS of each process
  - update the WS
    - page faults add pages to the WS
    - periodically remove pages of the resident set that are not in the WS. Basically: LRU with variable resident set size.

working set strategy: problems

- optimal Δ?
  - larger Δ → less page faults and larger |W|
  - compromise between number of page faults and WS size!
  - in any case the optimal value may vary



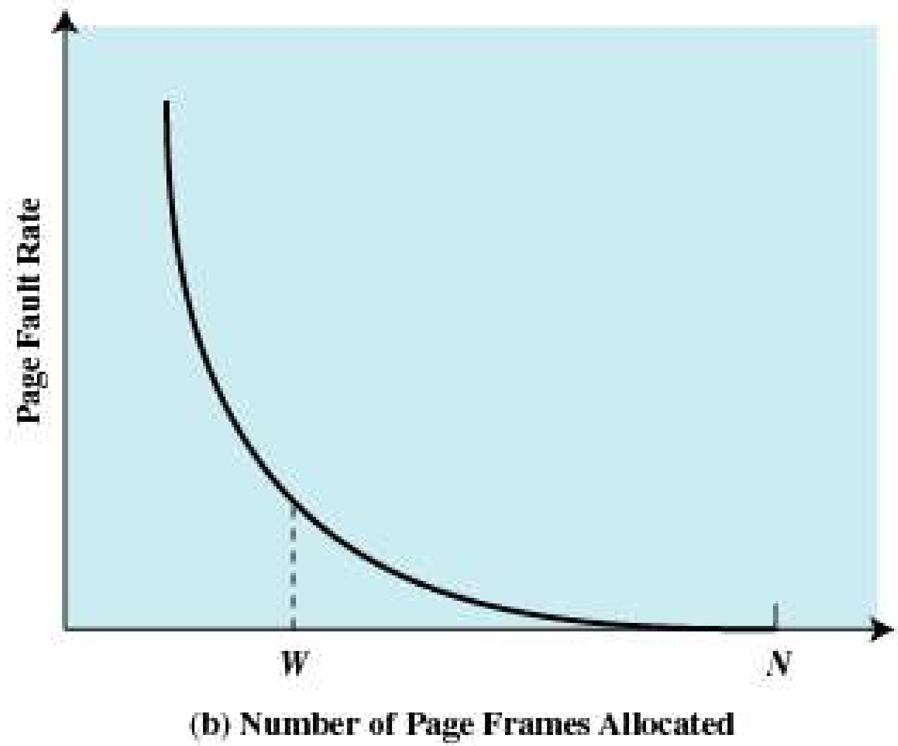


working set strategy: implementation problems

- we need to maintain the history of the reference for Δ
  - more and more difficult as Δ increase
- it should be done in real-time
  - keep a list of the memory reference in hw?
  - count memory reference and mark pages with the current value of the counter?
  - in any case we need hw support

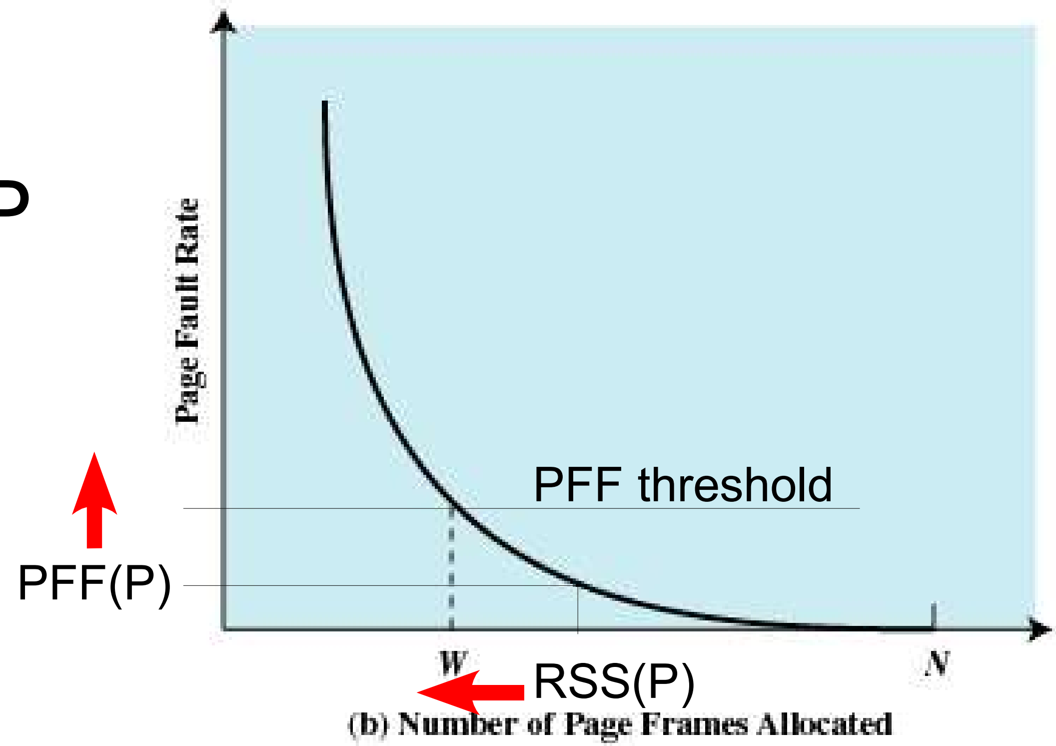
page fault frequency (PFF)

- page fault frequency depends on the resident set size
- monitor PFF instad of W
  - much easier



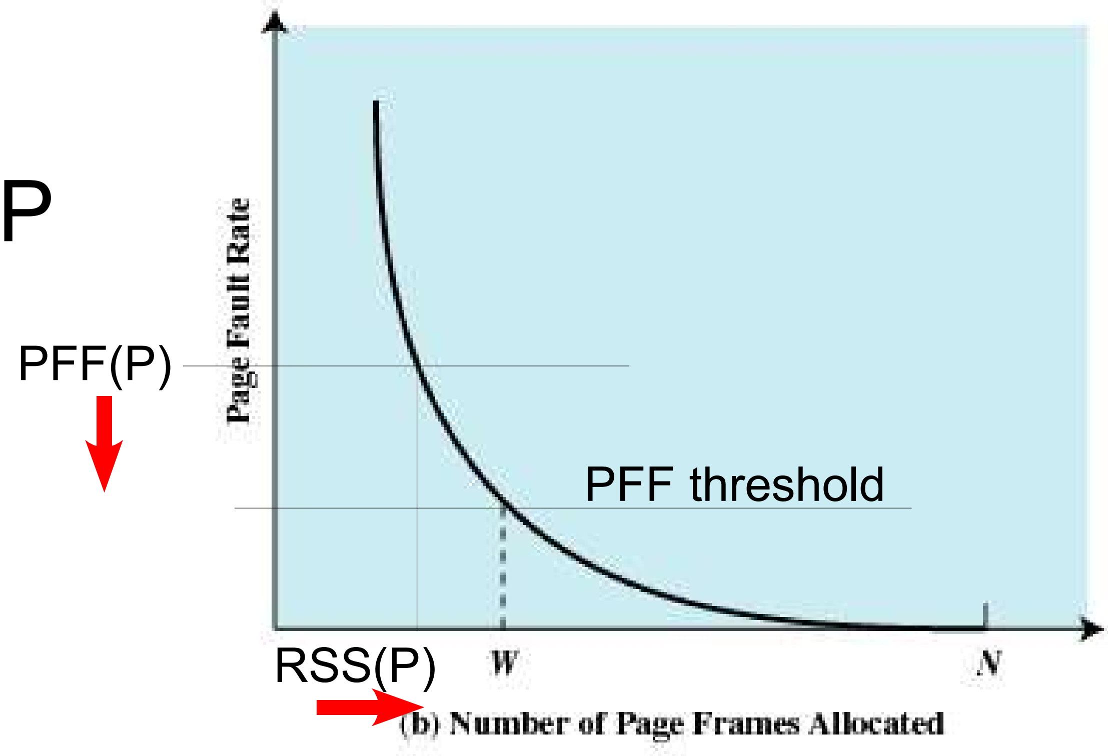
page fault frequency (PFF)

- if PFF is below a threshold for *P*, decrease RSS of P
- the whole system will benefit



page fault frequency (PFF)

- if PFF is above a threshold for *P*, increase RSS of P
- *P* will benefit



PFF policy implementation

- maintain a counter *t* of the memory references (it count virtual time)
- on each page fault update estimation of PFF
  - keeping the time t_1 of the last page fault PFF≈1/(t-t_1)
  - keeping a first order estimator

$$PFF_{now} = \alpha \frac{1}{t - t_1} + (1 - \alpha) PFF_{prev}$$

$$\alpha \in (0, 1]$$

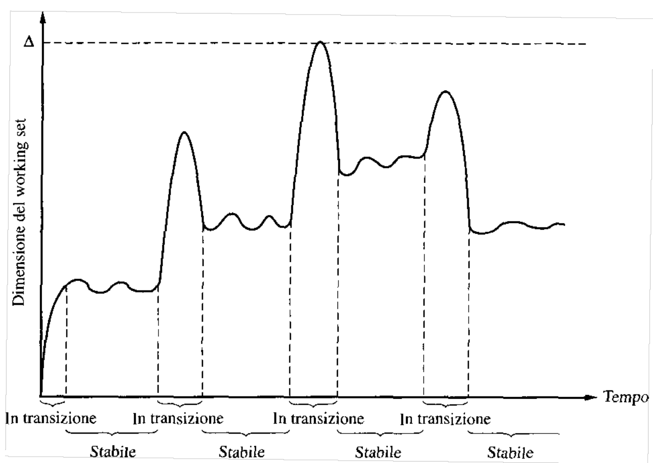
- decide action on estimated PFF

PFF policy implementation

- if PFF is above the $PFF_{threshold}$
  - increse the RSS
- if PFF is below the $PFF_{threshold}$
  - evict at least two pages from the resident set
    - on to make space for the new one and one to reduce the RSS
- in any case load in the page
- to avoid oscillations usually two distinct thresholds are used: PFF_{max} and PFF_{min}
  - $PFF_{max} > PFF_{min}$

PFF policy

- it may used with page buffering
- it performs poorly in transient periods
  - RSS grows rapidly while changing from one locality to another
  - big RSS trigger process suspension



variable-interval sampling WS

- divide the (memory or process) virtual time in intervals
  - during each interval RSS can only increase
  - at the end of each interval pages unused in the interval are evicted and use bits cleared
  - interval length *x* varies and decided for each interval
    - *L* maximum duration of an interval
    - *M* minimum duration of an interval
    - *Q* maximum number of page faults in an interval

variable-interval sampling WS

- as soon as $x>L$ the interval is finished RSS management is performed
- if the number of page faults is $>Q$
  - if $x<M$ wait till $x=M$
  - **if $x>M$ RSS management is performed** (parameters should be chosen so that this is the most frequent case)

when page faults occour frequently RSS management is performed frequently and unused pages are quickly evicted

Cleaning Policy

- Demand cleaning
  - A page is written out only when it has been selected for replacement
- Precleaning
  - Pages are written out in batches before selction for replacement

Cleaning Policy

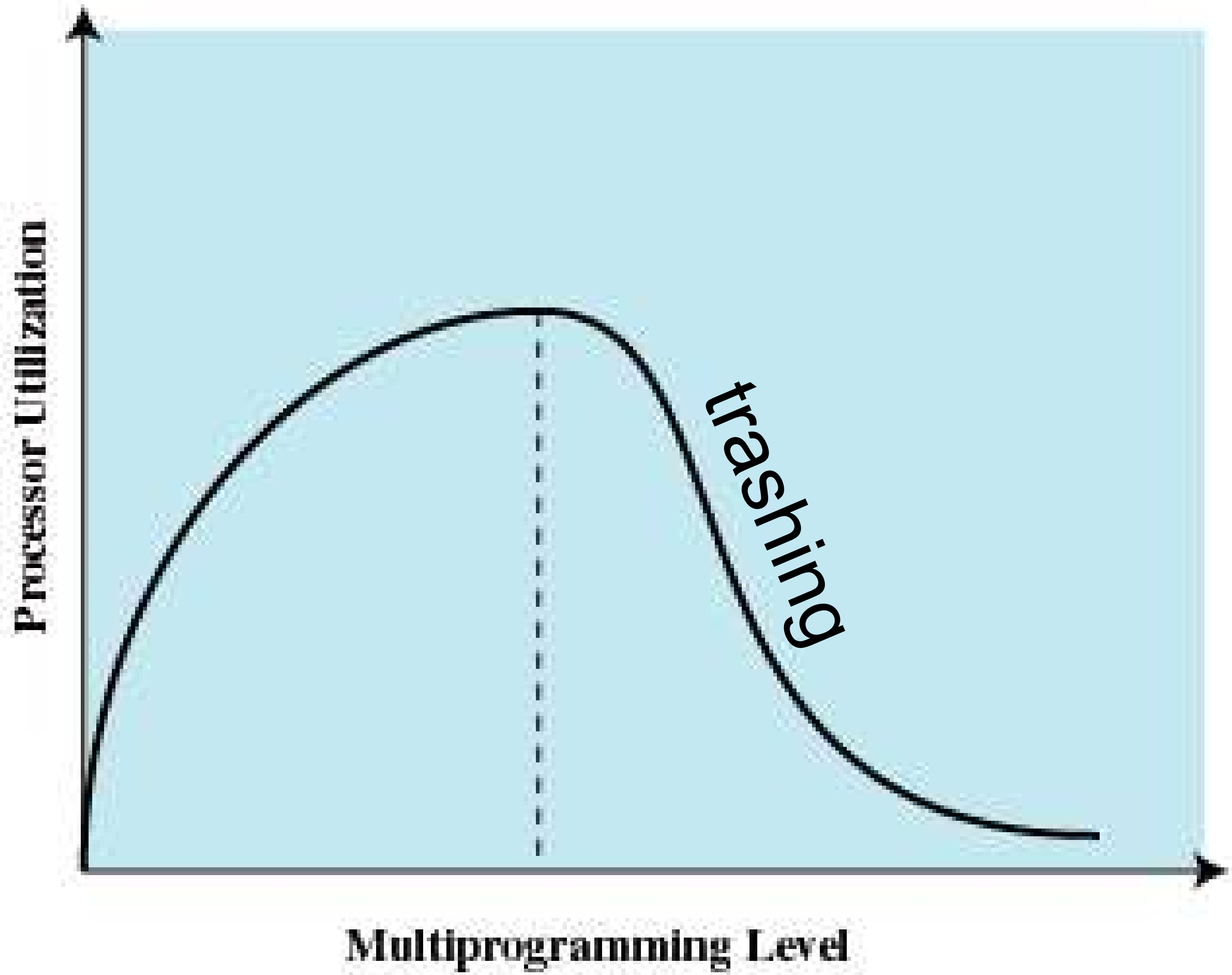
- Best approach uses page buffering
  - Replaced pages are placed in two lists
    - Modified and unmodified
  - Pages in the modified list are periodically written out in batches
  - Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

Load Control

- Desipte good design system may always trash!

- Determines the number of processes that will be resident in main memory
- Too few processes, many occasions when all processes will be blocked and much time will be spent in swapping
- Too many processes will lead to thrashing

Multiprogramming



Process Suspension

- Lowest priority process
- Faulting process
  - This process does not have its working set in main memory so it will be blocked anyway
- Last process activated
  - This process is least likely to have its working set resident

Process Suspension

- 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 window