Virtual Memory

the role of the operating system

not on the book

virtual memory vs. disk caching

- common objective
  - keep in main memory only data and/or programs that are really useful (frequently accessed)
- different action domain
  - virtual memory: processes, pages, segments
  - disk caching: files

	ram		disk
virtual memory	rarely used processes, pages or segments	→	swap area
disk caching	disk cache	←	frequently used parts of files

resident set

- the *resident set* (RS) of a process at a given time is the set of pages that are in main memory at that time
  - pages in RS content chages over time
  - size of RS may change over time or not, 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
    - if “prediction” is good, pages are already in memory when they are needed

Placement Policy

- Determines where in real memory a process piece (segment or page) is to reside
- Important in a segmentation system
  - see memory allocation approaches
- Paging: MMU hardware performs address translation
  - placement policy is irrelevant
  - in practice hw may impose some constraint

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 the 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

pager or swapper

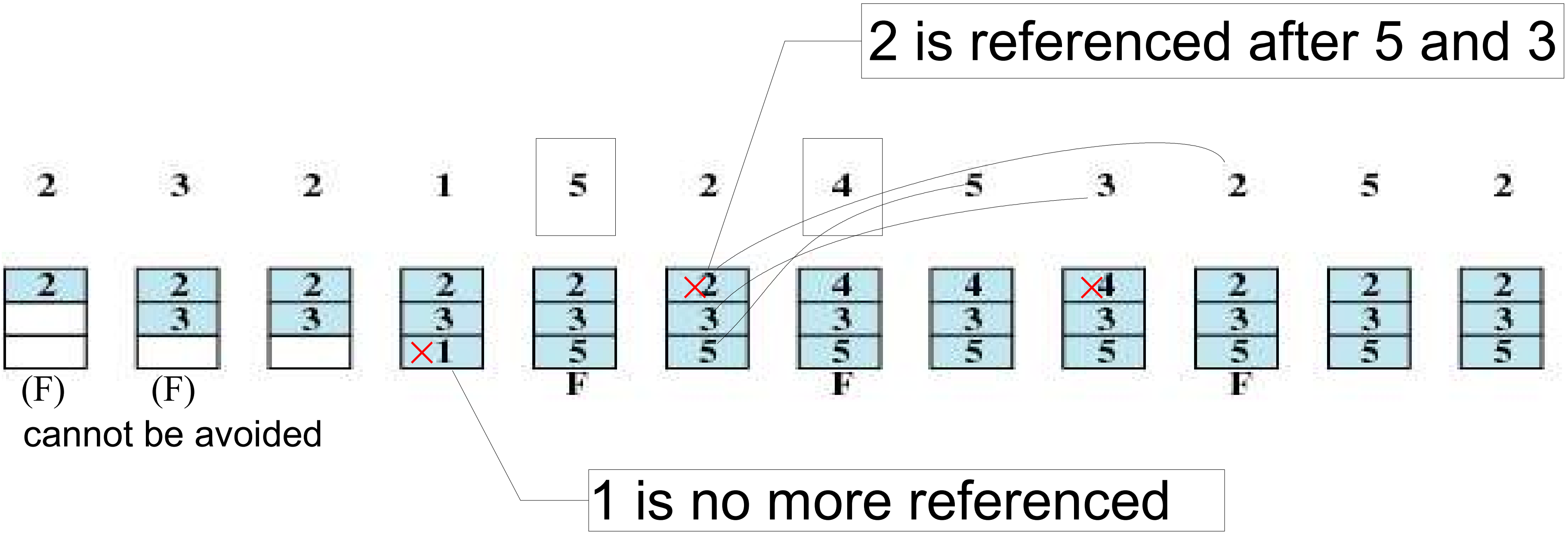
- the part of the kernel that manage the RS of the processes is called *pager* or *swapper*.
- it implements the replacement policy
  - page replacement is the most critical problem to solve for virtual memory efficiency/efficacy

Basic Replacement Algorithms/Policies

- **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

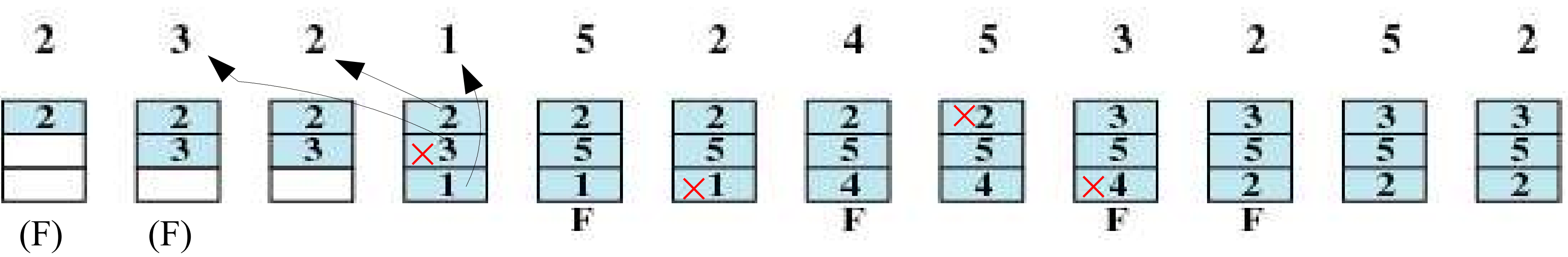
- page references stream:
  2 3 2 1 5 2 4 5 3 2 5 2
- 3 frames are available



Basic Replacement Algorithms/Policies

- **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 is 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

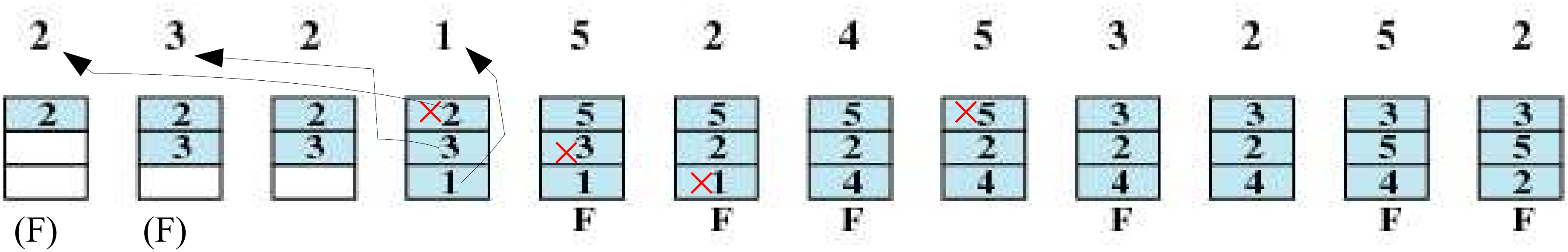


cannot be avoided

Basic Replacement Algorithms/Policies

- **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

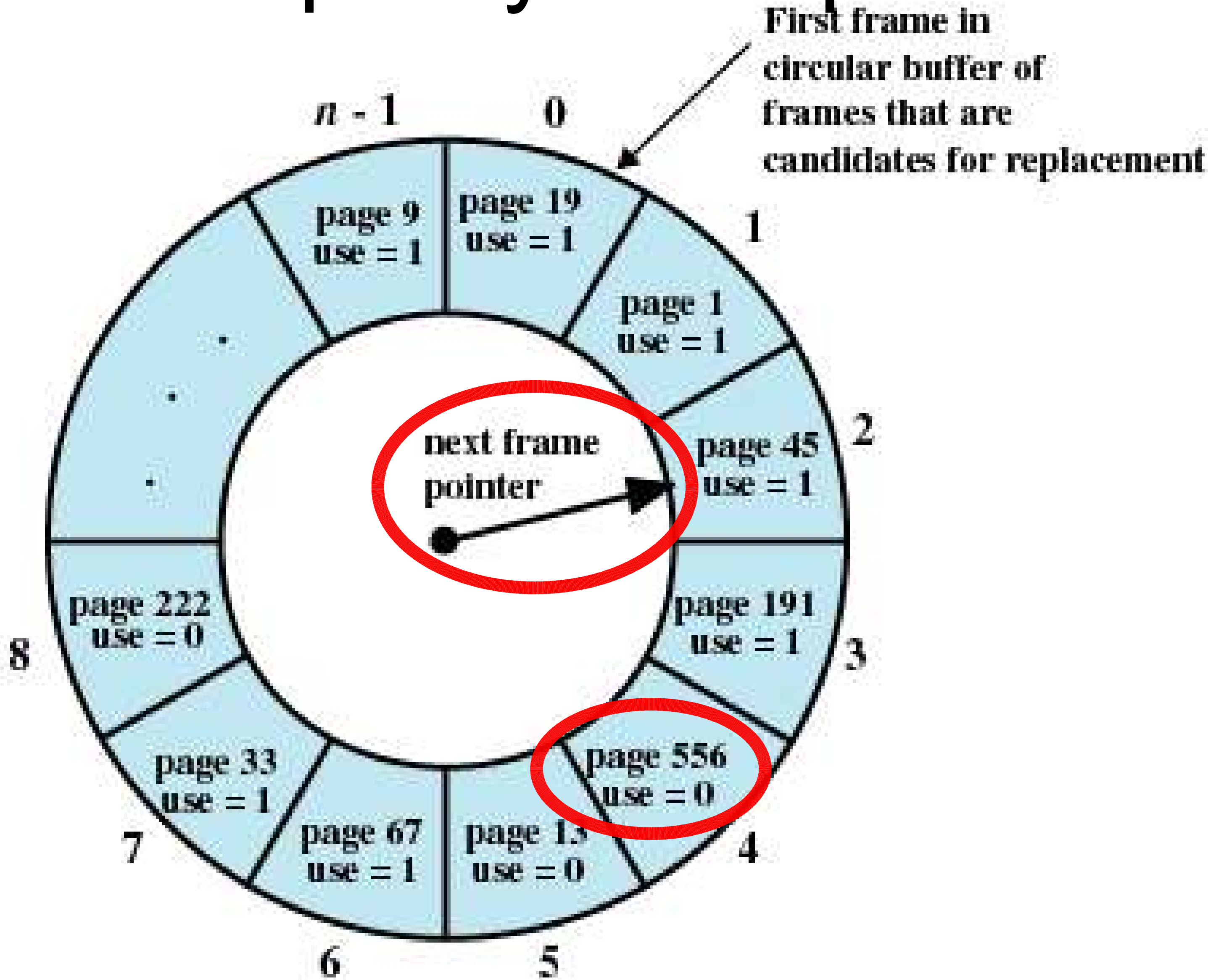


cannot be avoided

Basic Replacement Algorithms/Policies

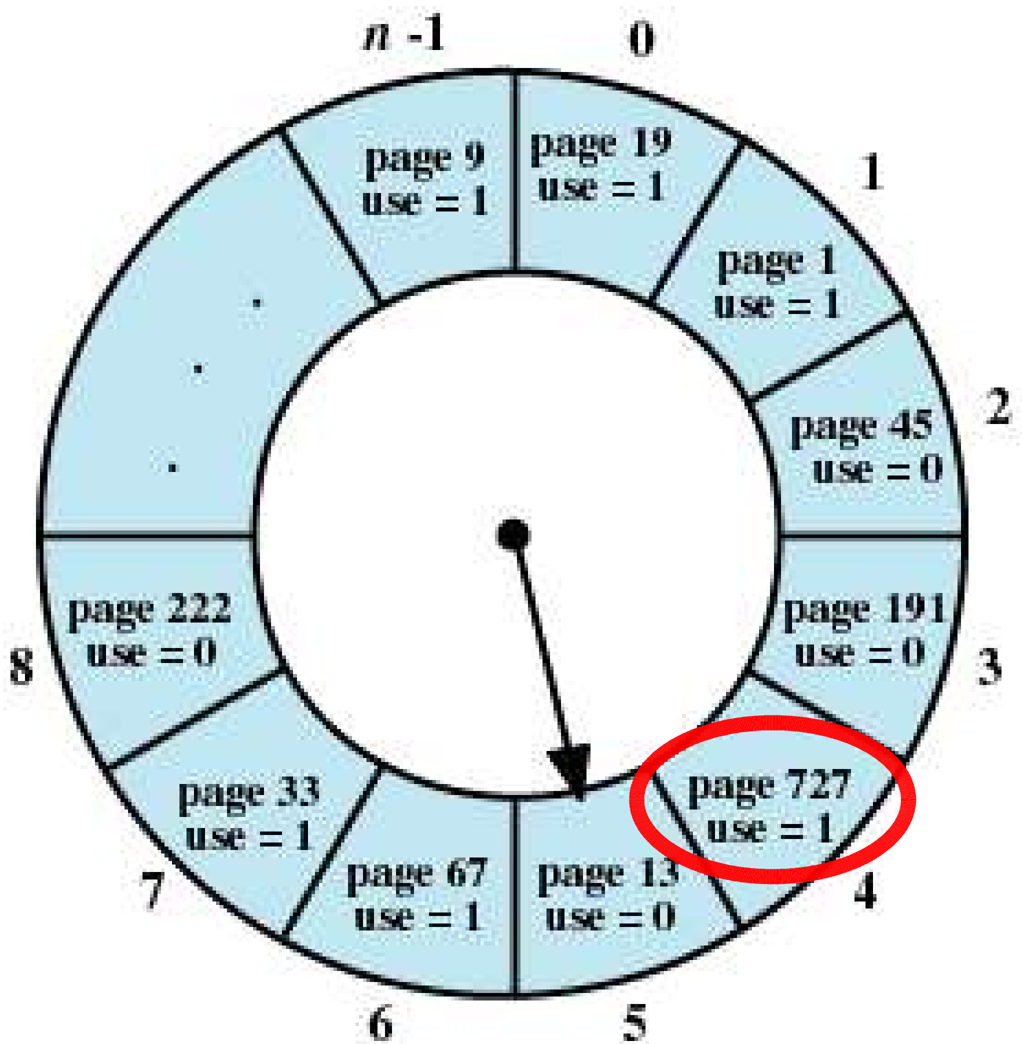
- **Clock Policy (second chance)**
  - one additional for each page bit called a use bit
  - set use=1
    - when a page is first loaded in memory
    - each time a page is referenced
  - when it is time to replace a page scan the frames...
    - the first frame encountered with use=0 is replaced
  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

clock policy example



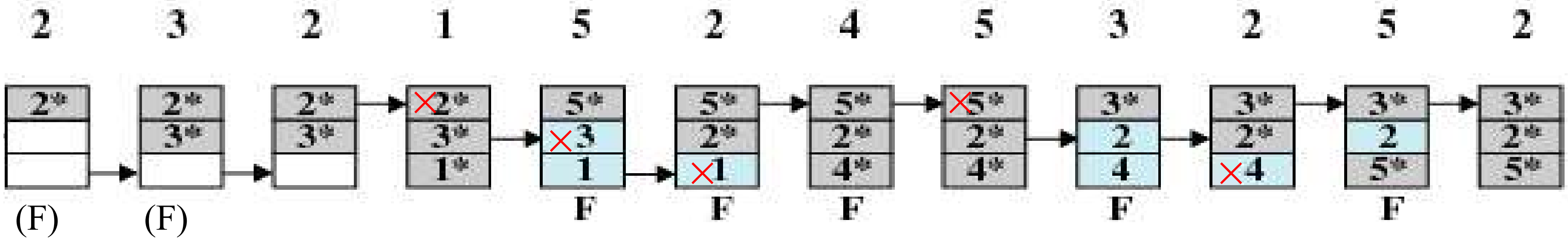
(a) State of buffer just prior to a page replacement

clock policy example



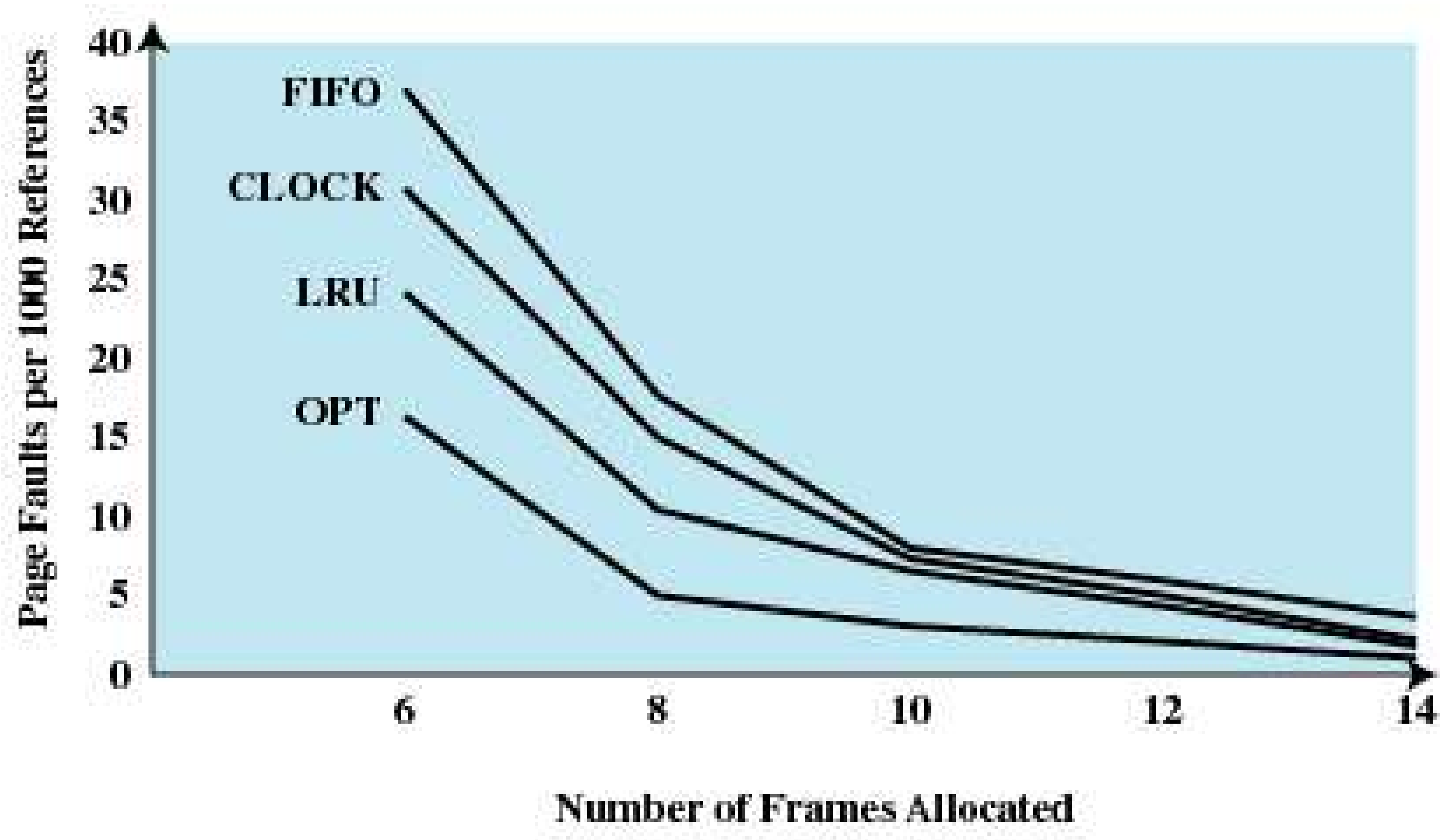
(b) State of buffer just after the next page replacement

clock policy example



cannot be avoided

comparison of replacement algorithms



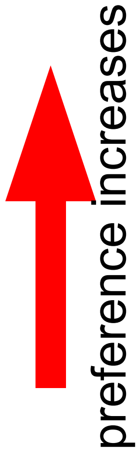
not on the book

CLOCK approximates LRU

- for each instance of CLOCK consider 2 sets
  - A: recently used pages (pages with use=1)
  - B: not recently used pages (pages with use=0)
- each time clock arm is moved a page is demoted from A to B
  - which one is quite arbitrary, depends of the position of the arm
- a page is promoted from B to A when it is accessed

CLOCK with “modified” bit

- 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 with “modified” bit

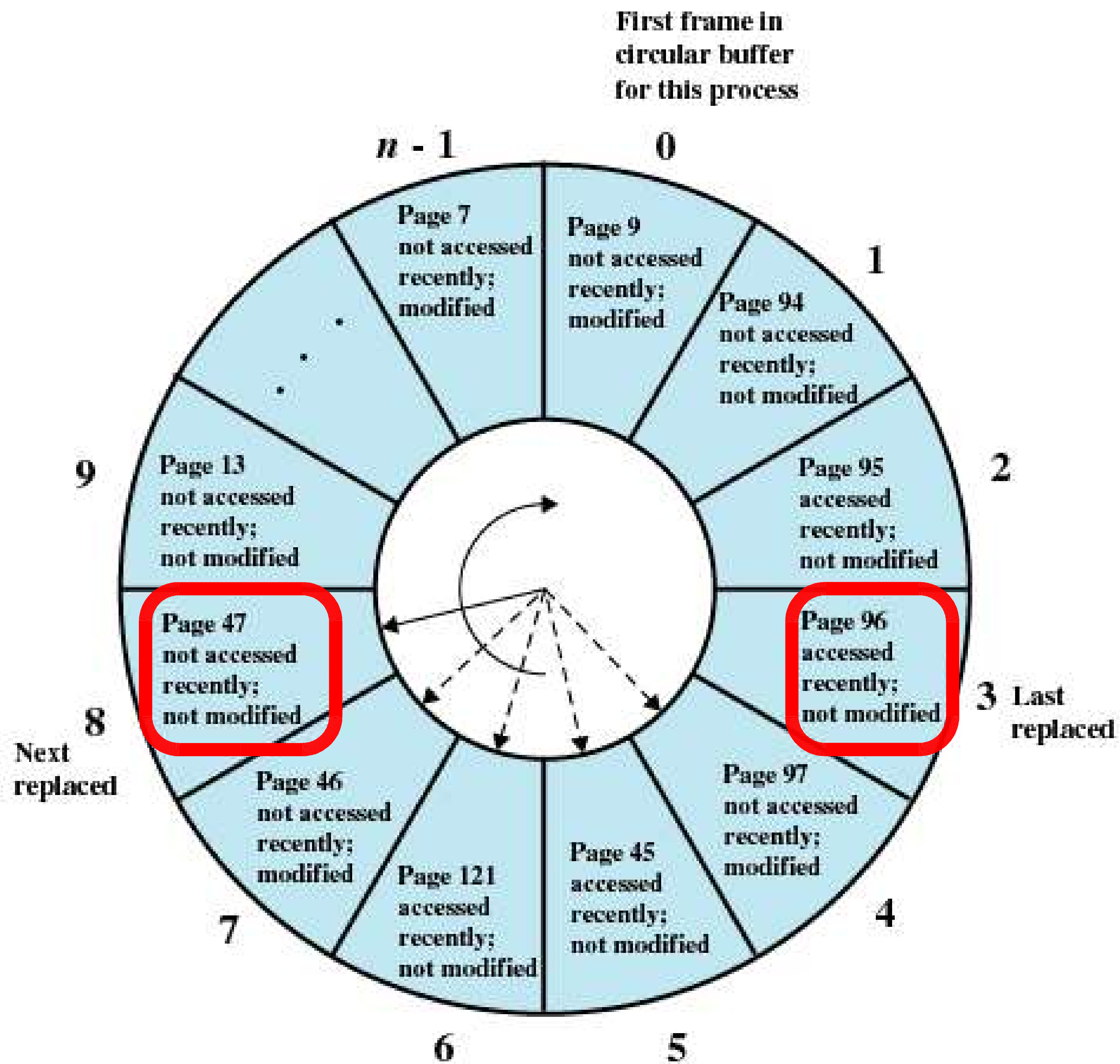
1 look for frames not accessed recently and not modified (use=0, mod=0)

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 with “modified” bit



not on the book

aging policy

(da Tannenbaum)

- for each age keeps an age “estimator”
  - the less is the value the older is the page
- it periodically sweeps all pages...
  - scans use bits and modifis estimator for each page
    - example: for page *p* shift right (that is divide by two) and insert the value of use bit for *p* as leftmost bit
      - it records the situation of the use bits for the last (e.g. 8) sweeps
    - theoretically, more complex extimators may be used
  - clear all use bits to record page usage for the next sweep
- evict pages starting from older ones
  - that is, those that have a lower estimator

not on the book

aging policy

version with right shift estimator

use bit for each page during at the sweep istant

time → sweep, sweep, sweep, sweep

	(a)	(b)	(c)	(d)	(e)
use bits (pages 0–5)	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

oldest pages at a certain instant

not on
the book

aging approximates LRU

- ages are quantized in time
  - many references between two sweeps are counted once
  - aging policy is much less precise than LRU
- very old references are forgotten
  - when an estimator reach zero it remains unchanged
  - impossible to discriminate among pages that were not referenced for very long time
    - LRU always maintains all the information it needs

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

Page Buffering

- if the page is claimed again it may be given to the process without any access to secondary memory
- we have a page fault but with very small overhead
  - no disk reading
  - just update data structures in main memory
    - page buffer → RS of the process

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, 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

fixed allocation, global scope

- not possible

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 a certain 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.

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

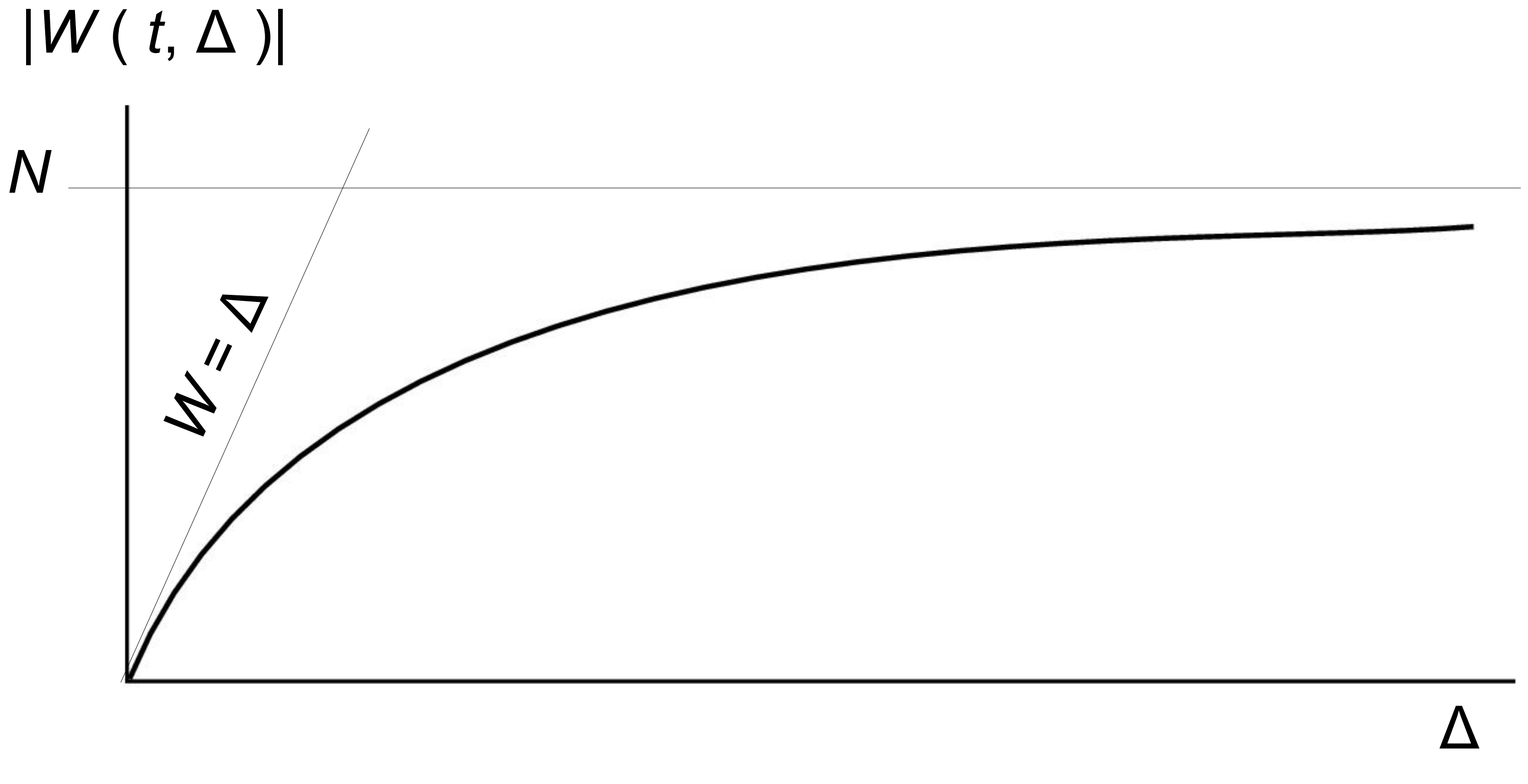
upper bound for the size of *W*

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

N number of pages in the process image

working set

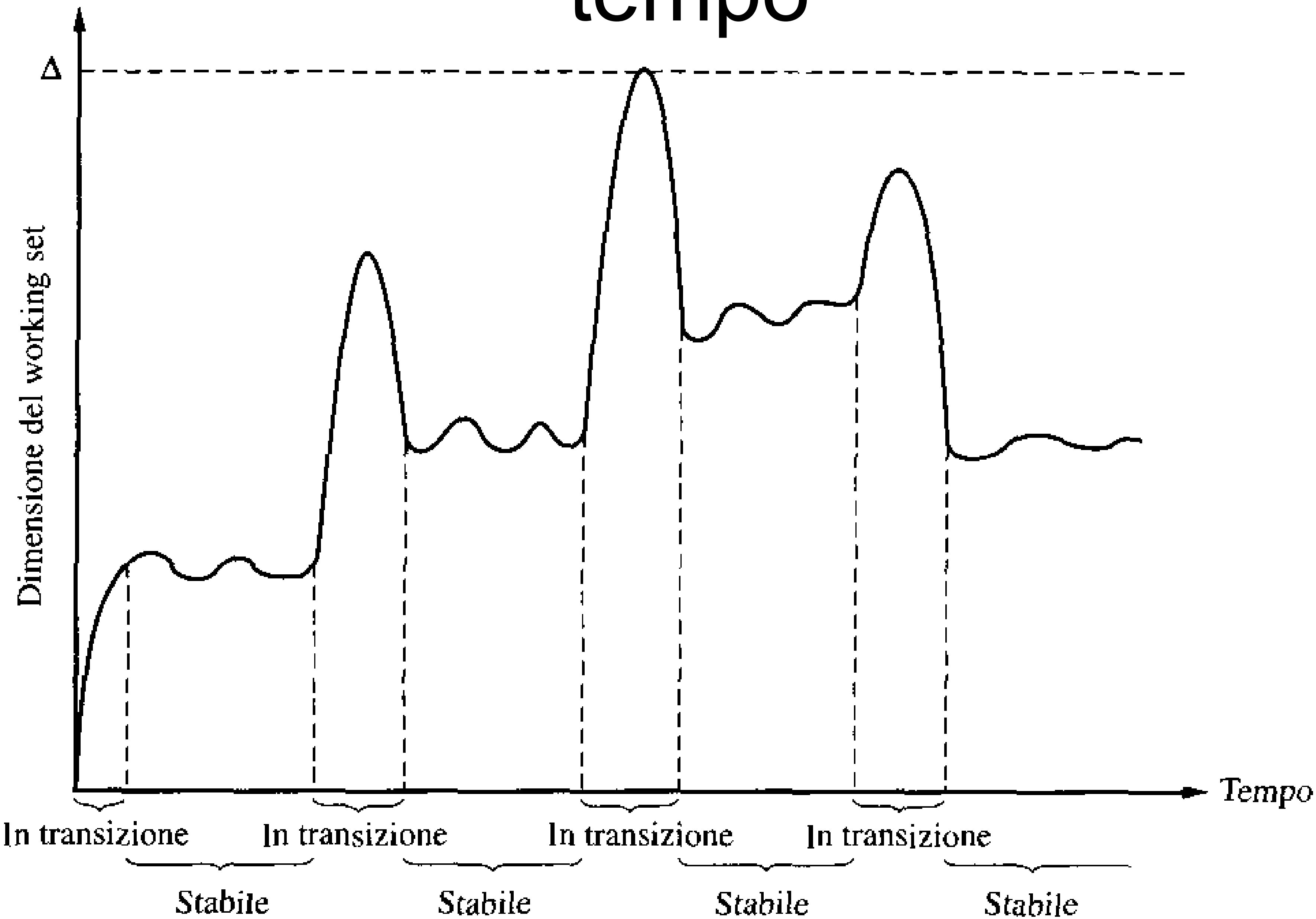
- values of $|W(t, \Delta)|$ varying Δ for t fixed and $t \gg N$



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

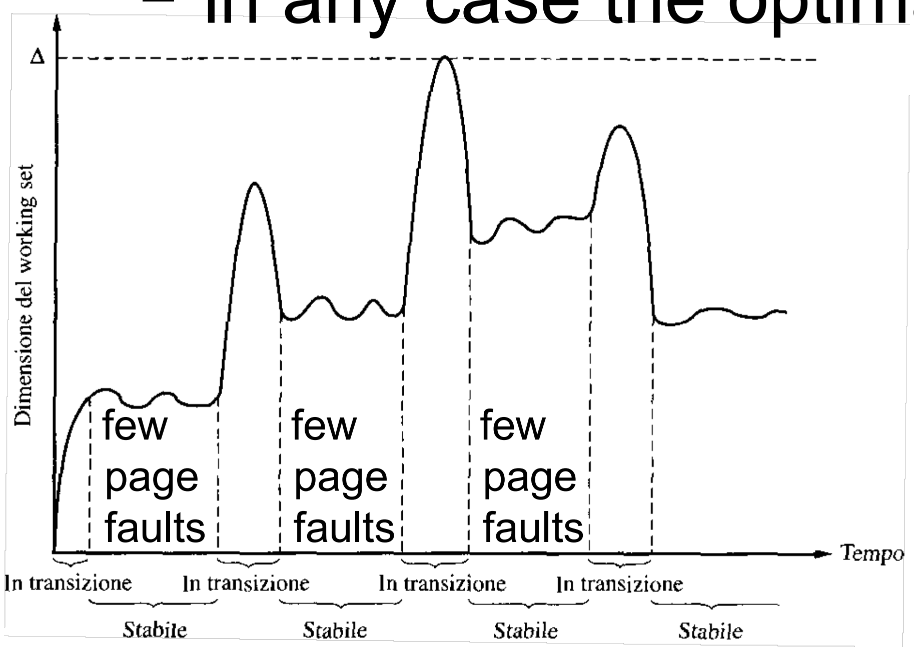


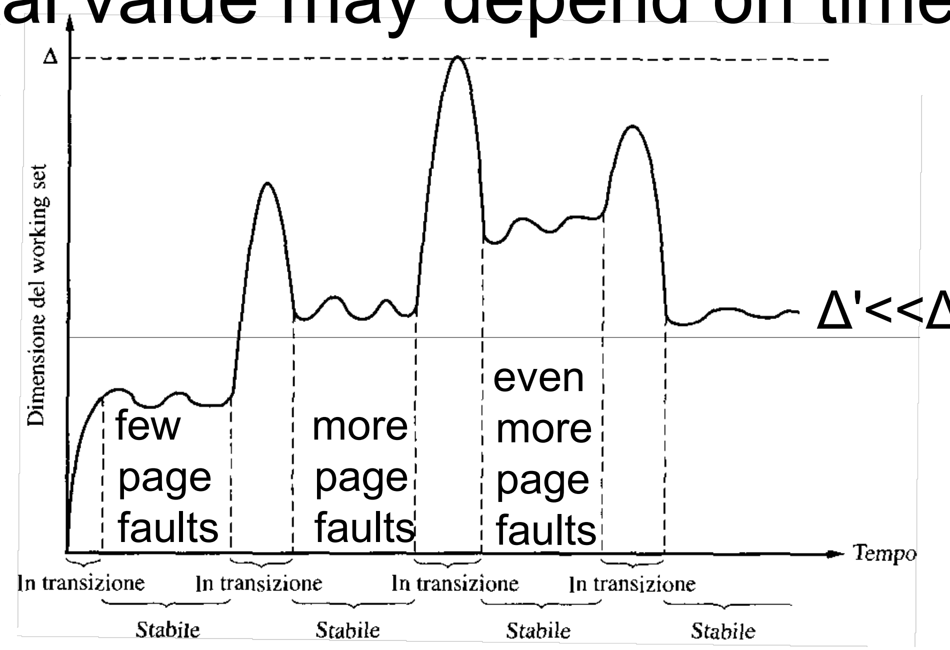
our goal

- ideally we would like to have always the working set of each process in memory (RS=WS, for a fixed Δ)
- WS (theoretical) strategy
  - monitor the WS of each process
  - update the RS according to the WS
    - page faults add pages to WS (and to RS)
    - periodically remove pages of the resident set that are not in the WS. In other words, LRU with variable resident set size.

working set strategy: problems

- optimal Δ?
  - larger Δ → less page faults and larger |W|
  - trade-off between number of page faults and WS size!
  - in any case the optimal value may depend on time





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

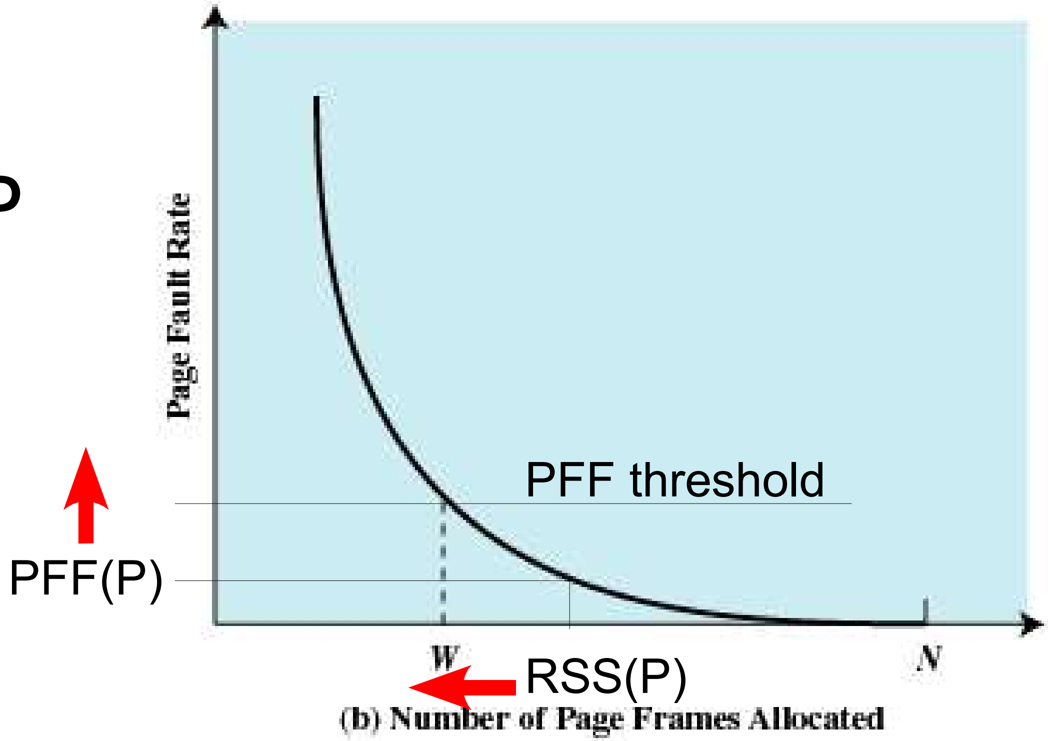
WS strategy approximation

- consider the frequency of page faults for a process (PFF)
- if the RS size of the process is larger than the WS size, PFF is low
- if the RS size of the process is smaller than the WS size, PFF is high

- we can use PFF to estimate the relationship between RS size and WS size

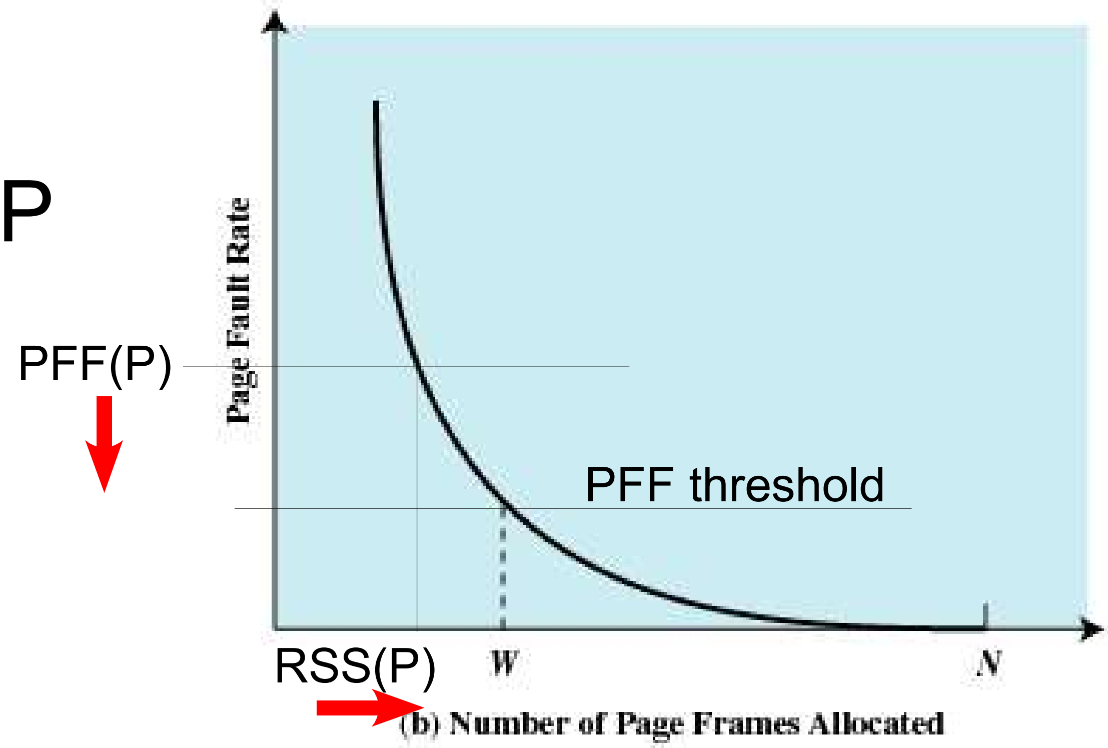
page fault frequency (PFF)

- 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



not on
the book

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

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

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

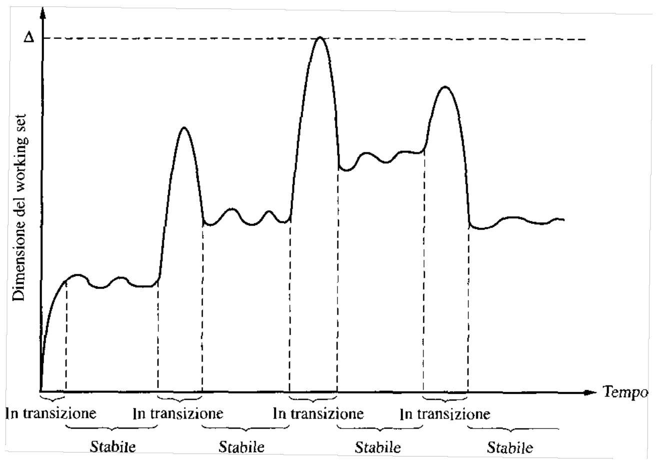
- decide action on estimated PFF

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
    - one 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 be 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



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

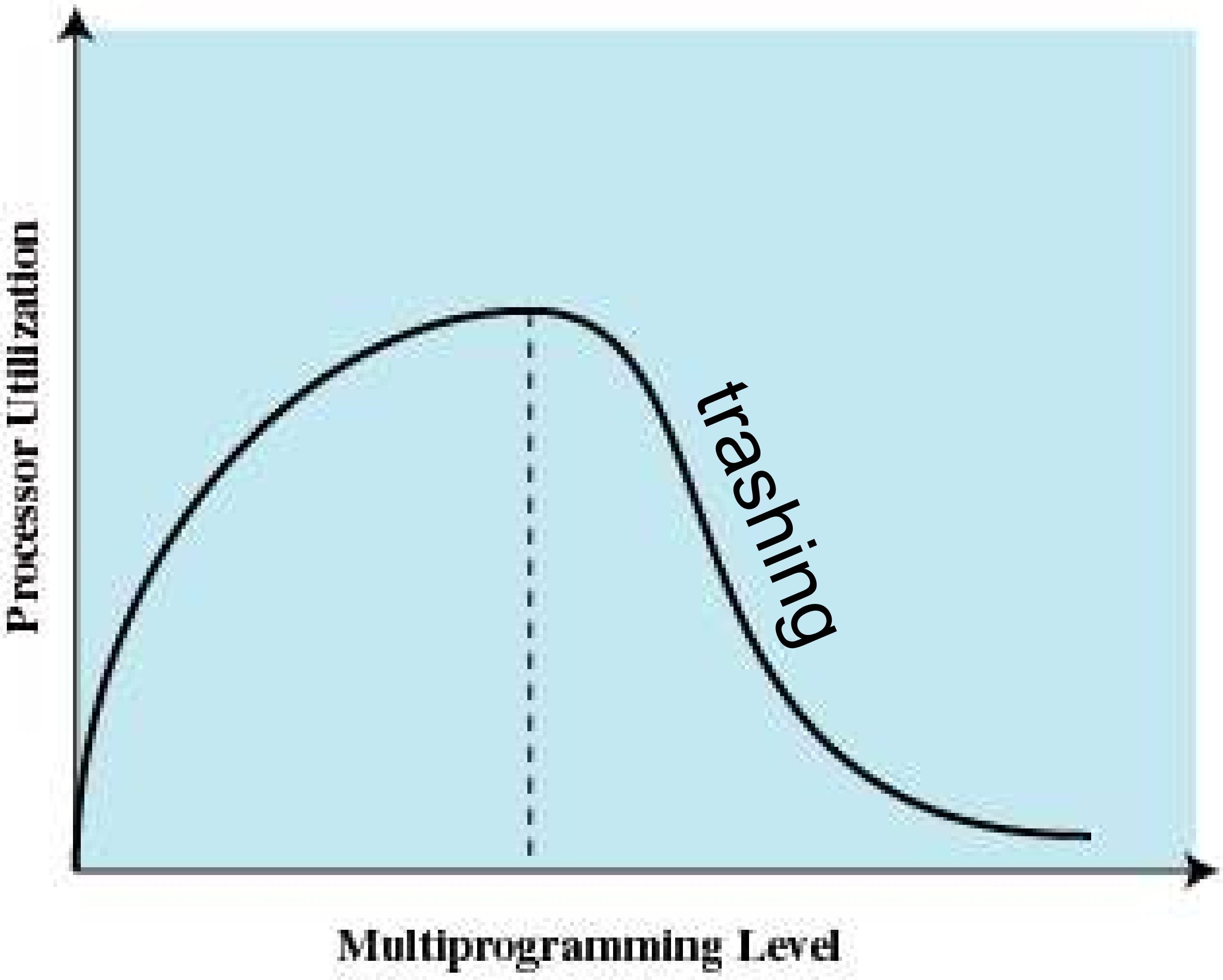
Cleaning Policy

- 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