Virtual Memory

not on
the book

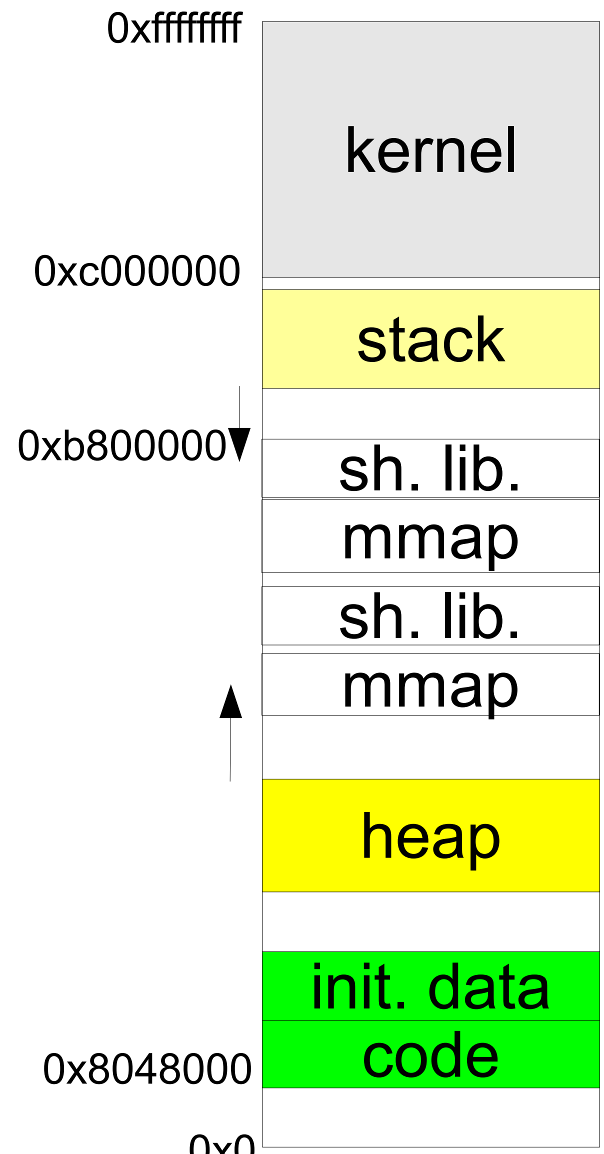
memory: the point of view of the process

- structural
  - one process address space divided in “legal” memory **regions**
    - for code (executable and libraries) and data (heap and stack)
    - rights (rwx)
    - regions can be shared (libraries, IPC, threads, fork - copy on write)
- behavioral
  - access to memory using machine language
  - regions: creation and change (mmap, brk, fork)
  - sharing (libraries, IPC, threads, fork - copy on write)

not on the book

process address space in linux

- many regions:
  - kernel (forbidden)
  - stack (rw)
  - code (rx)
  - init data (rw)
  - heap (rw, can grow)
  - many other
    - shared libraries
    - *memory mapped files*
    - *etc.*
- cat /proc/*pid*/maps



not on the book

advantages of virtual memory

- a process may be larger than all of main memory
- more processes may be maintained in main memory
  - only load in some of the pieces of each process
- with so many processes in main memory
  - in interactive systems users may run many applications, interfaces, etc.
  - it is very likely a process will be in the Ready state at any particular time
- **Resident set** - portion of process that is in main memory at a certain instant

virtual memory hw support

- hardware support
  - typically “paging”
    - memory references are dynamically translated into physical addresses at run time (by the hardware)
    - no relocation problems
- support for pages that are not in memory
  - “page not present” flag in page table entry
  - special interrupt to manage the situation
    - page fault

not on the book

page fault

- **page fault -** interrupt generated when a process access a memory address that is not in main memory
- the operating system places the process in a blocking state
  - the process is waiting its page from disk
  - this is equivalent to a blocking I/O request
  - the process that generated the page fault is placed in blocked state
  - another process is scheduled/dispatched
  - an interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

not on
the book

page fault

- major page fault
  - when input from disk is needed
- minor page fault
  - when input from disk is not needed
  - eg.
    - new free memory allocation (syscall brk, mmap)
      - memory allocation create a region, does not allocate a frame!
    - for same reason the page is not in the resident set of the process but it is in a frame in main memory
      - page buffering (we will see it)

no miracles: thrashing

- when physical memory is too short with respect of processes memory demand
- swapping out a piece of a process just before that piece is needed
  - if this happens frequently a lot of I/O is needed
  - at the extreme point all processes are waiting for their pages from the disk
- the processor has nothing to execute
- the disk is overbusy transferring pages

fetch policy

- determines when a page should be brought into memory
- *prepaging* brings in more pages than needed
  - if "prediction" is good, pages are already in memory when they are needed
  - rarely used
- **demand paging** only brings pages into main memory when really needed
  - save memory
  - many page faults during process start up
  - often used

Placement Policy

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

not on the book

eviction policy

- the strategy used by the OS to choose pages to take out of the RS of the processes
- a good page to evict will not be accessed in the near future
- the eviction strategy is the way the OS uses to predict the future
- long research history
  - optimal, lru, fifo, clock, aging belady anomaly, competitive on-line algorithms, etc....
  - we will see many eviction strategies

not on
the book

eviction policy

- a good eviction strategy greatly speed up whole system
- WARNING:
  - trashing is still there!
  - being smart allows to run more processes before system goes thrashing
  - anyway there always be a limit after that system goes thrashing
    - even for OPTimal eviction policy (see after)

cleaning policy

- before eviction of a modified (dirty) page this has to be written to disk
- demand cleaning
  - A page is written out only when it has been selected for replacement
- precleaning
  - Pages are written out in batches before selection for replacement
  - e.g. when disk is idle

cleaning policy and page buffering

- system always keeps a small amount of free frames ready for re-assignment
- frames of evicted pages are added to one of two lists
  - free-clean frame list, if page has not been modified
  - free-dirty frame list, otherwise
- frames in the clean are ready-for re-assignment
- frames in the dirty list are periodically written out in batches and put into the clean list

Page Buffering

- suppose a page P is still contained in a free-frame (either clean or dirty)
  - nobody has overwritten it, yet
- P can be claimed again by its process
- P can be given to the process without any access to secondary memory
- we have a (minor) 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 as eviction policys

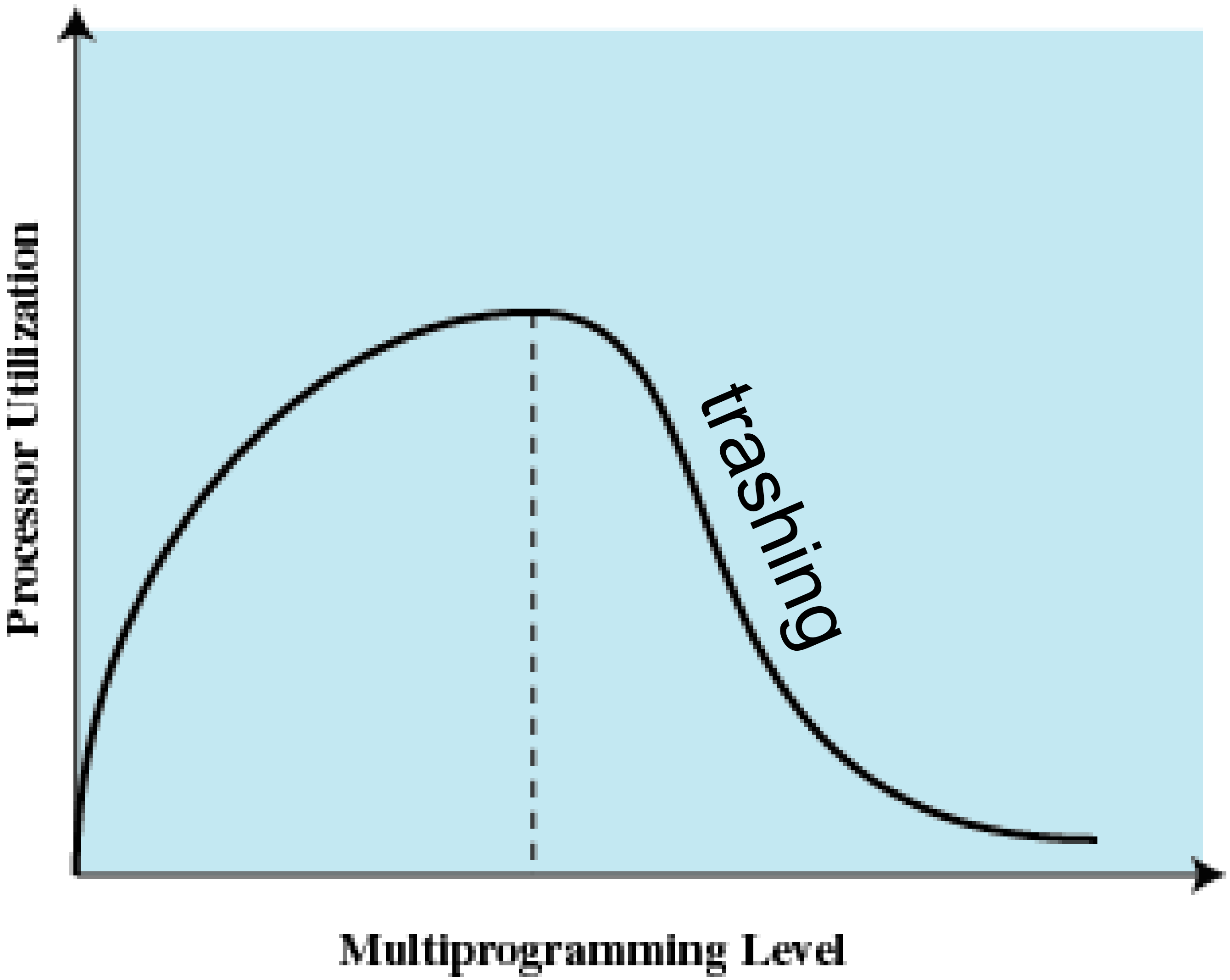
- page buffering “corrects” simple eviction policies implementing a sort of LRU eviction strategy
  - see after

Load Control

- Desipte good design (e.g. good eviction policy) system may always trash!

- Determines the number of processes that will be resident in main memory
- Too many processes will lead to thrashing
- Too few processes, cpu under utilized

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

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

not on the book

virtual memory vs. disk caching

- disk cache needs to take mostly the same kinds of decisions as virtual memory
  - fetch, placement, eviction, cleaning
  - some files are used by more processes as some pages are shared by more processes
  - file parts are brought into memory “on demand” as in demand paging
- common solution: memory-mapped files
- it is a new i/o primitive
  - reads and writes are handled by ad-hoc caching

not on
the book

memory-mapped files

- a process can ask to see a part of a file as memory
  - unix syscall mmap(void *start_hint, size_t length, int protection, int flags, int fd, off_t offset)
- no input during the syscall, just creation of a new memory region
- page fault brings in memory what is needed
- cleaning write on disk what is changed
- reads and writes are not performed as syscall but as processor memory access: a lot faster!

not on
the book

memory-mapped files

- several kinds
  - read-only, shared, private, anonymous (mapped on swap area)
- typical usage
  - executable: demand paging, shared libraries
    - mmap called by dynamic linker which is the only think is loaded by execve syscall, it then mmap's the executable and all shared libraries change them a bit (private mmap)
  - efficient i/o based applications: e.g. DBMS

not on
the book

memory-mapped files

- drawbacks
  - need to read the page before writing
  - real write is preferomed on "cleaning" or unmapping of the file
    - unsuitable when user should have control of when something is written (eg. text editors, save...)
  - file size change unsupported

an example

protection

filename mapped

anonymous mapping

```
pizzonia@pisolo:~$ cat /proc/self/maps
08048000-0804f000 r-xp 00000000 08:03 6750220    /bin/cat
0804f000-08050000 rw-p 00006000 08:03 6750220    /bin/cat
08050000-08071000 rw-p 08050000 00:00 0          [heap]
b7dec000-b7ded000 rw-p b7dec000 00:00 0
b7ded000-b7f36000 r-xp 00000000 08:03 11796591   /lib/tls/i686/cmov/libc-2.7.so
b7f36000-b7f37000 r--p 00149000 08:03 11796591   /lib/tls/i686/cmov/libc-2.7.so
b7f37000-b7f39000 rw-p 001[illegible]a000 08:03 11796591   /l[illegible]/cmov/libc-2.7.so
b7f39000-b7f3c0[illegible]000 0[illegible]
b7f55000-b7f57[illegible]00 [illegible]
b7f57000-b7f58000 r-xp b7f57000 00:00 0          [vdso]
b7f58000-b7f72000 r-xp 00000000 08:03 7061540    /lib/ld-2.7.so
b7f72000-b7f74000 rw-p 00019000 08:03 7061540    /lib/ld-2.7.so
bfb78000-bfb8d000 rw-p bffeb000 00:00 0          [stack]
```

offset in the file

device

inode number

hw support for virtual memory

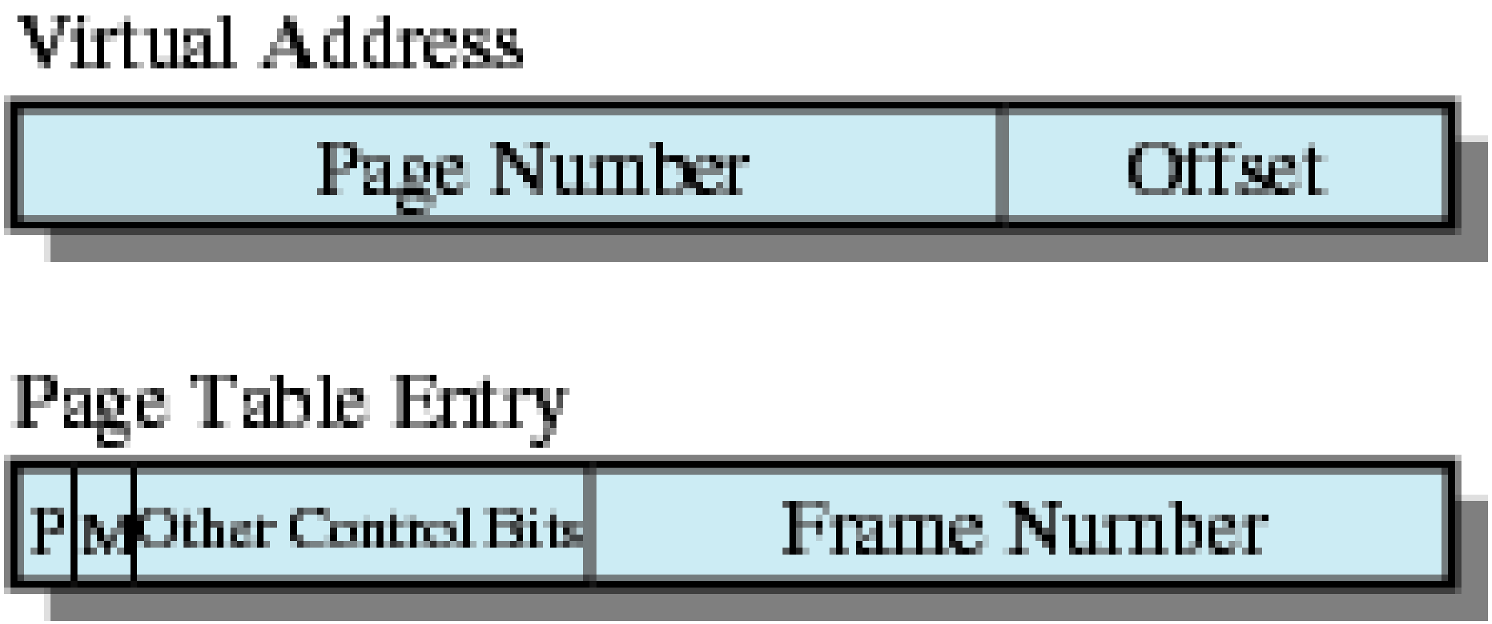
Hw Support Needed for Virtual Memory

- Hardware must support paging and/or segmentation...
  - ...plus indication of "page not resident"
- Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory
  - and decide which is the "best page" to evict
  - we will see that we need a few additional "bits" from the hw

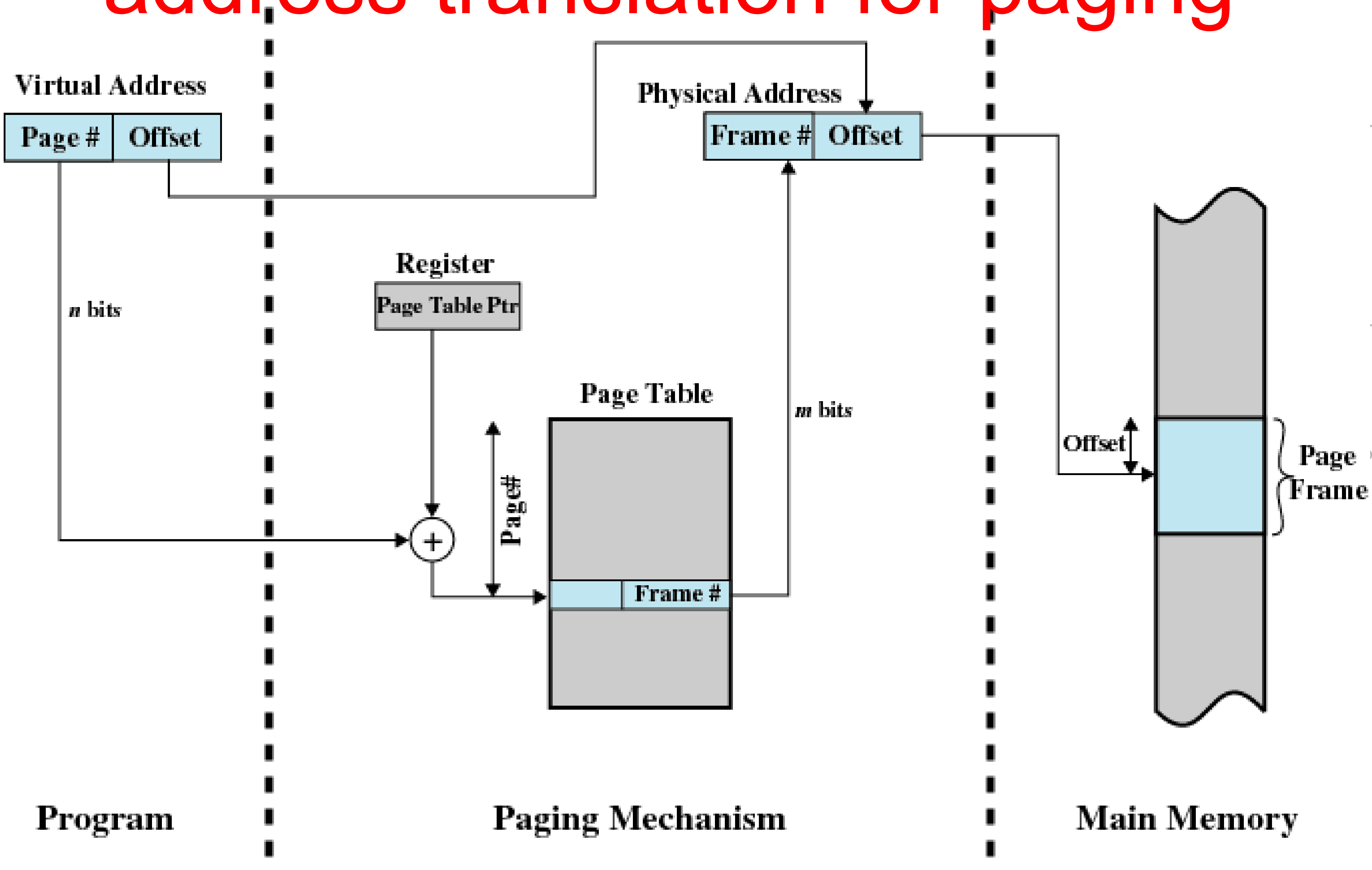
page table for virtual memory

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- An additional bit is needed to indicate whether the page is in main memory or not
- An additional bit is needed to indicate whethere the page has been altered since it was last loaded into main memory
  - no change → the frame does not have to be written to disk when page is evicted

Paging


Virtual Address
Page Number
Offset
Page Table Entry
P
M
Other Control Bits
Frame Number

address translation for paging



very big page tables

- what if a process use a limited number of small parts of the page table?
  - other parts may be not used at the moment or not used at all
  - a lot of memory wasted for unused page table entries
- page tables should be treated largely as part of the process image
- hierarchical page tables, inverted page tables

Two-Level Scheme for 32-bit Address (pentium like)

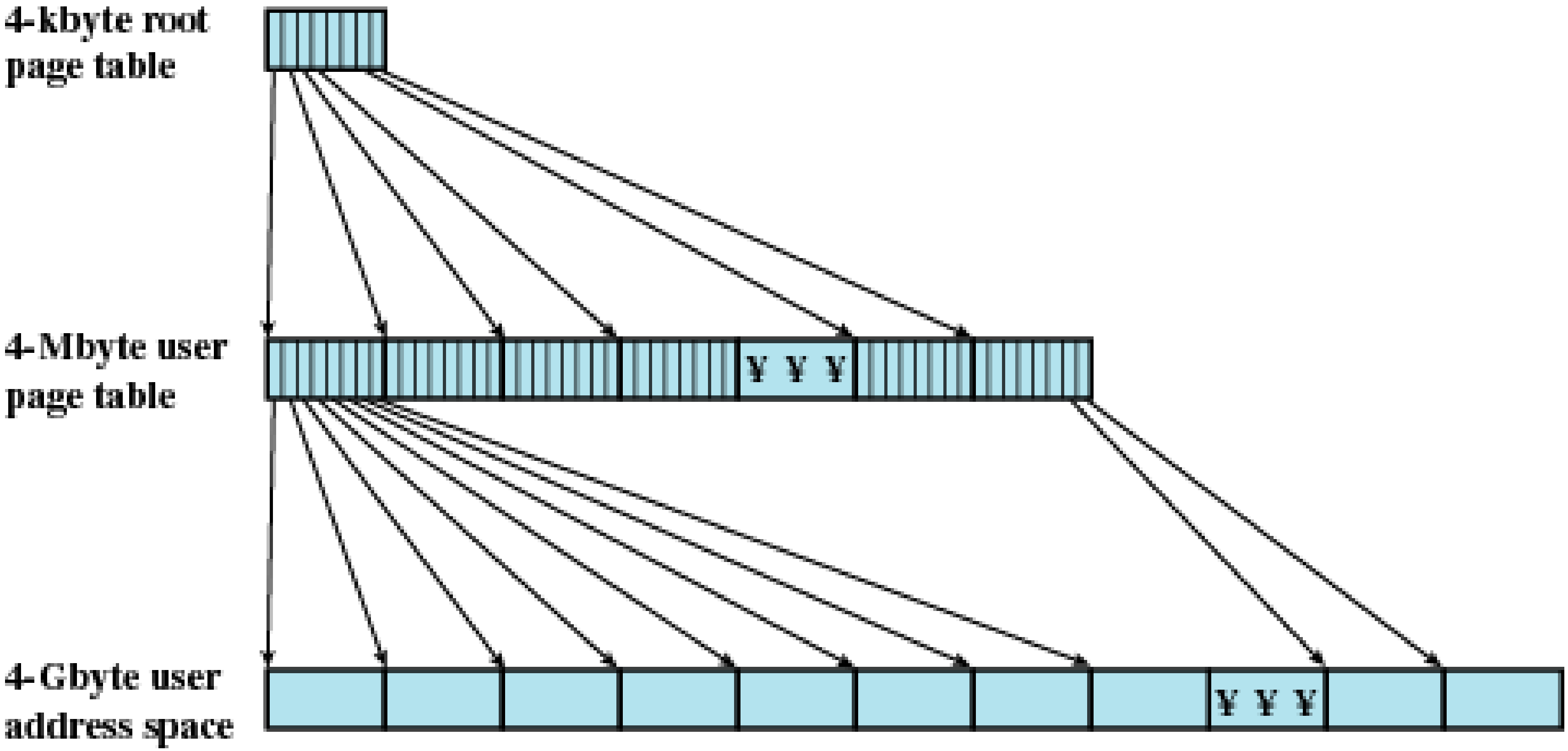
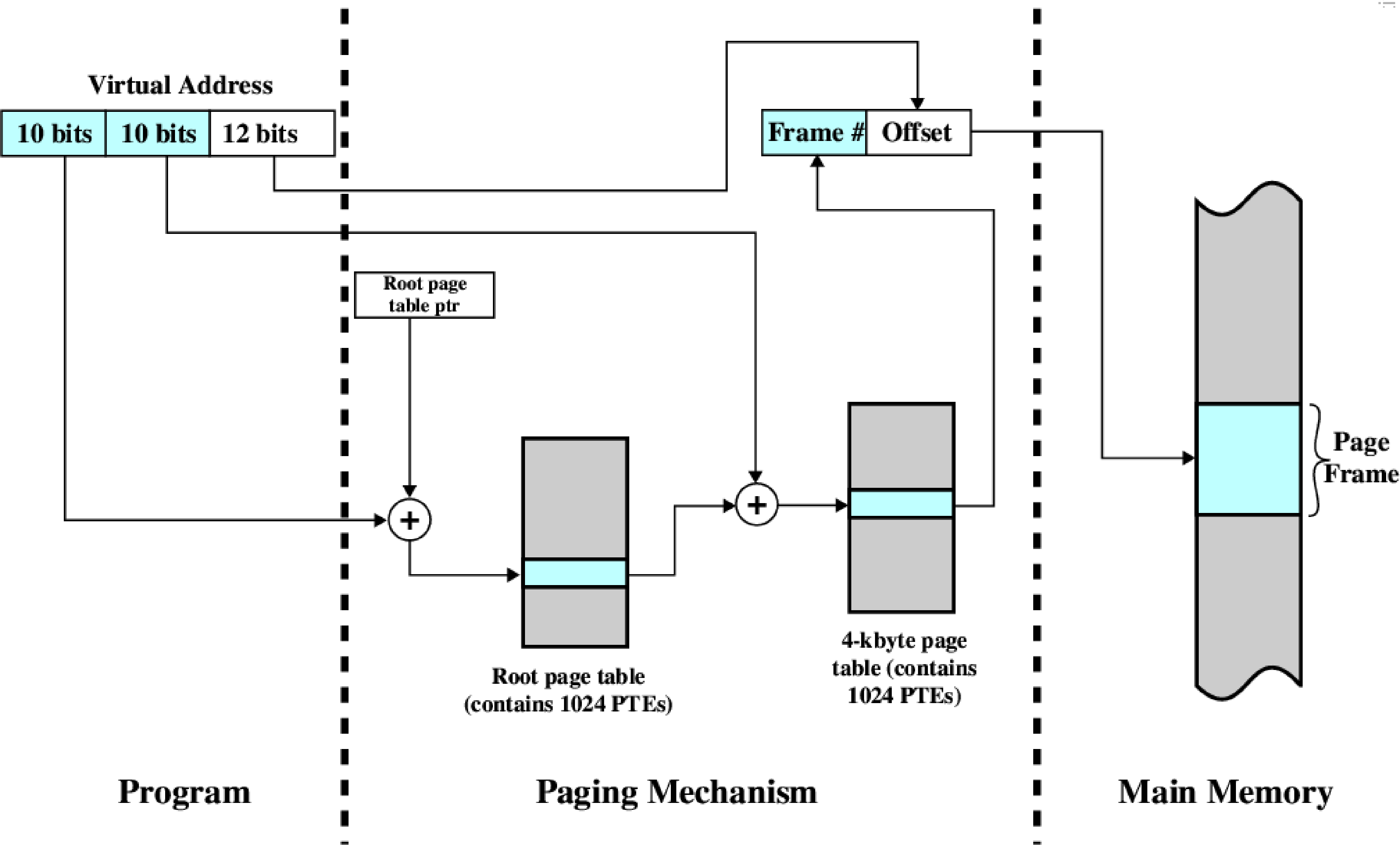


Figure 8.4 A Two-Level Hierarchical Page Table

address translation in a two-level schema



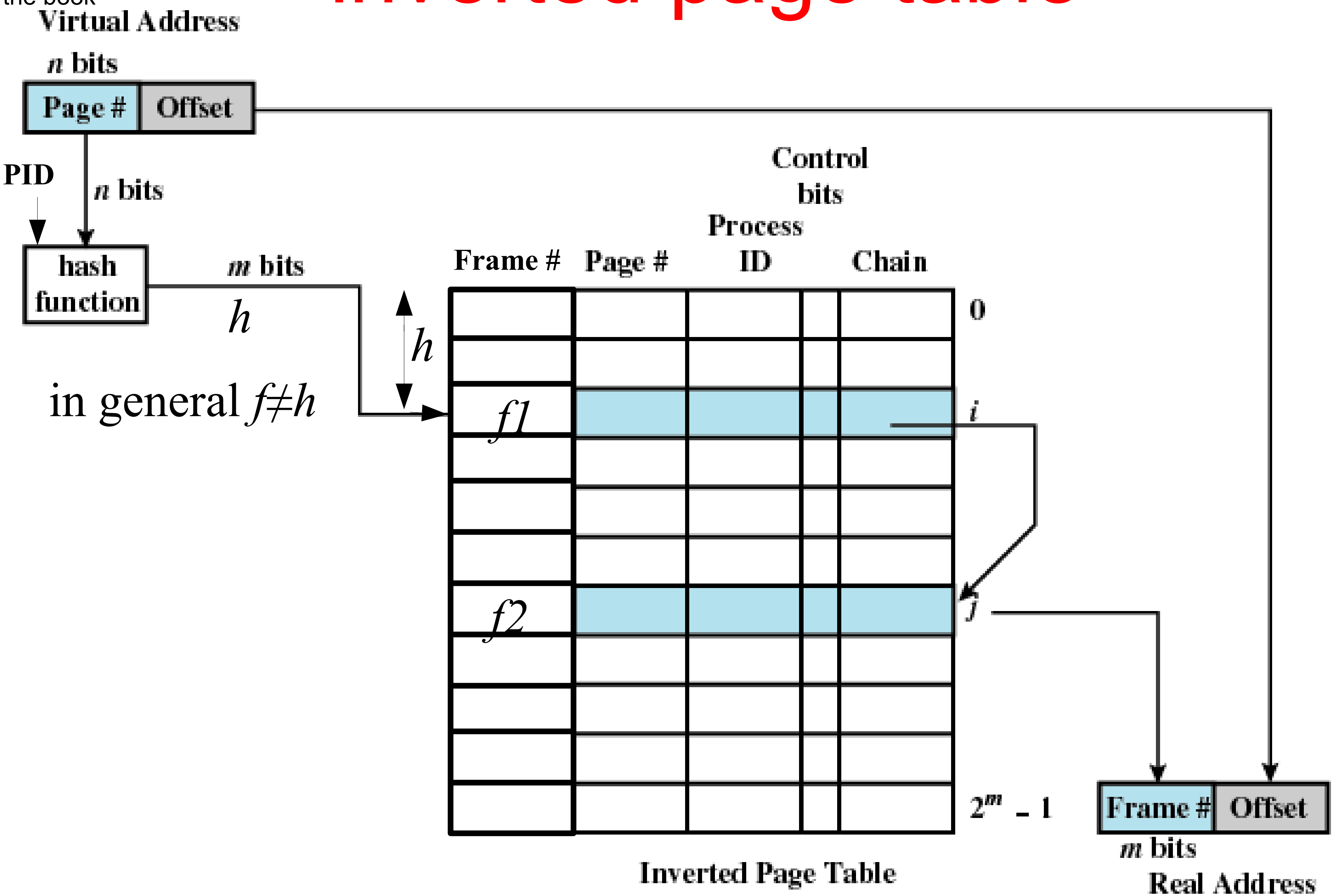
not on
the book

Inverted Page Table (IPT)

- page number portion of a virtual address and PID are mapped into a hash value
- the **hash value** points into the page table entry
  - entry contains info to check validity (pid and page#) since it may not be related to the process due to collision
    - collisions are solved by chaining
  - entry contains frame number
  - as many entries as the number of frames
- used by PowerPC, UltraSPARC, and Intel Itanium architecture

inverted page table

not on the book



not on the book

updating the IPT (OS)

- the frame h_1 is computed by hashing, read the table entry for h_1
- if h_1 is free
  - update the entry h_1 with pid/pagenumber/framenumber and set chain=0
- else
  - choose a new entry h_2 (e.g. by applying hashing again)
  - if h_2 is free, update the entry of h_2 with pid/pagenumber/framenumber, set chain=0 and set chain of the entry h_1 to point to the entry h_2
- if h_2 is occupied, iterate again possibly producing longer chains

not on the book

reading the IPT (CPU)

- compute the hash h_1
- if the entry for h_1 contains the right pid and page number, read the frame number from this entry and perform memory access
- otherwise follow the chain until find the right pid/pagenumber
- if chain end is reached, the page is not in memory
  - page fault or illegal memory access

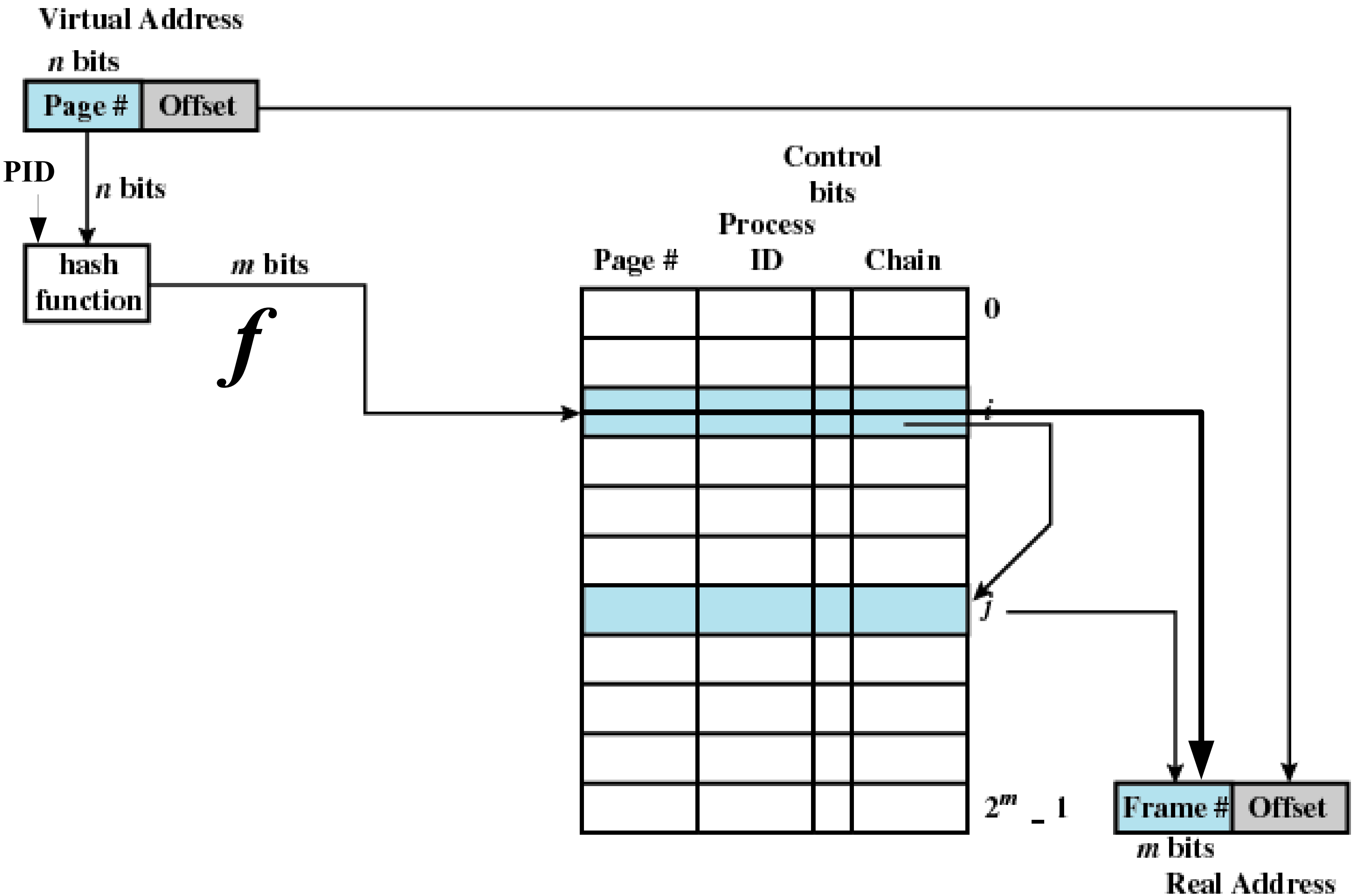
not on
the book

IPT in real architectures

- in real architectures the IPT does not have a frame# field
- the result of the hash function **is** the frame number!
- this constrains OSes to select the frame chosen by the hash function for hosting the page...
  - ...or to introduce a chain

IPT in real architectures

not on the book



not on
the book

IPT in real architectures

- IPT are used when virtual address space is really huge
- this happens in OS that...
  - ... run on 64 bits hw architecture
  - ... adopt a “single address space” model

not on
the book

SAS vs. MAS

- linux windows etc. are multiple address space OSes (MAS-OS)
  - each process occupies a distinct address spaces
- in single address space OSes (SAS-OS) processes occupies distinct portions of the same virtual address space
  - no page table switching is needed when switching process (but rights changes)
  - sharing of memory is easier
  - huge virtual address space is needed to host all processes!

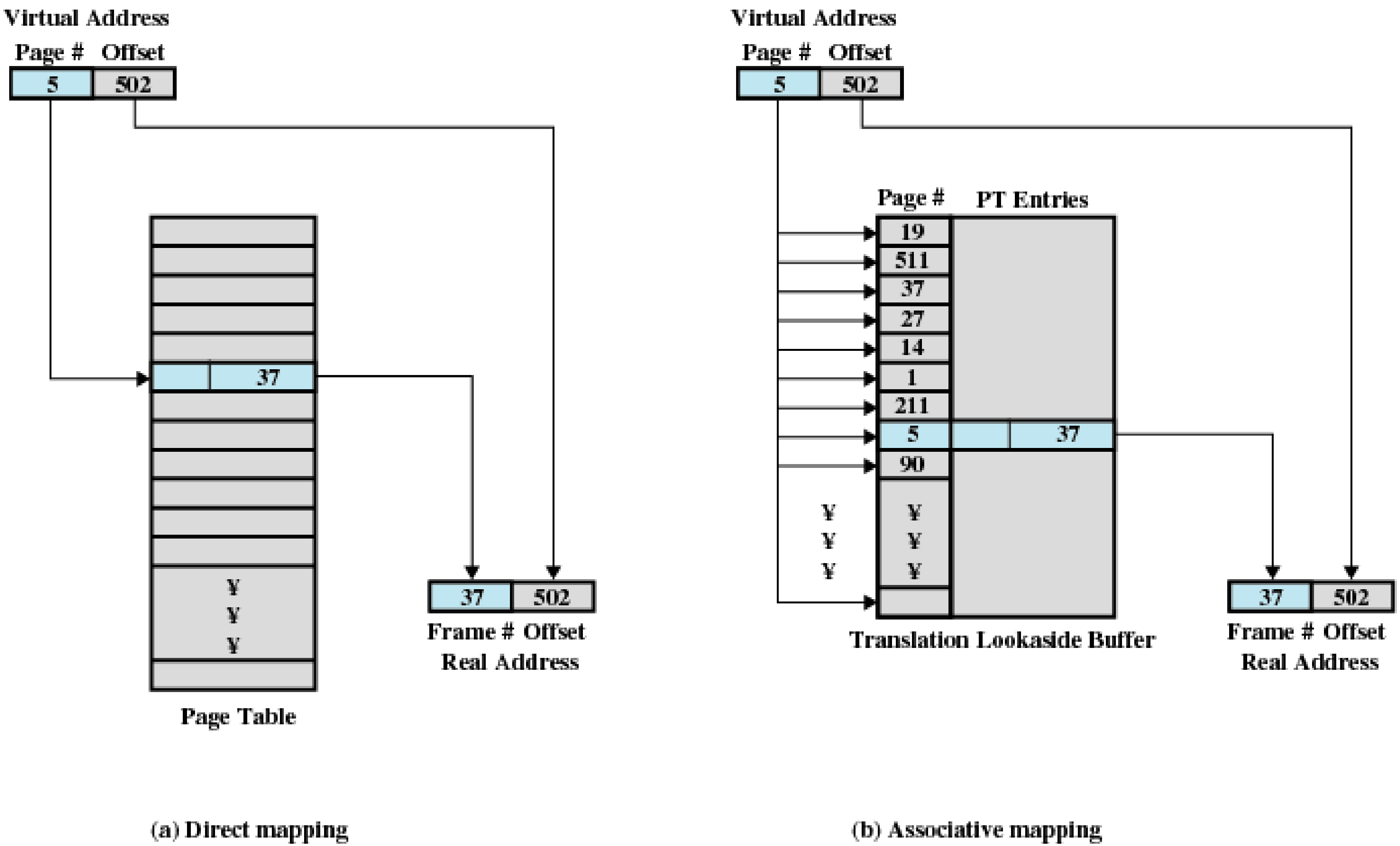
Translation Lookaside Buffer

- Each virtual memory reference can cause two (or more) physical memory accesses
  - One to fetch the page table entry
  - One to read/write the data
- To overcome this problem a **high-speed cache** is set up for page table entries
  - Called a Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer

- Contains page table entries that have been most recently used
- it performs an **associative mapping** between page numbers and page table entries

direct vs. associative mapping



Translation Lookaside Buffer

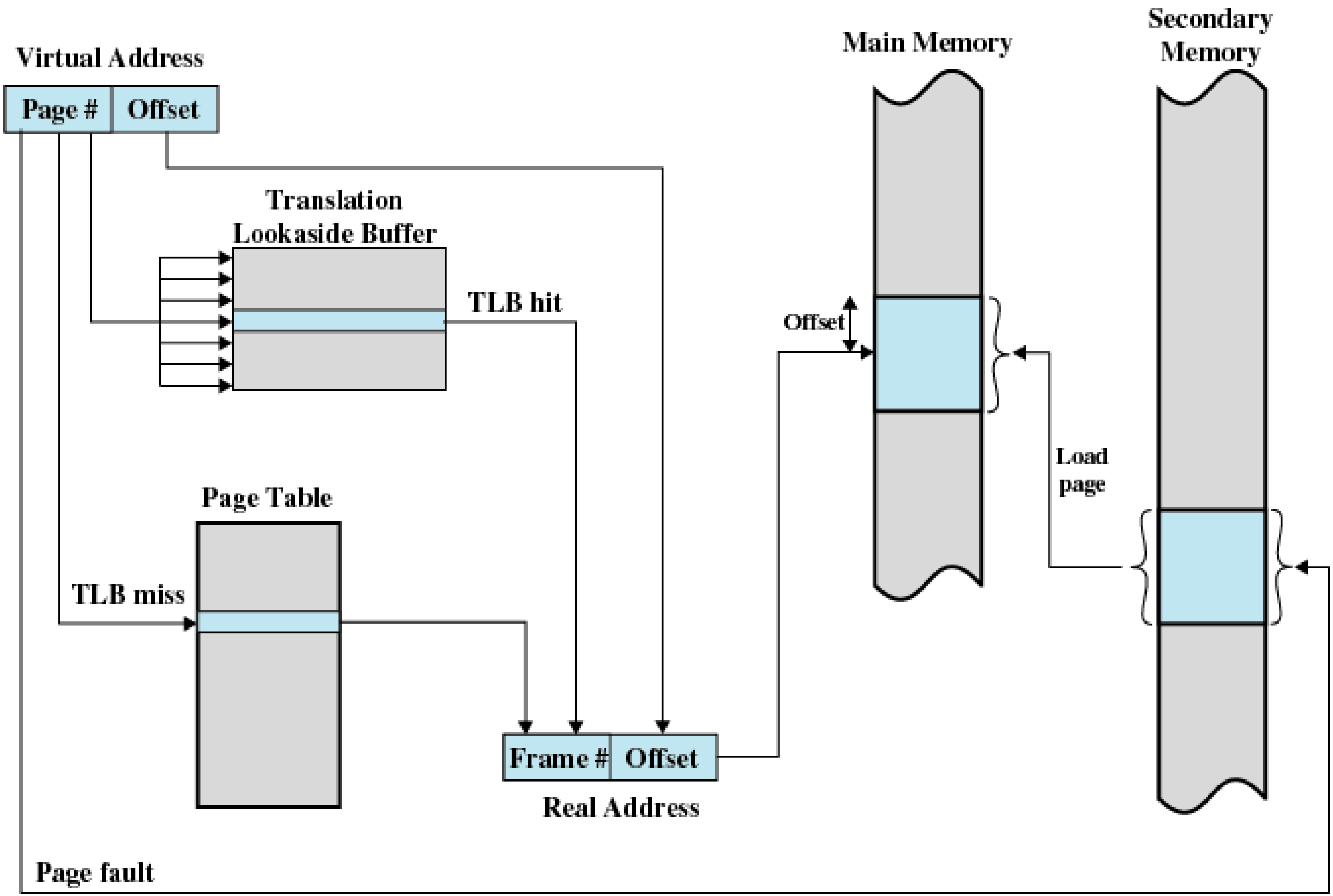
- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table

Translation Lookaside Buffer

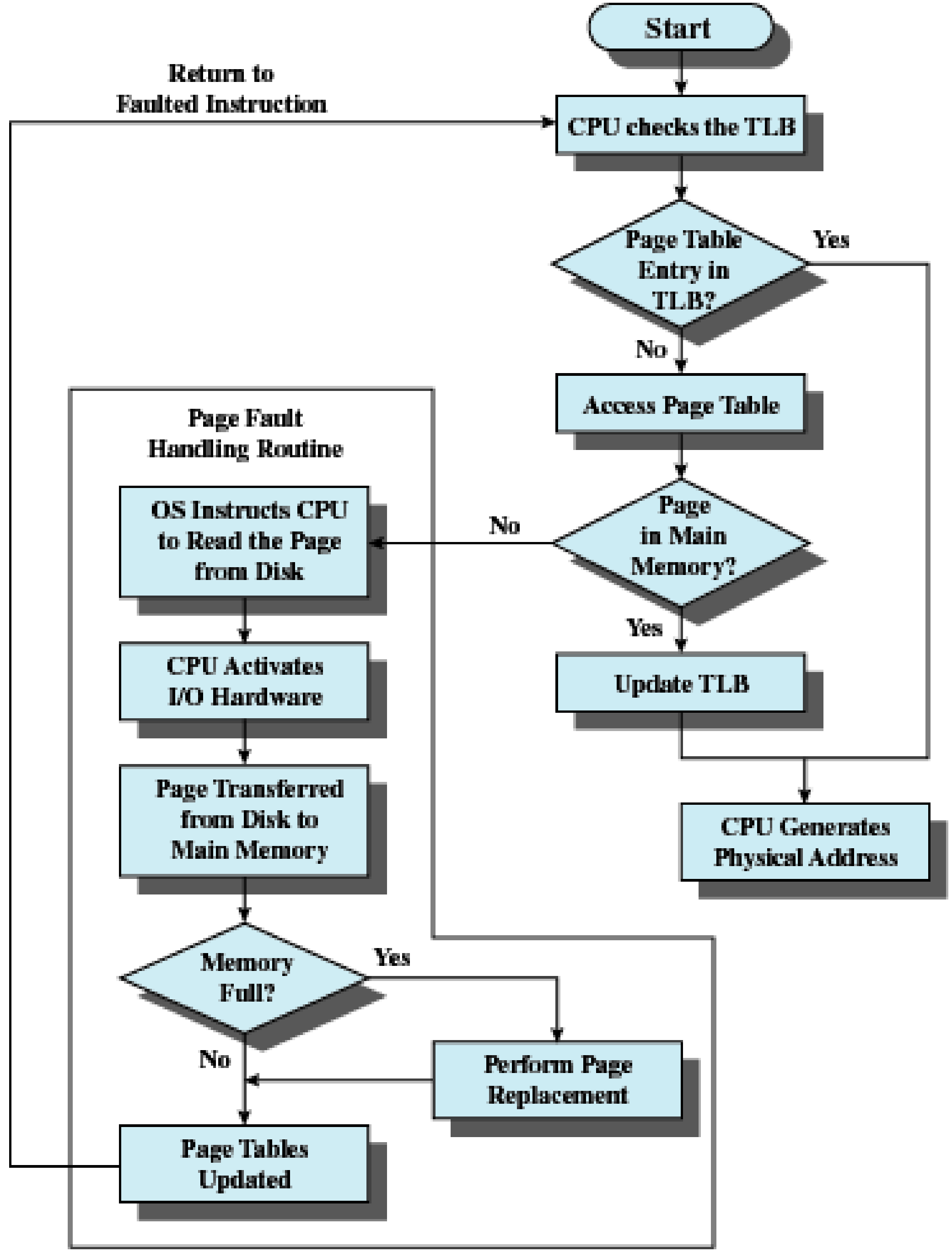
- if page is already in main memory the TLB is updated to include the new page entry
  - If not in main memory a page fault is issued and OS is called

- TLB should be reset on process switch
  - it caches entries of a certain page table.
  - if the page table is changed (process switch)TLB content became useless

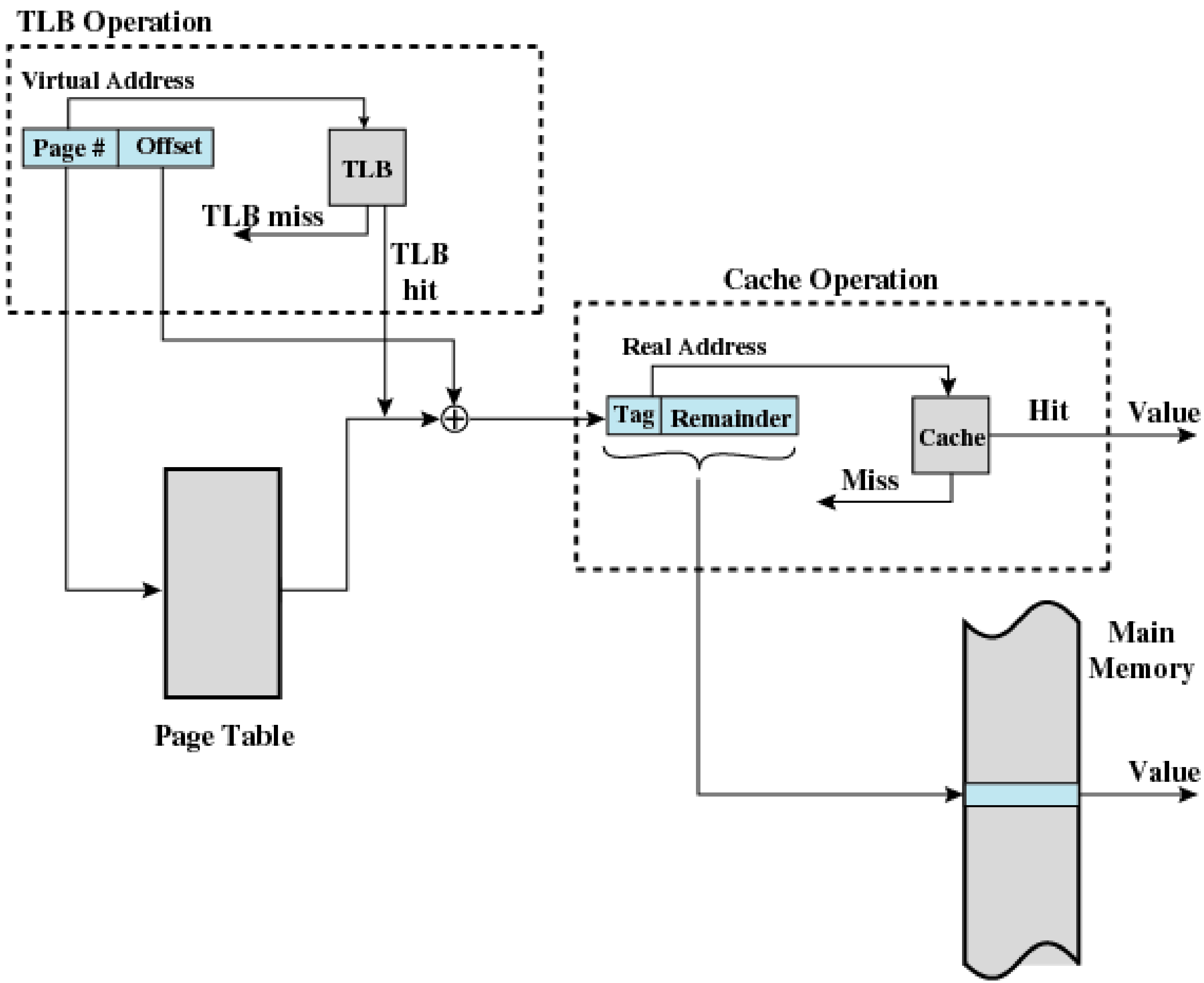
address translation with TLB



lookup algorithm for virtual memory paging with TLB



TLB and memory cache



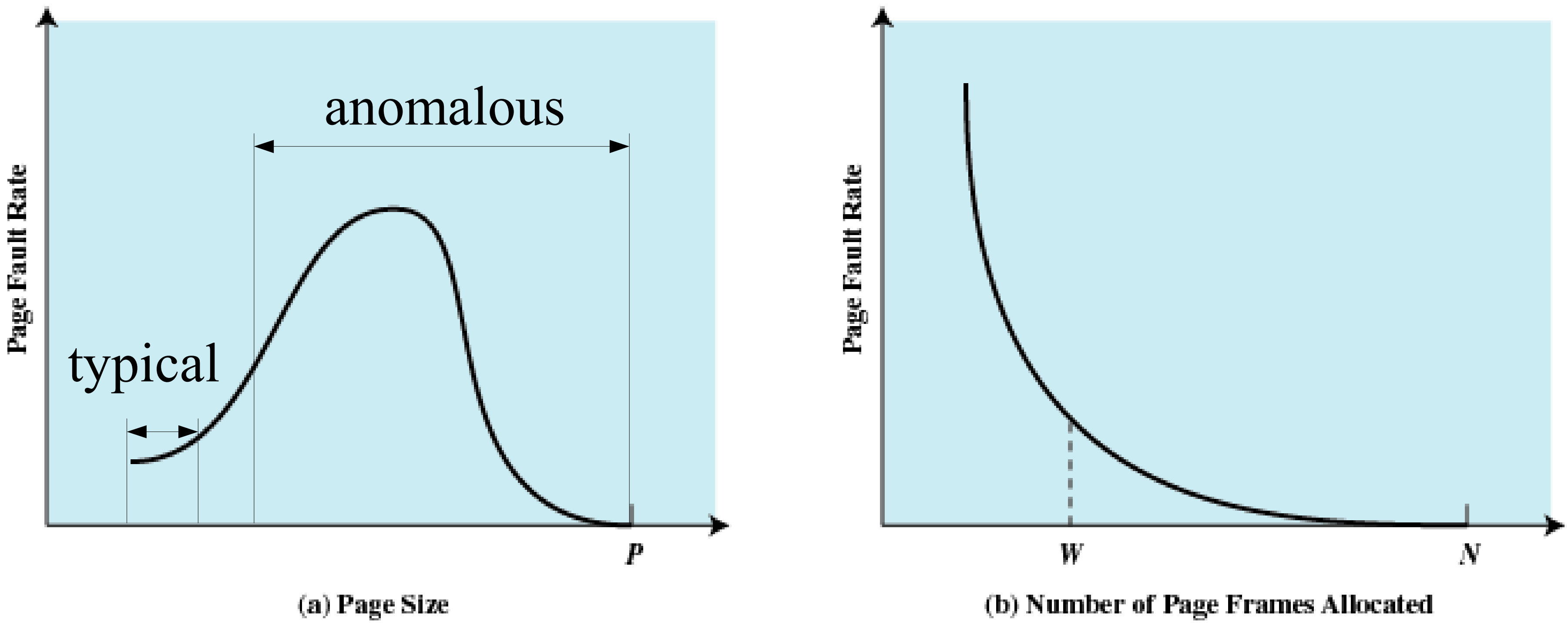
Page Size

- Smaller page size, less amount of internal fragmentation
- Smaller page size, more pages required per process
- More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory
- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better

Page Size

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

typical paging behavior



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

Figure 8.11 Typical Paging Behavior of a Program

Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit word
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 kbyes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
PowerPc	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Segmentation

- Segments may have be unequal size
- segment size may dynamically increase
  - may simplify handling of growing data structures
- Allows modules of programs to be altered and recompiled independently
- makes easy to share data among processes
- implements protection mechanisms

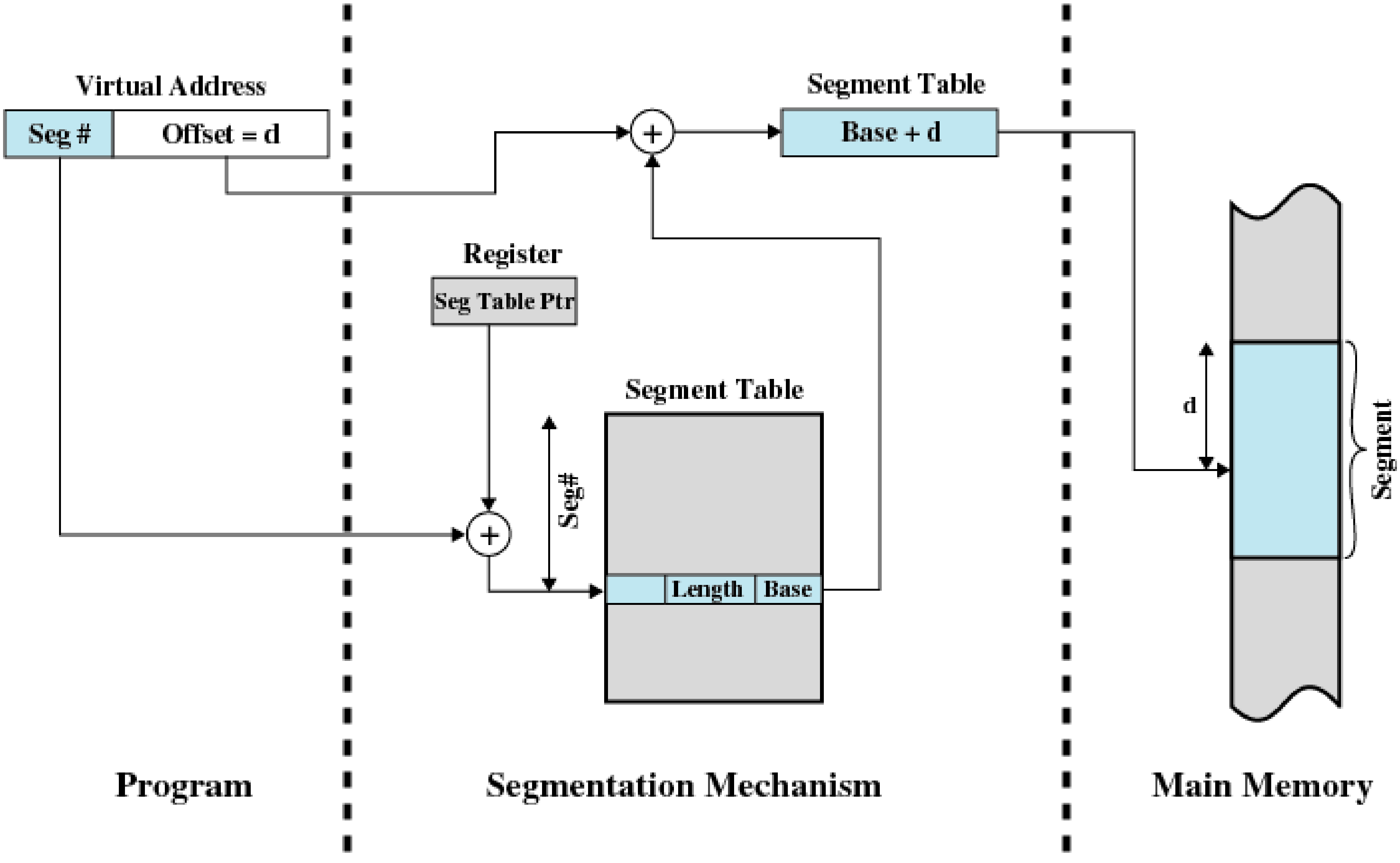
Segment Tables

- one entry for each segment of the process
- each entry contains
  - base address for the segment in main memory
  - the length of the segment
- A bit is needed to determine if segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

Segment Table Entries

(b) Segmentation only

address translation for segmentation



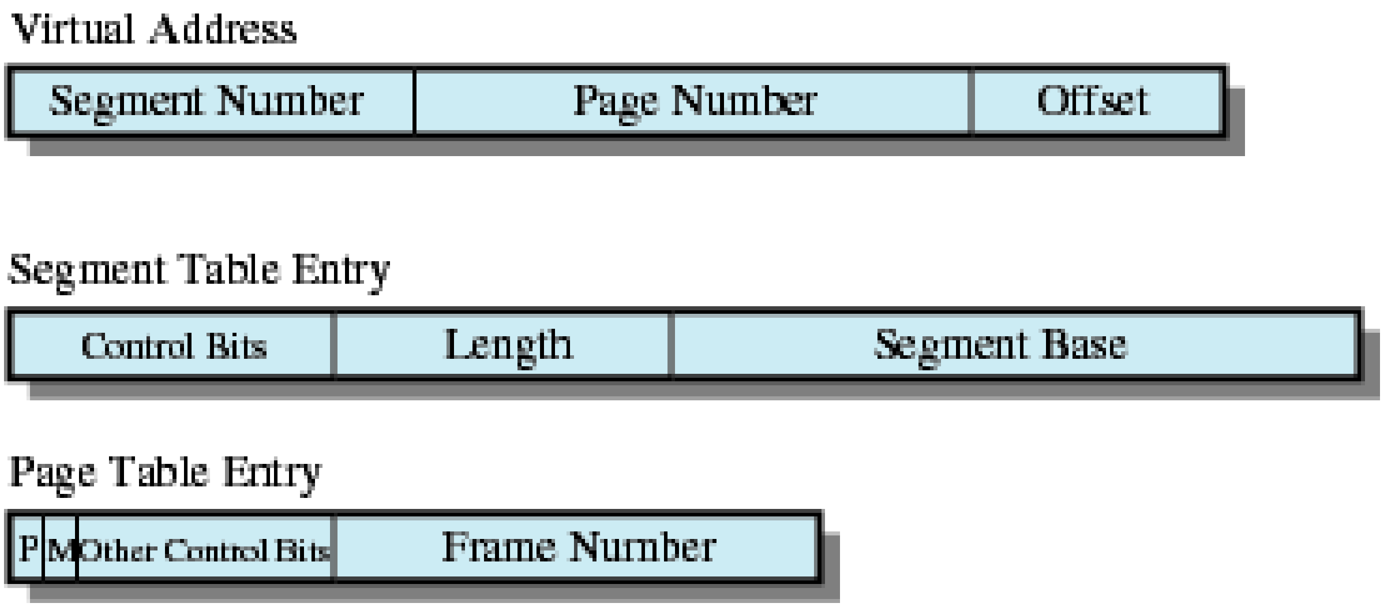
segmentation and virtual memory

- segments are usually very big
- impractical to use with virtual memory
- obsolete
  - segments are usually divided into pages

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

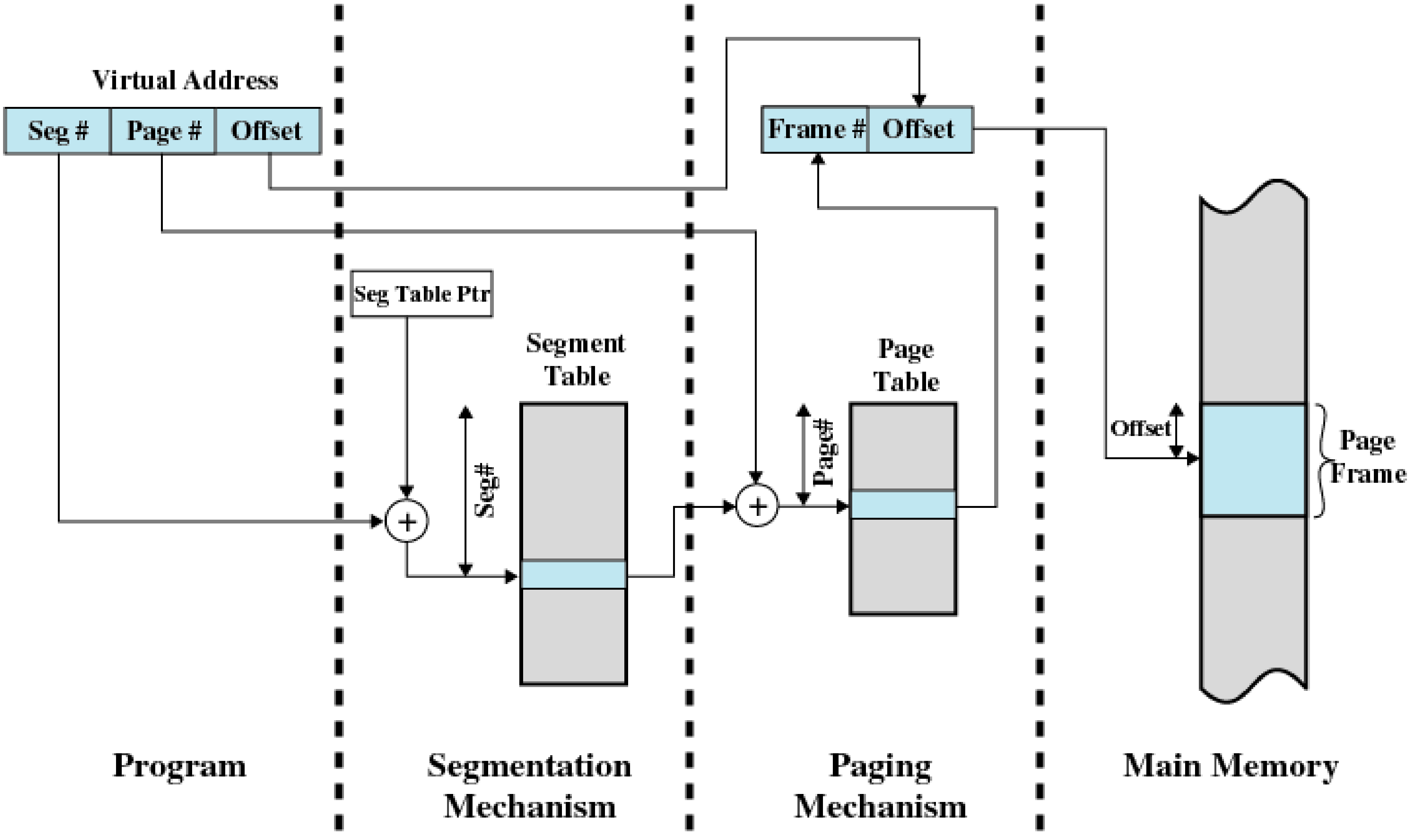
Combined Segmentation and Paging



P= present bit
M = Modified bit

(c) Combined segmentation and paging

address translation for segmentation/paging systems



rss management and eviction policies

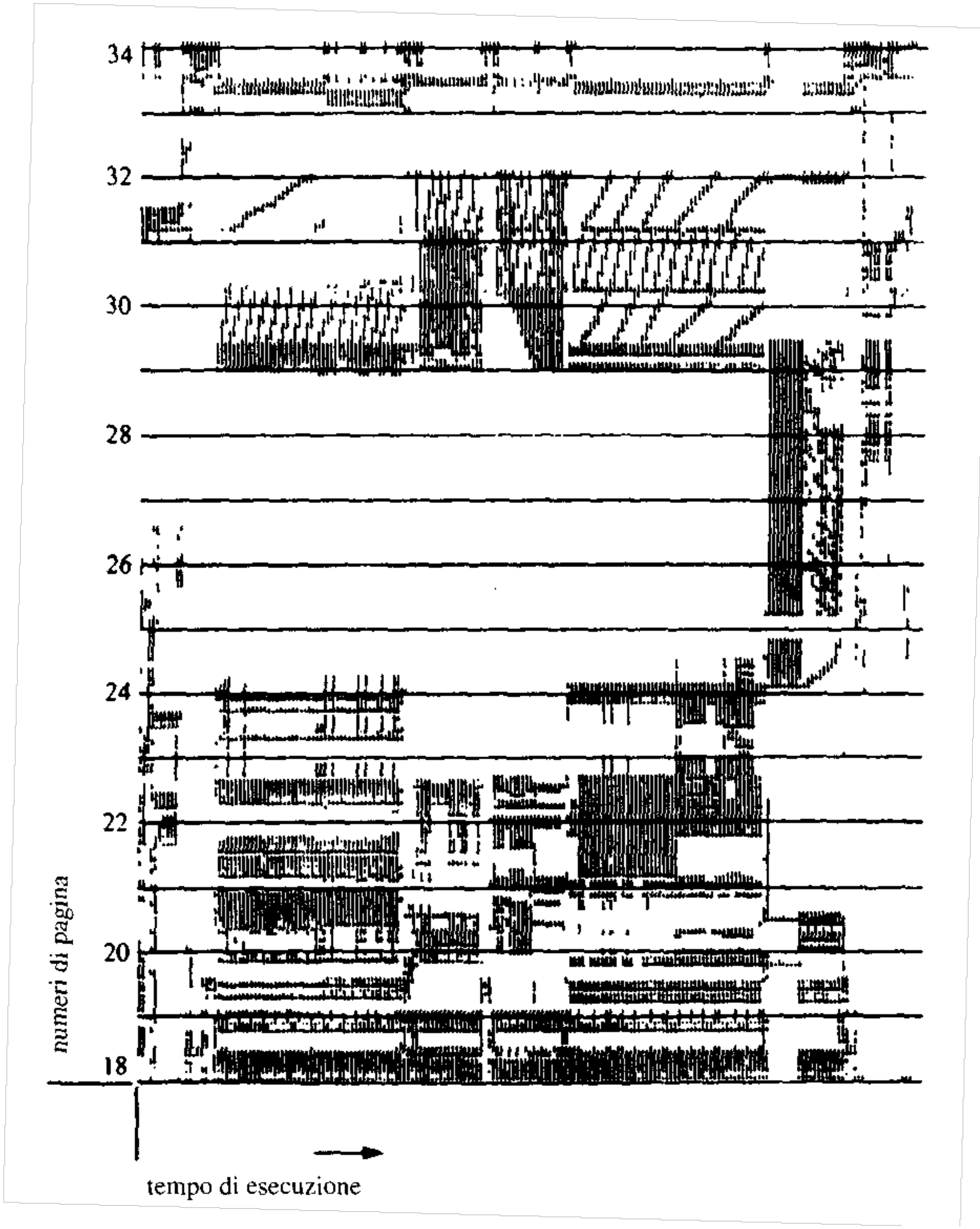
rss management

rss allocation eviction scope	fixed	variable
local	bad usage of main memory	• new process: allocate a number of page frames based on application type, program request, or other criteria • page fault: evict a page in the resident set of the process that caused the fault • Reevaluate allocation from time to time (see **working set**)
global	impossible	• 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, evict one page from any process

locality principle

- program and data references within a process tend to cluster
  - in time and space
- only a few pieces of the process address space are needed over a short period of time
- the behavior of a process in the imminent future is likely to be the same as in the recent past
- this suggests that virtual memory work efficiently in all practical cases

principle of locality



Replacement Policy

- Replacement Policy
  - Which page is evicted?
  - 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

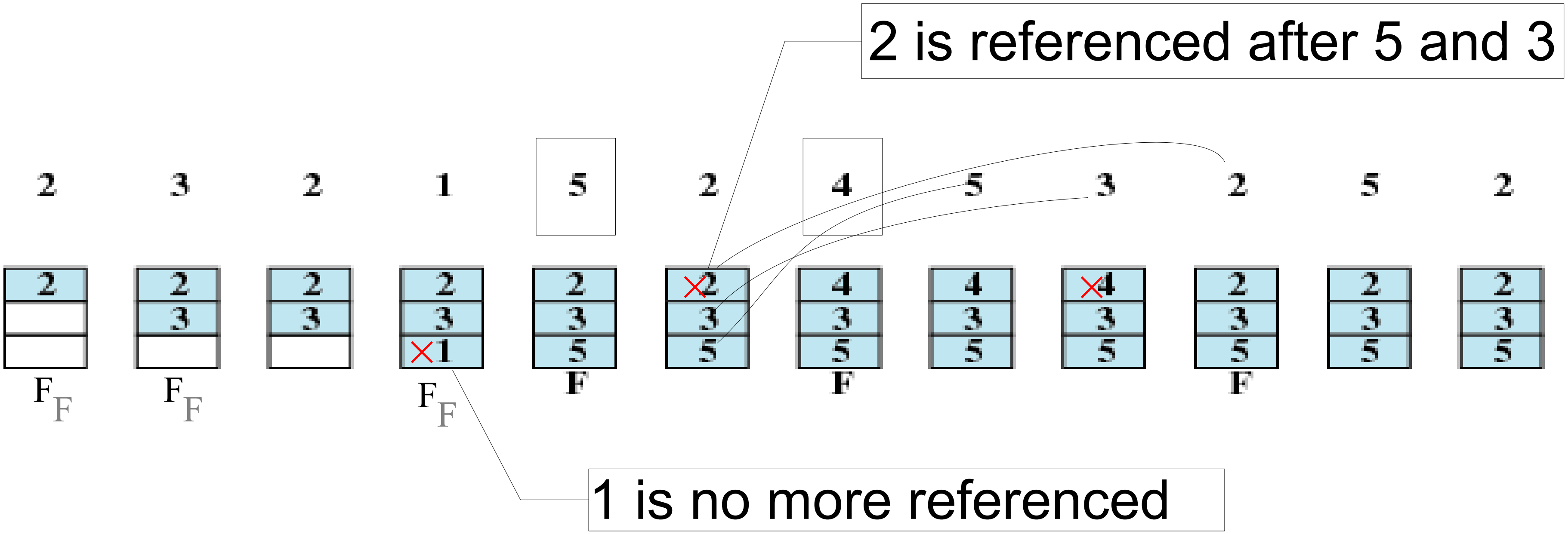
- the part of the kernel that manage the RS of the processes is called *pager* or *swapper*.
- it implements the replacement policy
  – page replacement is the most critical problem to solve for virtual memory efficiency/efficacy

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

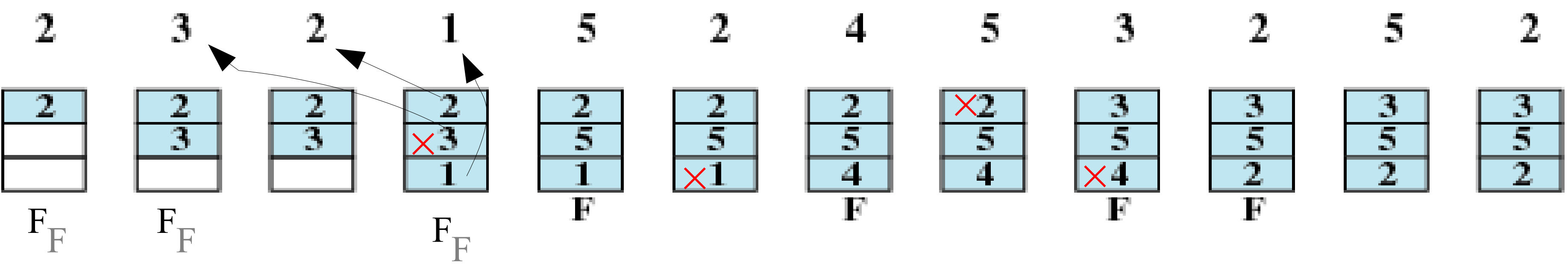
- page references stream:
  2 3 2 1 5 2 4 5 3 2 5 2
- 3 frames are available



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!

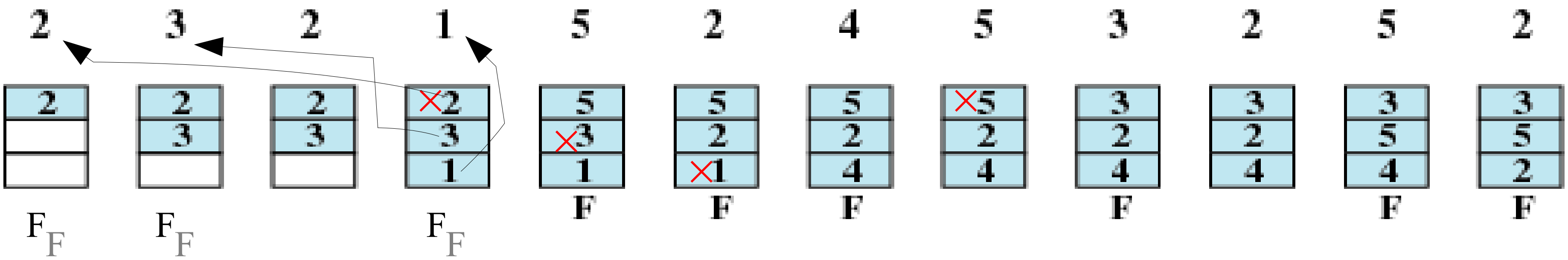
LRU policy example



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

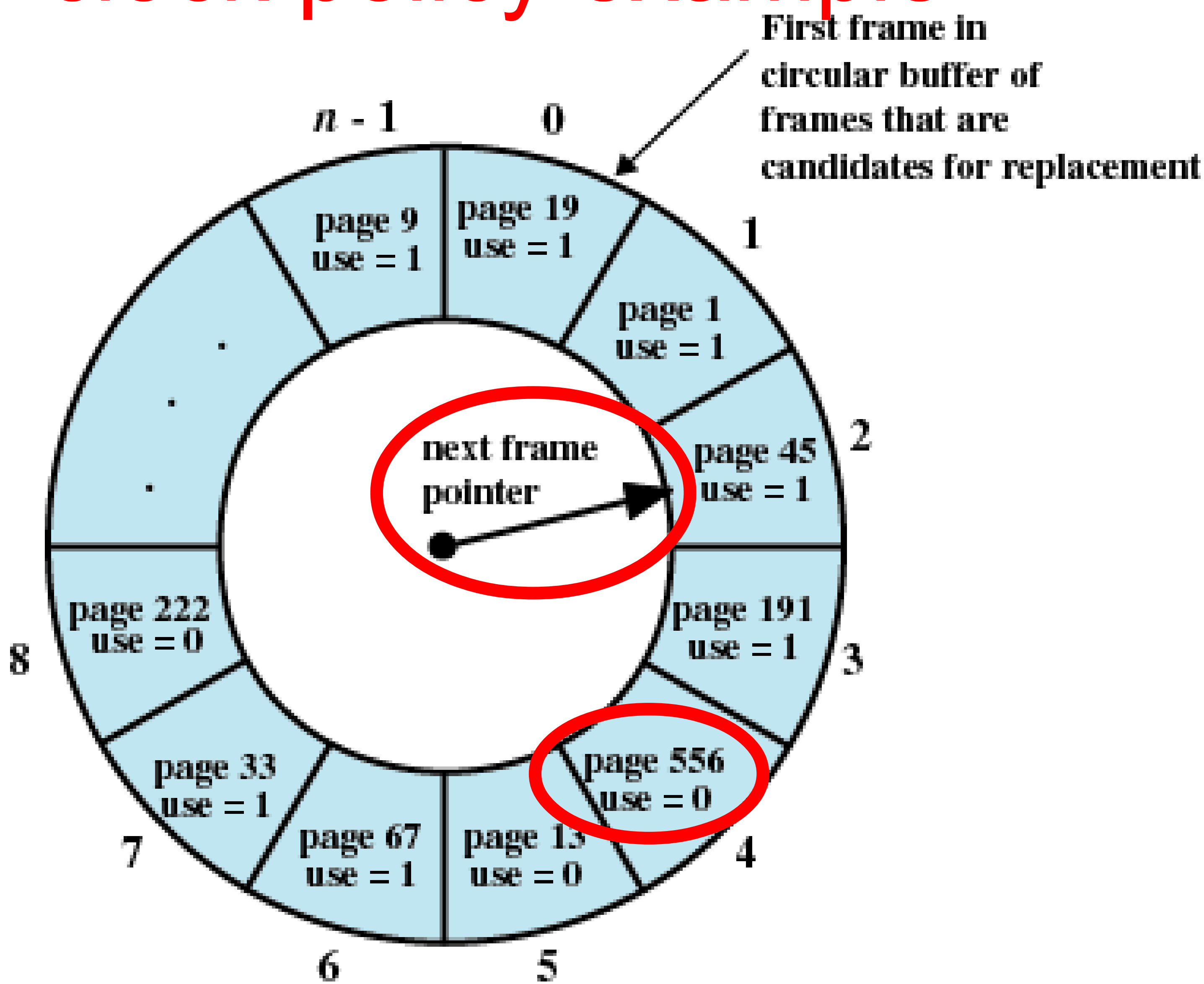
FIFO policy example



Basic Replacement Algorithms/Policies

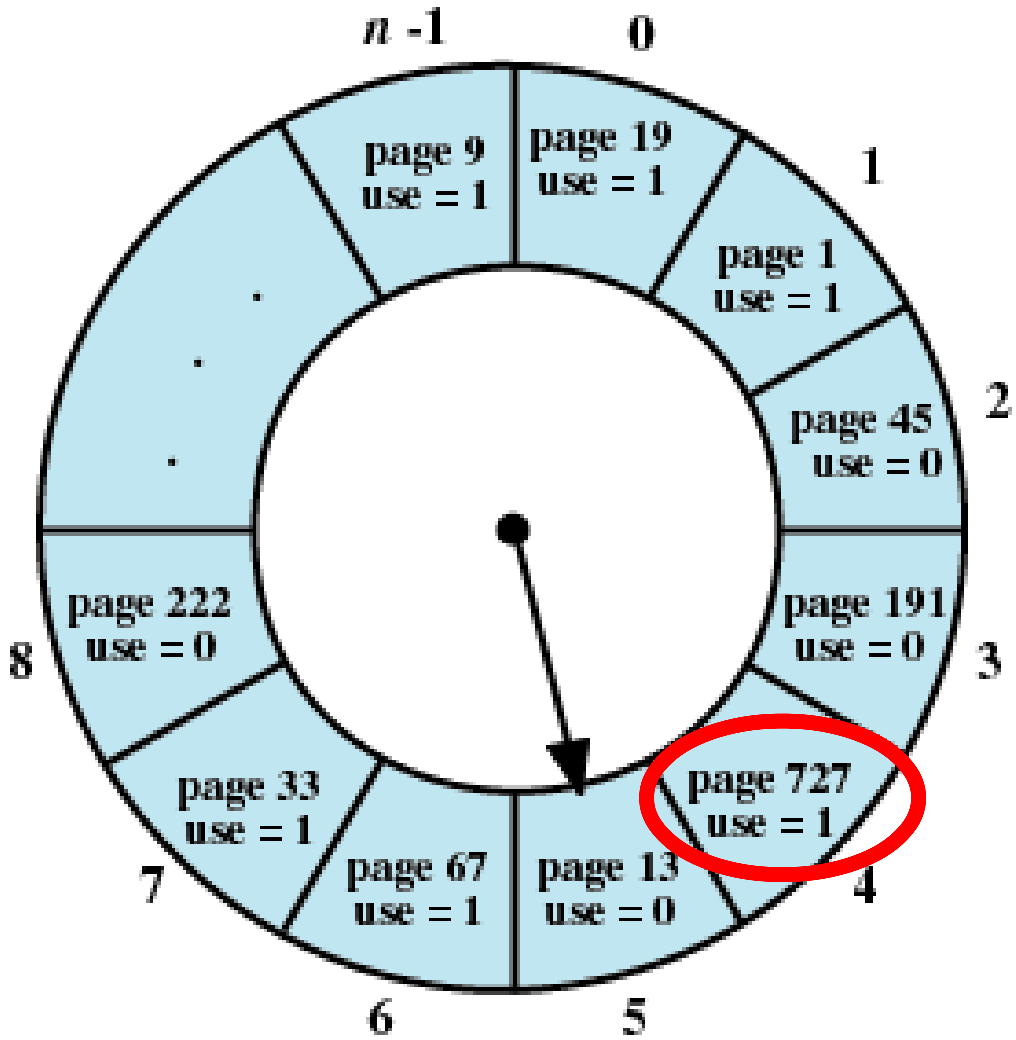
- **Clock Policy (second chance)**
  - one additional for each page bit called a use bit
  - set use=1
    - when a page is first loaded in memory
    - each time a page is referenced
  - when it is time to replace a page scan the frames...
    - the first frame encountered with use=0 is replaced
  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

clock policy example



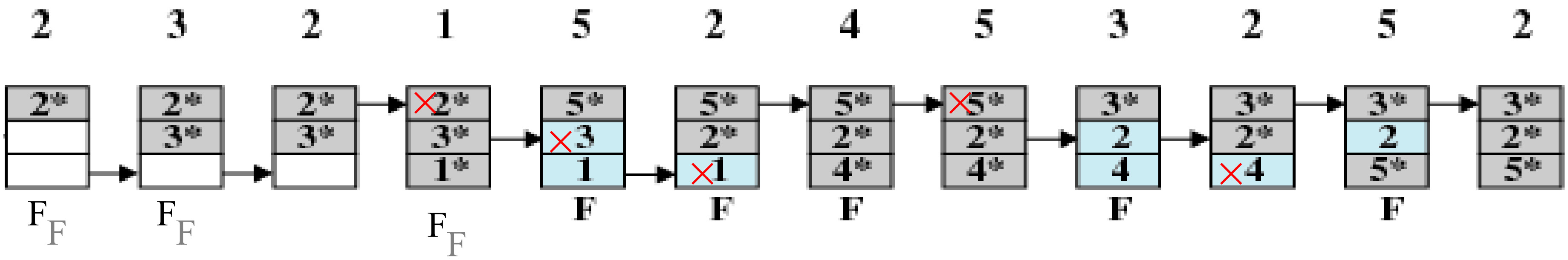
(a) State of buffer just prior to a page replacement

clock policy example

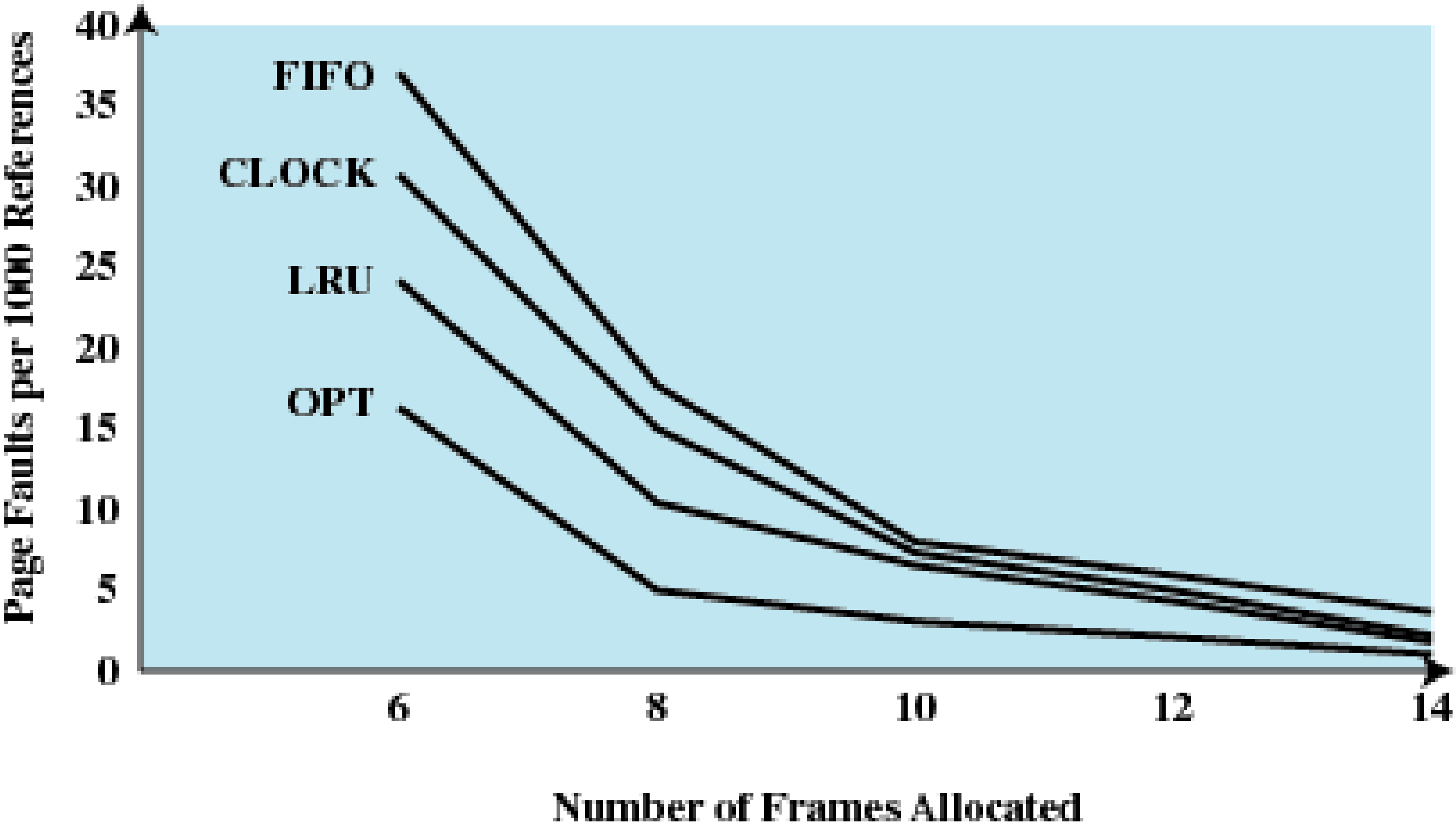


(b) State of buffer just after the next page replacement

clock policy example



comparison of replacement algorithms



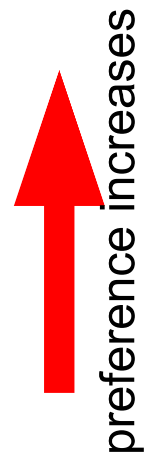
not on
the book

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 on 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

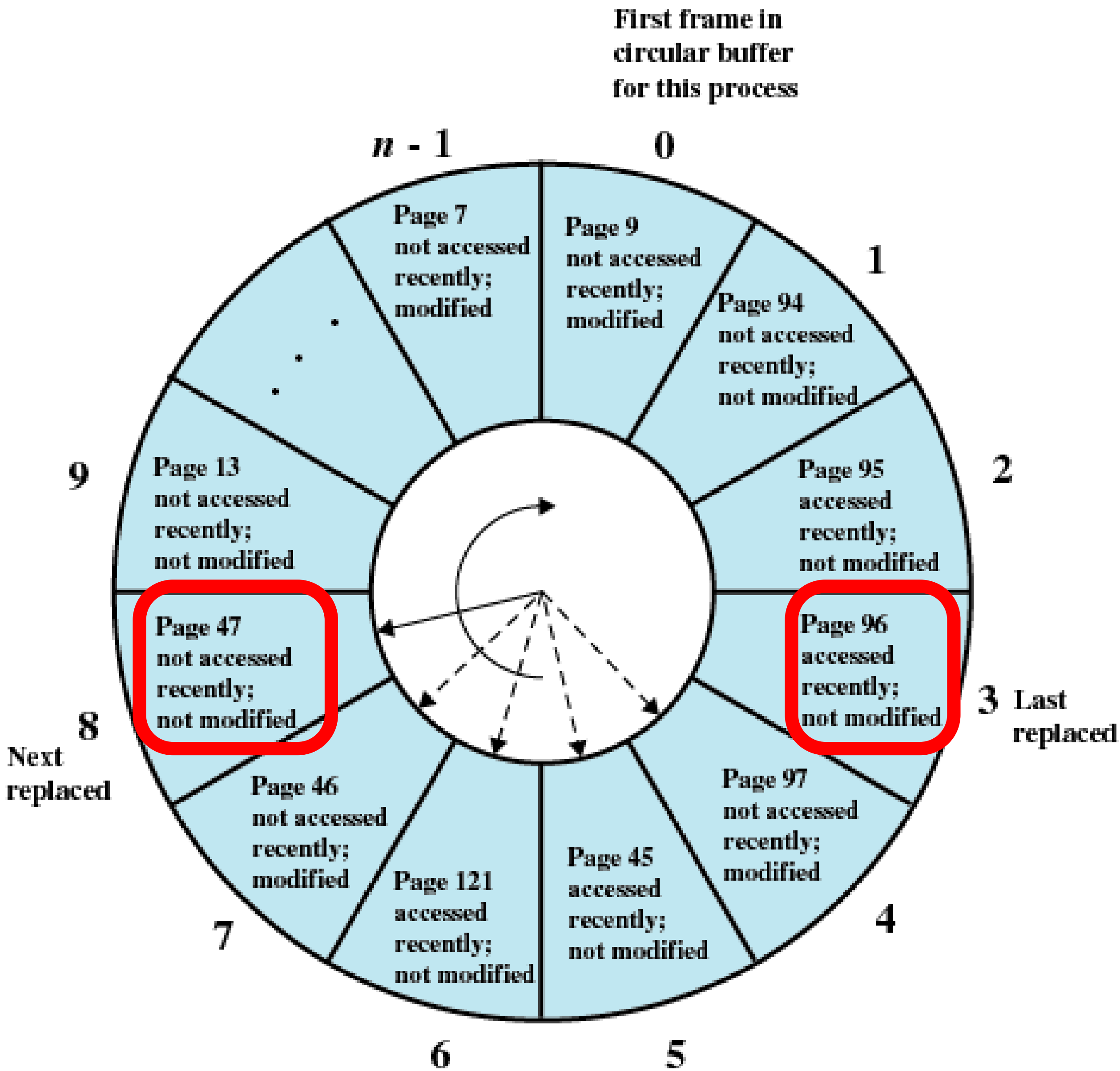
1 look for frames not accessed recently and not modified (use=0, mod=0)

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

(from Tannenbaum)

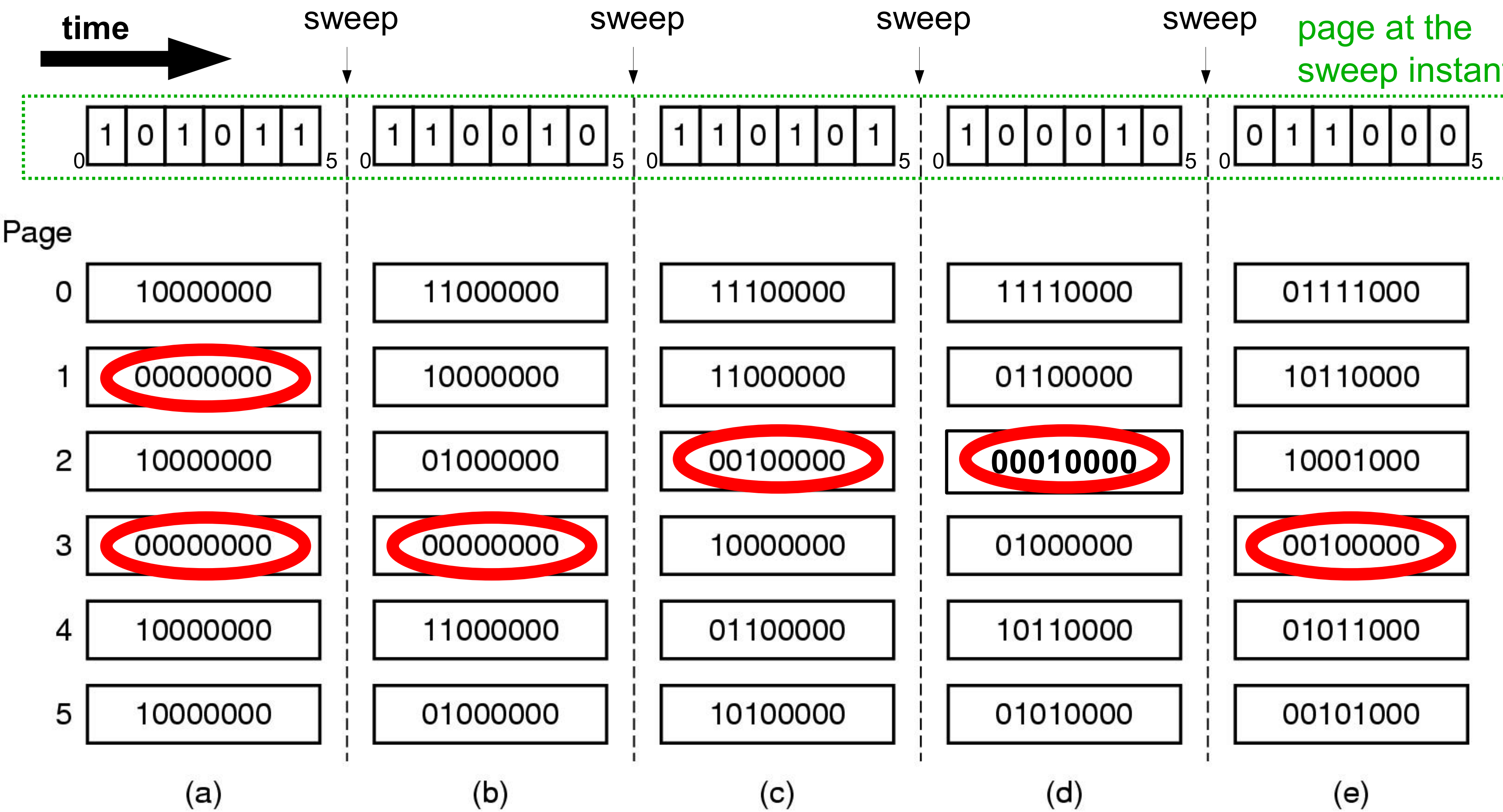
- for each age keeps an age "estimator"
  - the less is the value the older is the page
- it periodically sweeps all pages...
  - scans use bits and modifies estimator for each page
    - example: for page *p* shift right (that is divide by two) and insert the value of use bit for *p* as leftmost bit
      - 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

value of use bits for each page at the sweep instant



“oldest” pages at a certain instant

not on
the book

estimator initialization

- when a page is loaded from the disk what is its estimator?
  - 00000000
  - 00000001
  - 10000000
  - 11111111

not on
the book

estimator initialization

- resonably this page should remain in memory since it has been accessed right now
- estimator should indicate a havily accessed page (e.g. 11111111)

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

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.

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

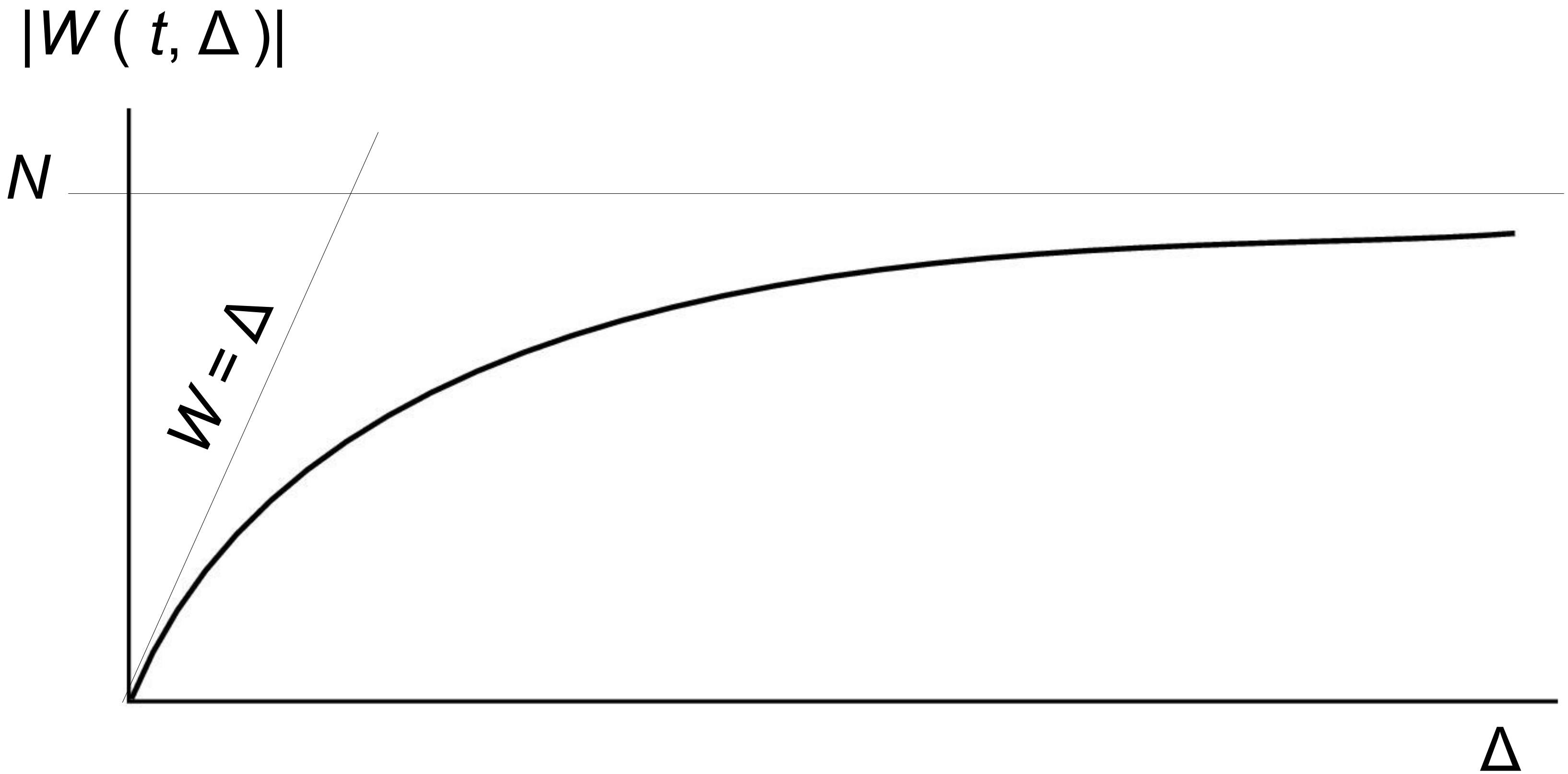
upper bound for the size of *W*

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

N number of pages in the process image

working set

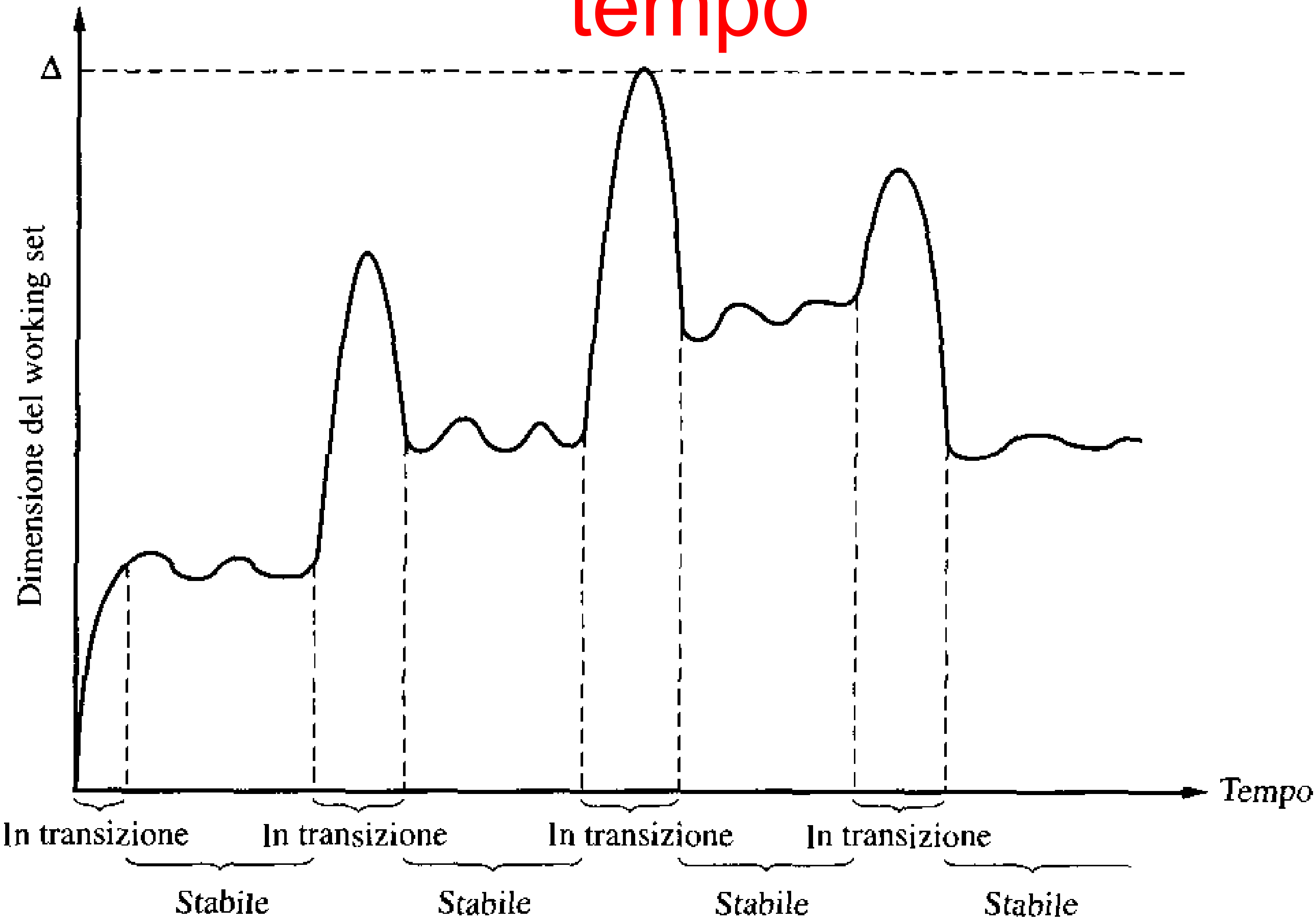
- values of $|W(t, \Delta)|$ varying Δ for t fixed and $t >> N$



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

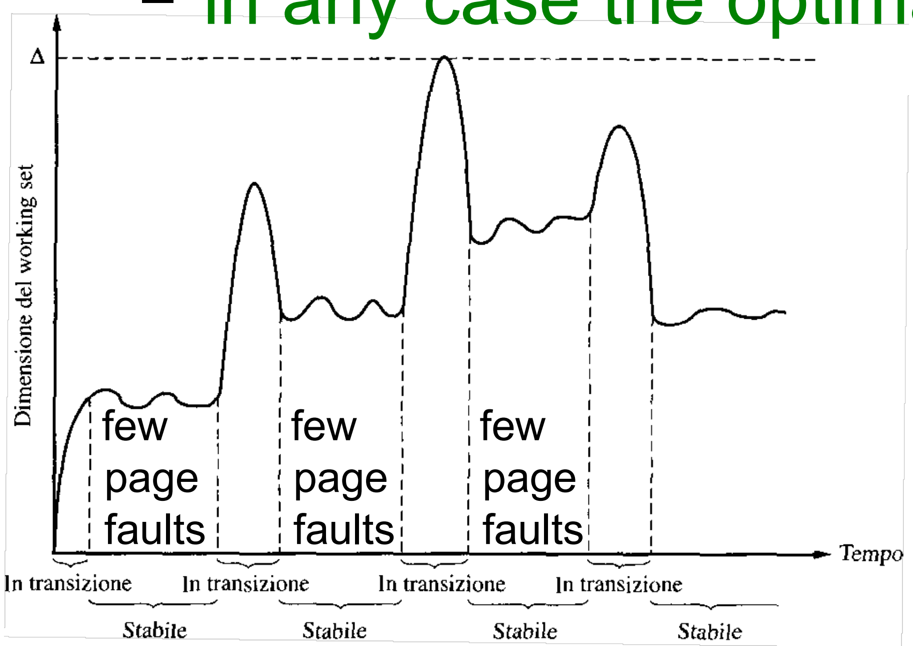


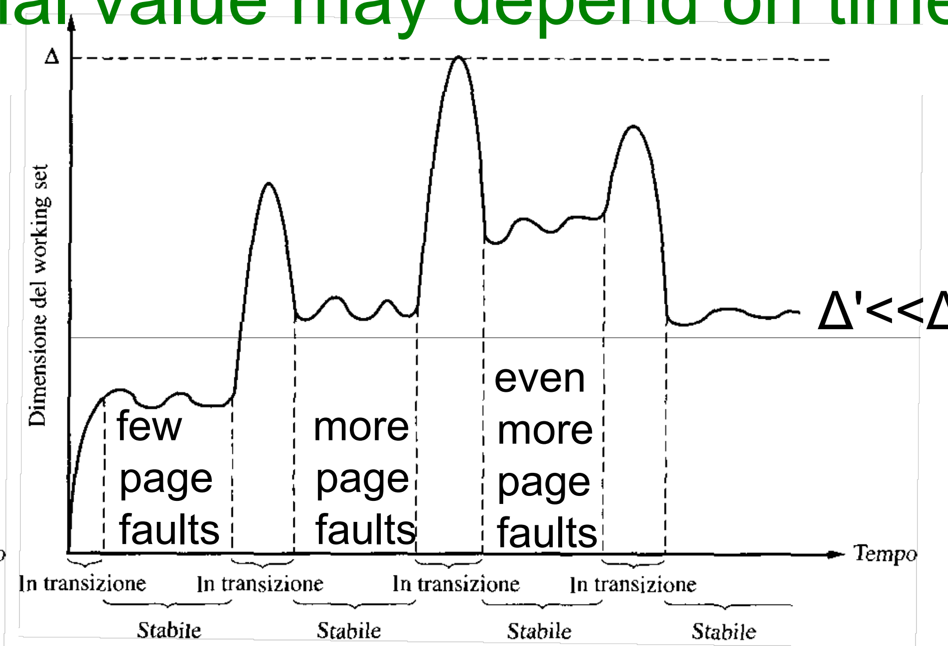
our goal

- 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

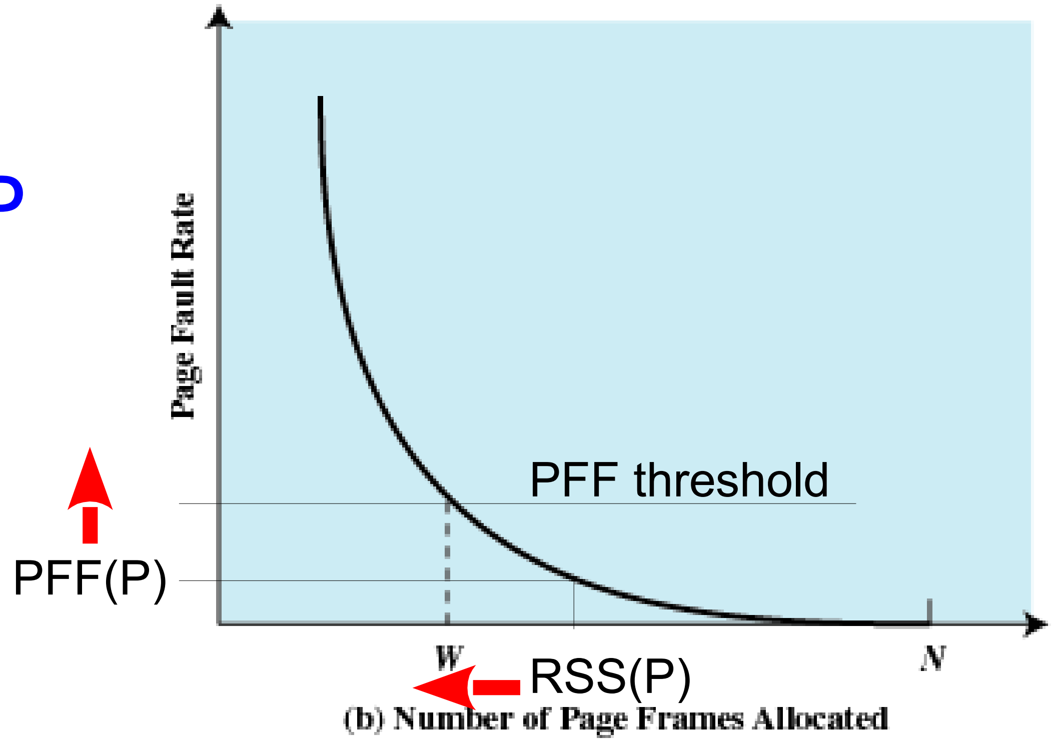
WS strategy approximation

- 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

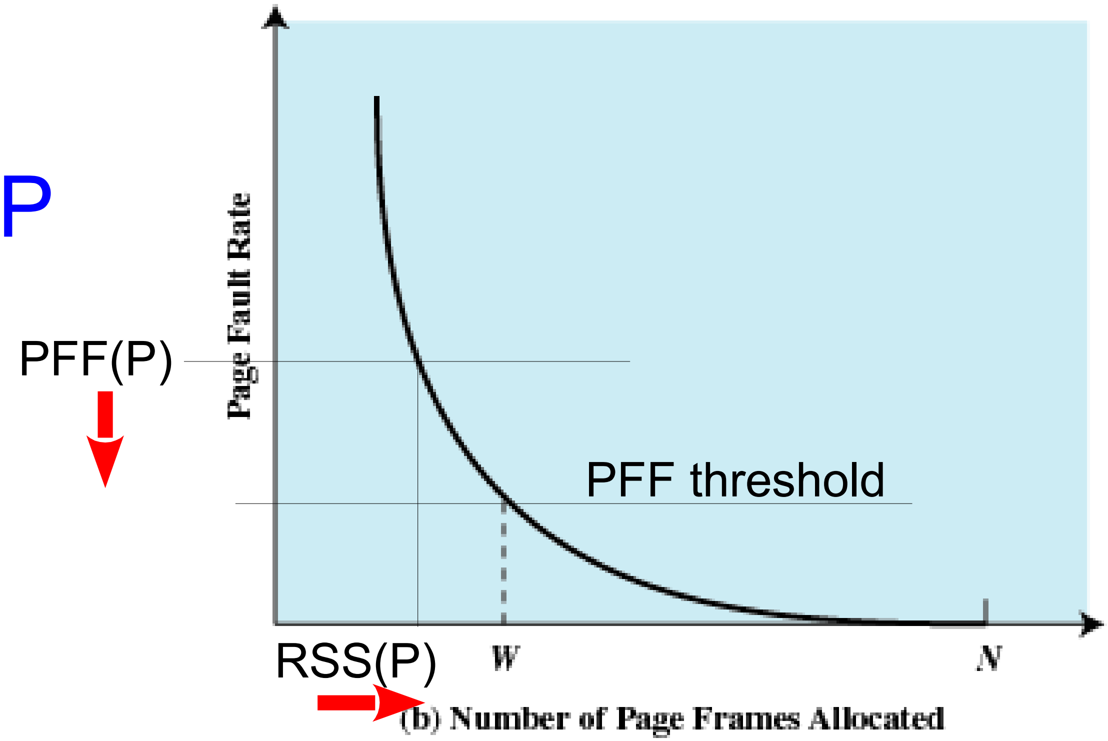
page fault frequency (PFF)

- 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 (memory virtual time can be approximated with real 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