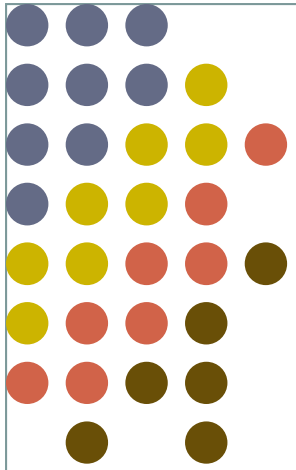


Page replacement algorithms

When a page fault occurs

- OS has to choose a page to evict from memory
- If the page has been modified, the OS has to schedule a disk write of the page
- The page just read overwrites a page in memory (e.g. 4Kbytes)
- Clearly, it's better not to pick a page at random
- Same problem applies to memory caches

Benchmarking

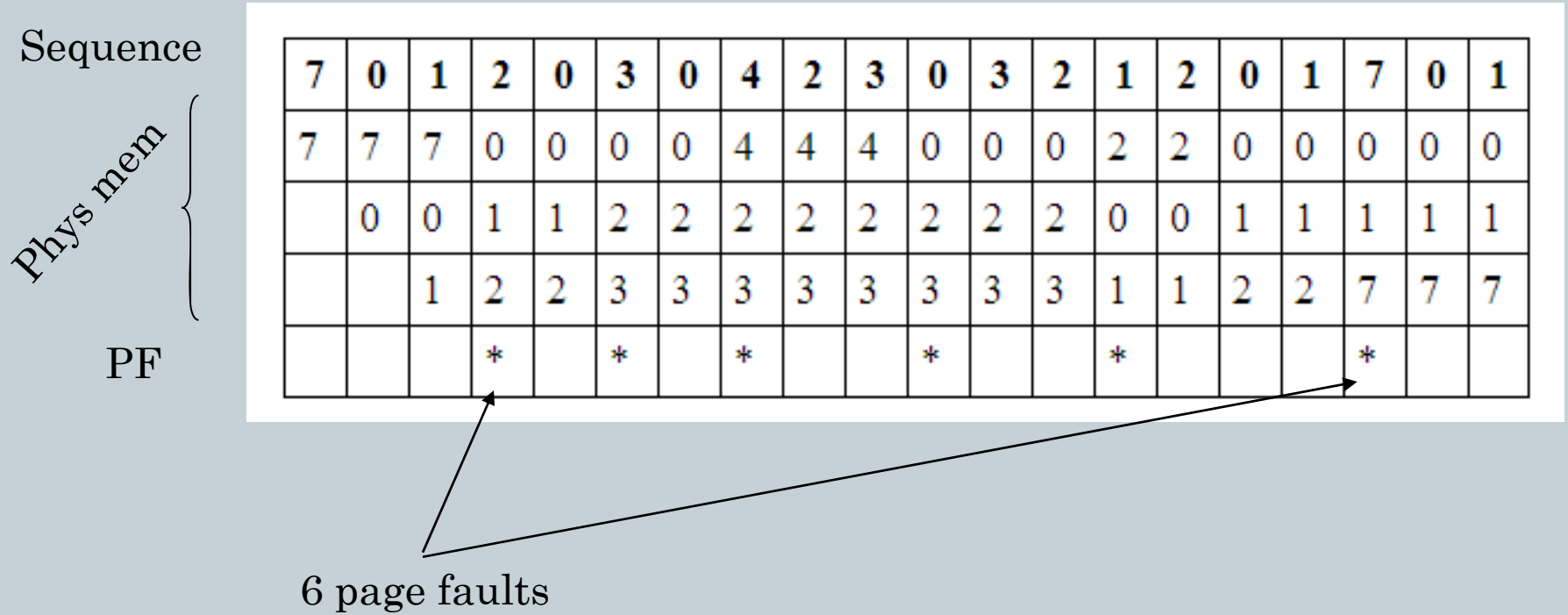
- Tests are done by generating page references (either from real code or random)
- Sequences of page numbers (no real address, no offset)
- Example:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1

Optimal page replacement

- At the moment of page fault:
  - Label each page in memory is labeled with the number of instructions that will be executed before that page is first referenced
  - Replace the page with the highest number: i.e. postpone as much as possible the next page fault
- Nice, optimal, but unrealizable
  - The OS can't look into the future to know how long it'll take to reference every page again

Example: optimal



7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	0	0	0	0	4	4	4	0	0	0	2	2	0	0	0	0	0
	0	0	1	1	2	2	2	2	2	2	2	2	0	0	1	1	1	1	1
		1	2	2	3	3	3	3	3	3	3	3	1	1	2	2	7	7	7
			*		*		*			*			*				*		

Belady's anomaly

Try this sequence

1	2	3	4	1	2	5	1	2	3	4	5

With 3 page frames

With 4 page frames

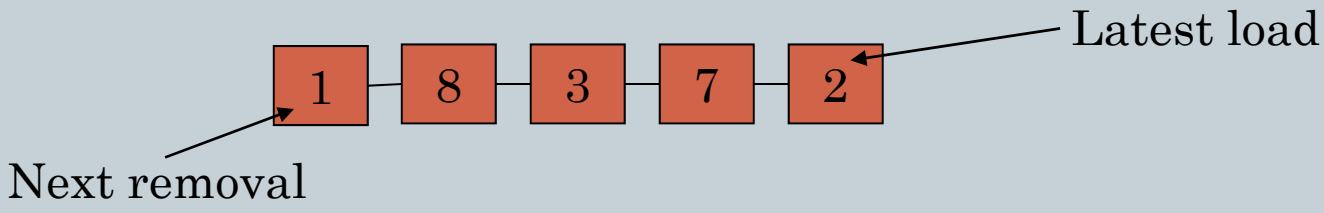
With FIFO, with the optimal algorithm, (later) with the LRU

“Not recently used” algorithm

- Use Referenced and Modified bits
- R&M are in hardware, potentially changed at each reference to memory
  - R&M are zero when process is started
- On clock interrupt the R bit is cleared
- On page fault, to decide which page to evict:
  - Classify:
    - Class 0 – R=0,M=0
    - Class 1 – R=0,M=1
    - Class 2 – R=1,M=0
    - Class 3 – R=1,M=1
  - Replace a page at random from the lowest class

FIFO replacement

- FIFO, first in first out for pages
- Clearly not particularly optimal
- It might end up removing a page that is still referenced since it only looks at the page's age
- Rarely used in pure form…



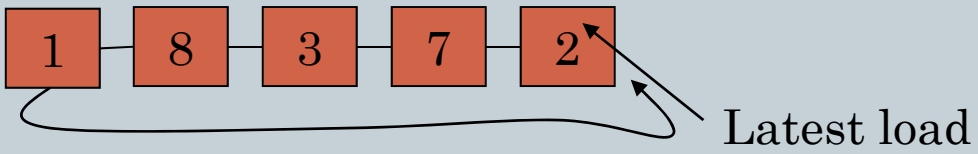
Example (FIFO)

Sequence	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Phys mem	7	7	7	0	0	1	2	3	0	4	2	2	2	3	0	0	0	1	2	7
		0	0	1	1	2	3	0	4	2	3	3	3	0	1	1	1	2	7	0
			1	2	2	3	0	4	2	3	0	0	0	1	2	2	2	7	0	2
PF				*		*	*	*	*	*	*			*	*			*	*	*

12 page faults

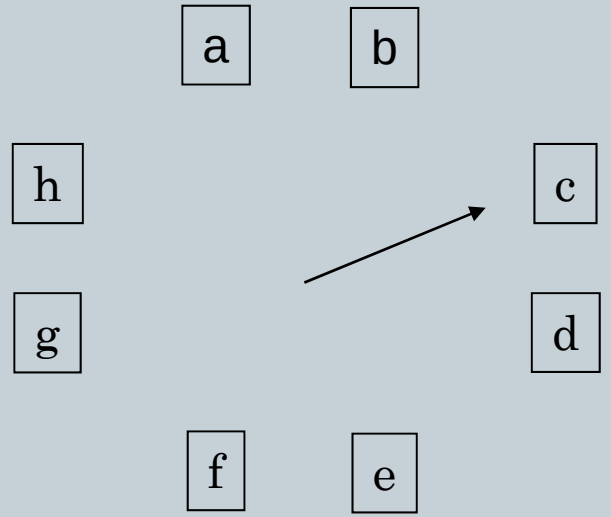
“Second chance” algorithm

- Like FIFO but…
- Before throwing out a page checks the R bit:
  - If 0 remove it
  - If 1 clear it and move the page to the end of the list (as it were just been loaded)
  - If all pages have R=1, eventually the algorithm degenerates to FIFO (why?)



Clock page algorithm

- Like “second chance” but…
- …implemented differently:
  - Check starting from the latest visited page
  - More efficient: doesn’t have to move list’s entries all the time



Least recently used (LRU)

- Why: pages recently used tend to be used again soon (on average)
- List of all pages in memory (most recently in the head, least recently used in the tail)
- List update at each memory reference!
- List operations are expensive (e.g. find a page)
- So, difficult…

Least recently used (LRU)

- Idea! Get a counter, maybe a 64bit counter
- The counter is incremented after each instruction and stored into the page entry at each reference
- Store the value of the counter in each entry of the page table (last access time to the page)
- When is time to remove a page, find the lowest counter value (this is the LRU page)

- Nice & good but expensive: it requires dedicated hardware

Example LRU

Sequence	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Phys mem	7	7	7	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7
		0	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
			1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
PF				*		*		*	*	*	*			*		*		*		

9 page faults

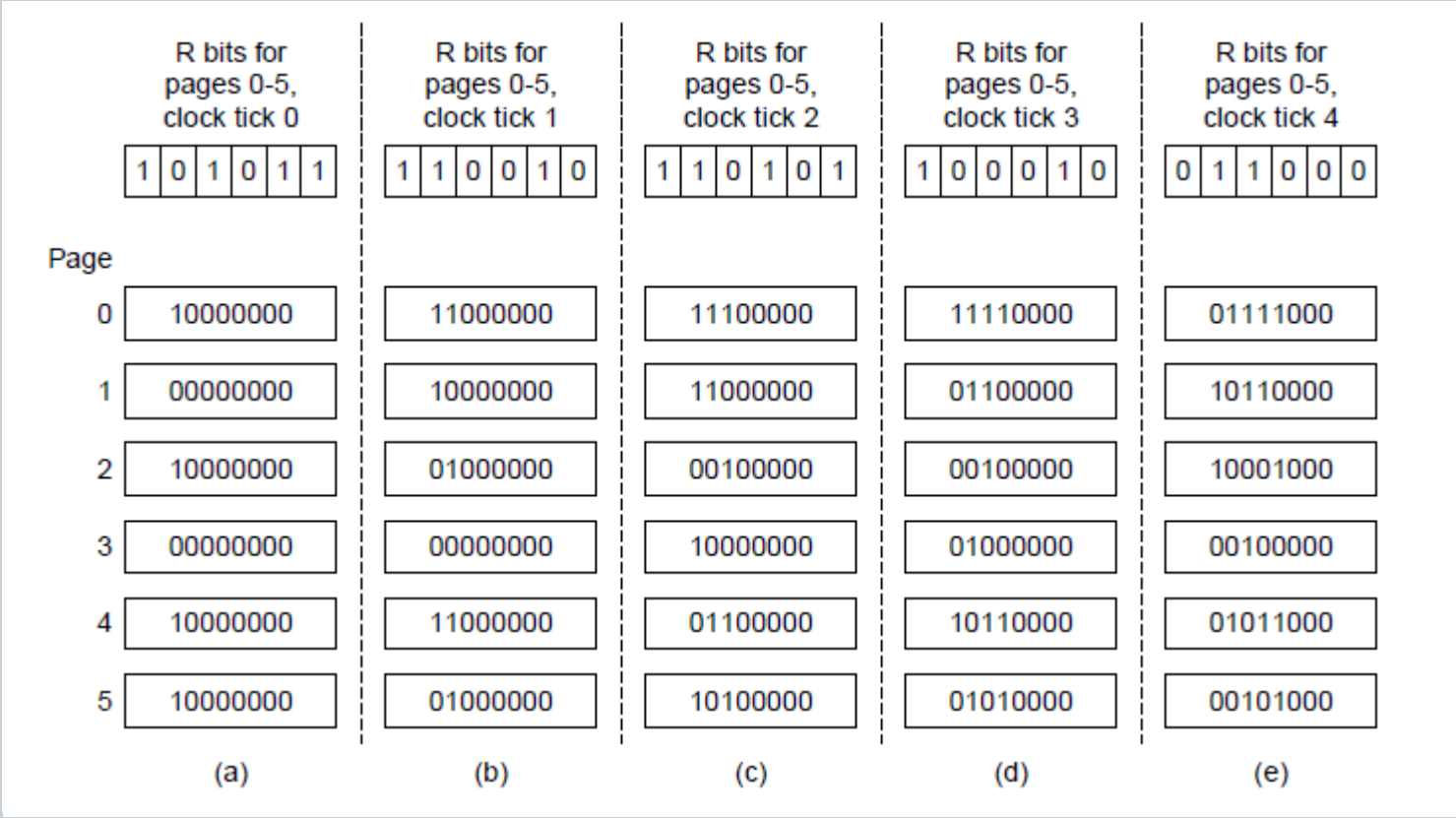
NFU algorithm

- Since LRU is expensive
- NFU: “Not Frequently Used” algorithm
- At each clock interrupt add the R bit to a counter for each page: i.e. count how often a page is referenced
- Remove page with lowest counter value
- Unfortunately, this version tends not to forget anything

Aging (NFU + forgetting)

- Take NFU but…
- At each clock interrupt:
  - Right shift the counters (divide by 2)
  - Add the R bit to the left (MSB)
- As for NFU remove pages with lowest counter
- Note: this is different from LRU since the time granularity is a clock tick and not every memory reference!

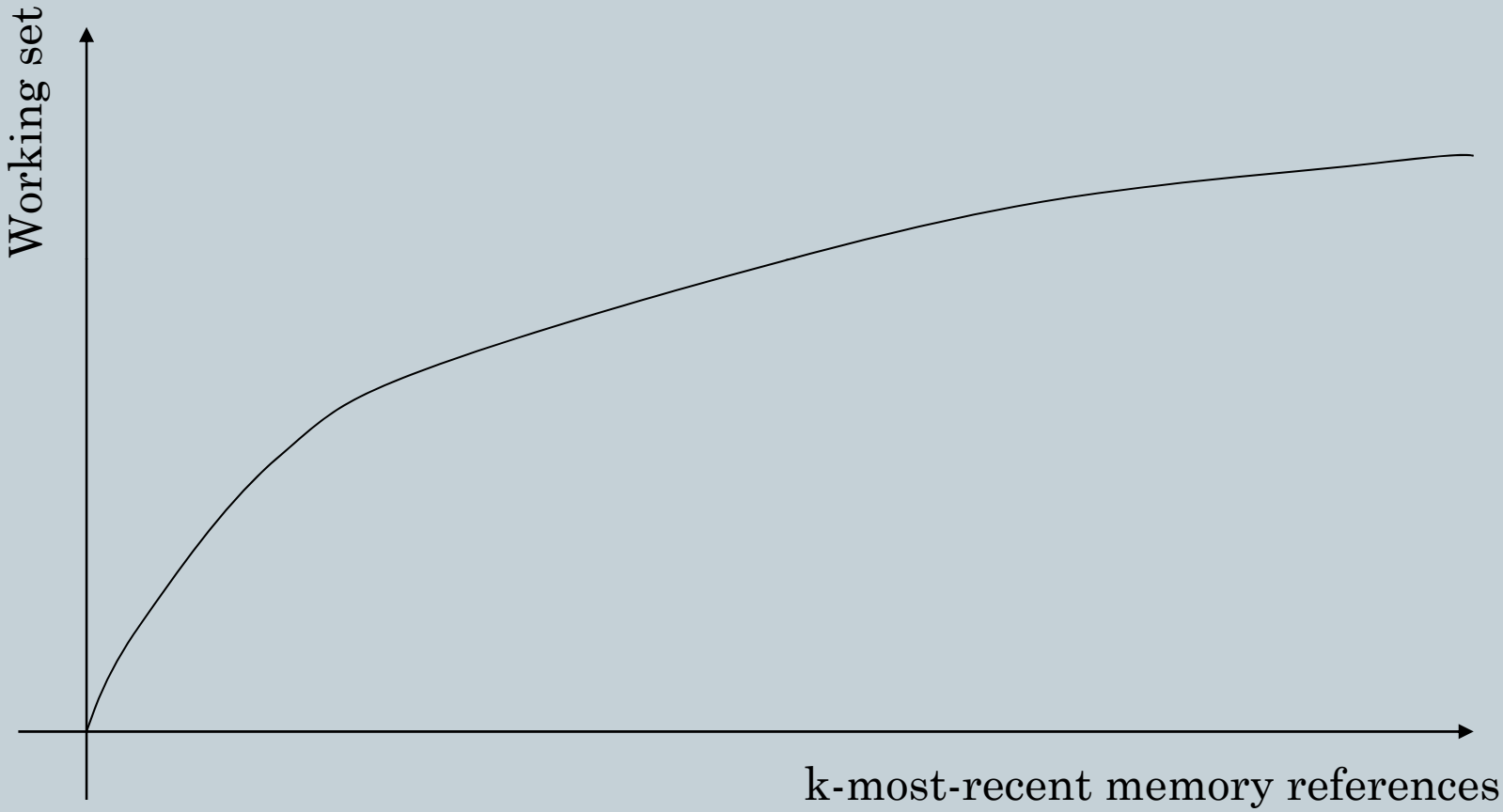
NFU+ageing


R bits for pages 0-5, clock tick 0
1 0 1 0 1 1
R bits for pages 0-5, clock tick 1
1 1 0 0 1 0
R bits for pages 0-5, clock tick 2
1 1 0 1 0 1
R bits for pages 0-5, clock tick 3
1 0 0 0 1 0
R bits for pages 0-5, clock tick 4
0 1 1 0 0 0
Page
0 10000000 11000000 11100000 11110000 01111000
1 00000000 10000000 11000000 01100000 10110000
2 10000000 01000000 00100000 00100000 10001000
3 00000000 00000000 10000000 01000000 00100000
4 10000000 11000000 01100000 10110000 01011000
5 10000000 01000000 10100000 01010000 00101000
(a)
(b)
(c)
(d)
(e)

Process' behavior

- Locality of reference: most of the time the last *k* references are within a finite set of pages < a large address space
- The set of pages a process is currently using is called the *working set* (WS) of the process
- Knowing the working set of processes we can do very sophisticate things (e.g. pre-paging)
- Pre-paging: load pages before letting the process to run so that the page-fault rate is low, also, if I know the WS when I swap the process then I can expect it to be the same in the future time when I reload the process in memory

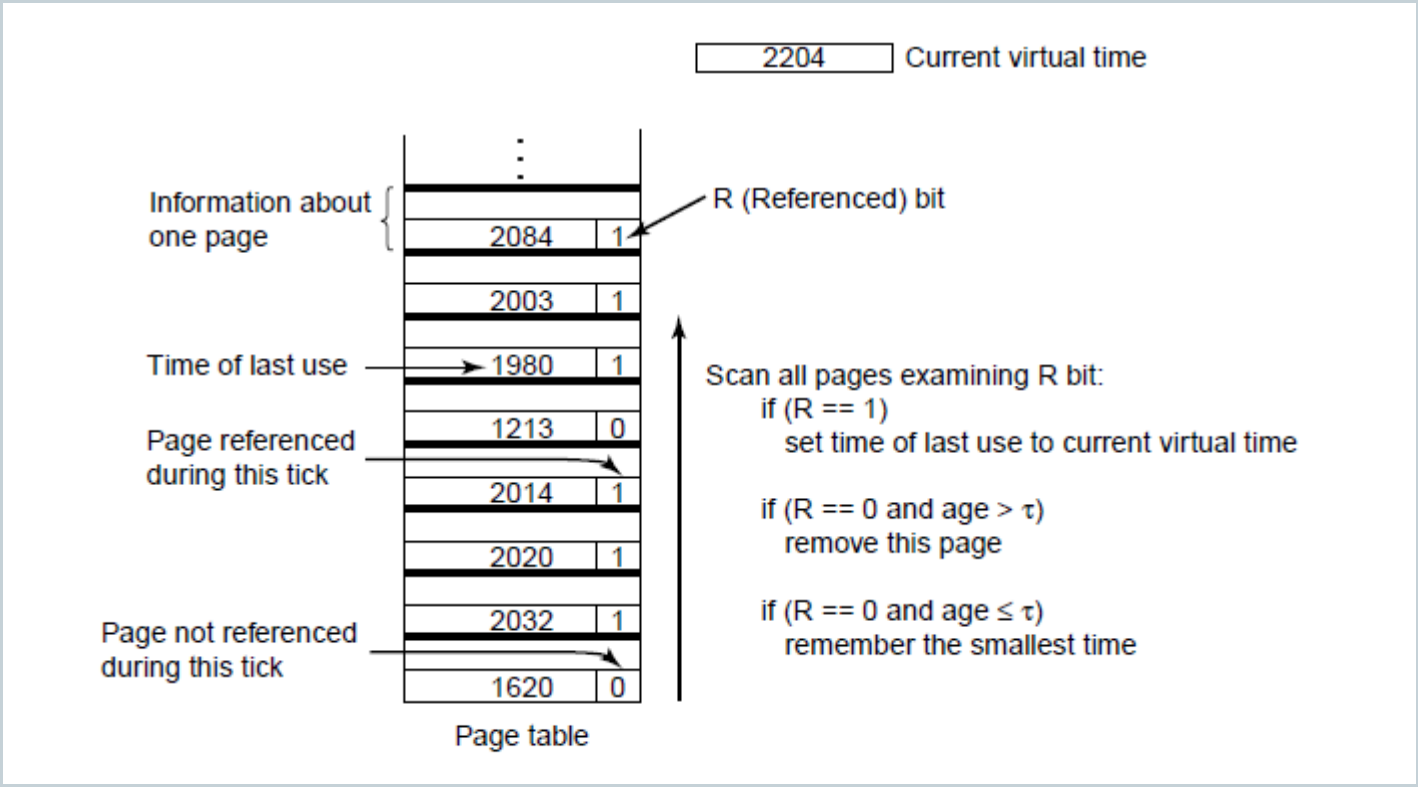
Working set



WS based algorithm

- Store execution time information in the table entries (storing reference is too expensive!)
- At clock interrupt handle R bits as usual (clear them)
- At page fault, scan entries:
  - If R=1 just store current time in the entry
  - If R=0 compute "current-last time page was referenced" and if > *threshold* the page can be removed since it's no longer in the working set (not used for *threshold* time)
- **Note**: we're using time rather than actual memory references

e.g.: WS based algorithm



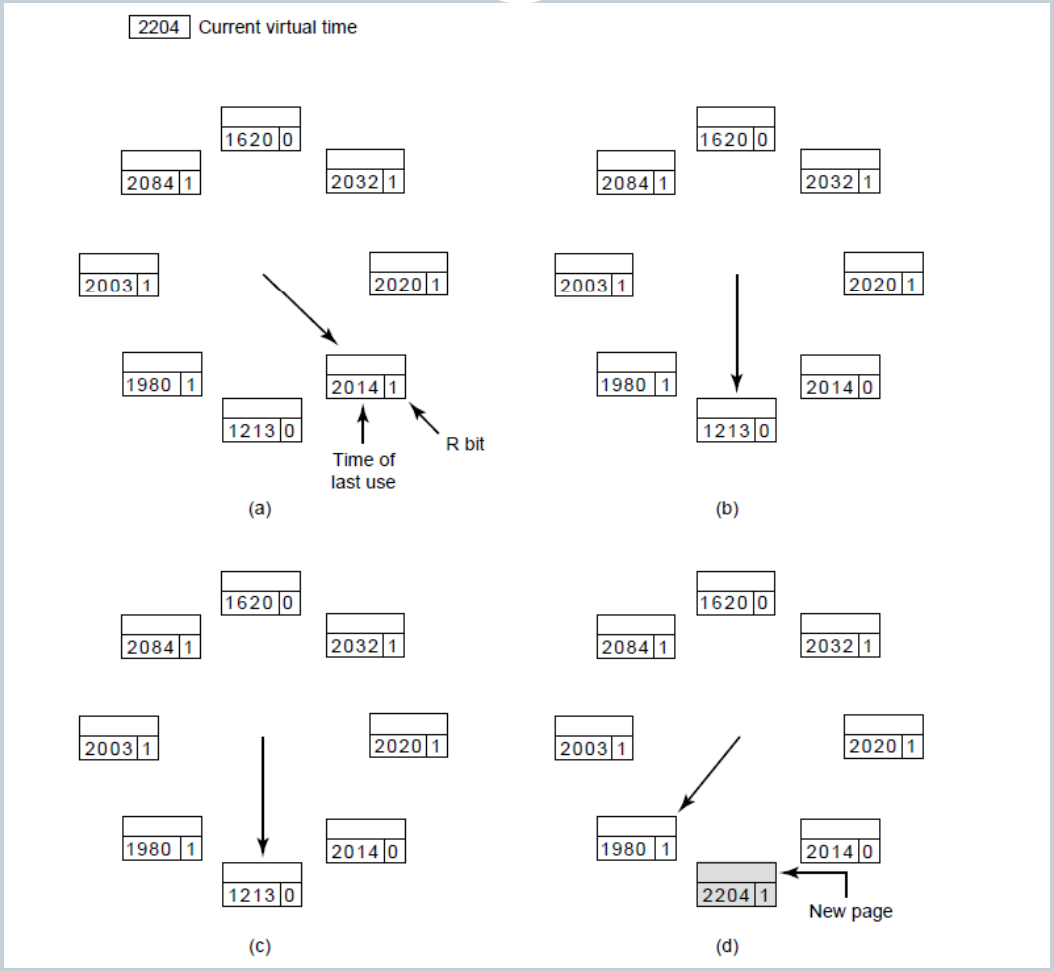
WSClock algorithm

- Use the circular structure (as seen earlier)
- At page fault examine the page pointed by the handle
- R=1, page in the WS – don't remove it (set R to zero)
- R=0, M=0 no problem (as before, check age, page clean and decide depending on age)
- If M=1, schedule disk write appropriately to procrastinate as long as possible a process switch
  - If return to starting point, then one page will eventually be clean (maybe after a context switch)
  - Scheduling multiple disk write can be efficient in efficient systems (with disk scheduling and multiple disks)
  - No write is schedulable (R=1 always), just choose a clean page
  - No clean page, use the current page under the handle

WSClock

R=1

R=0



Summary

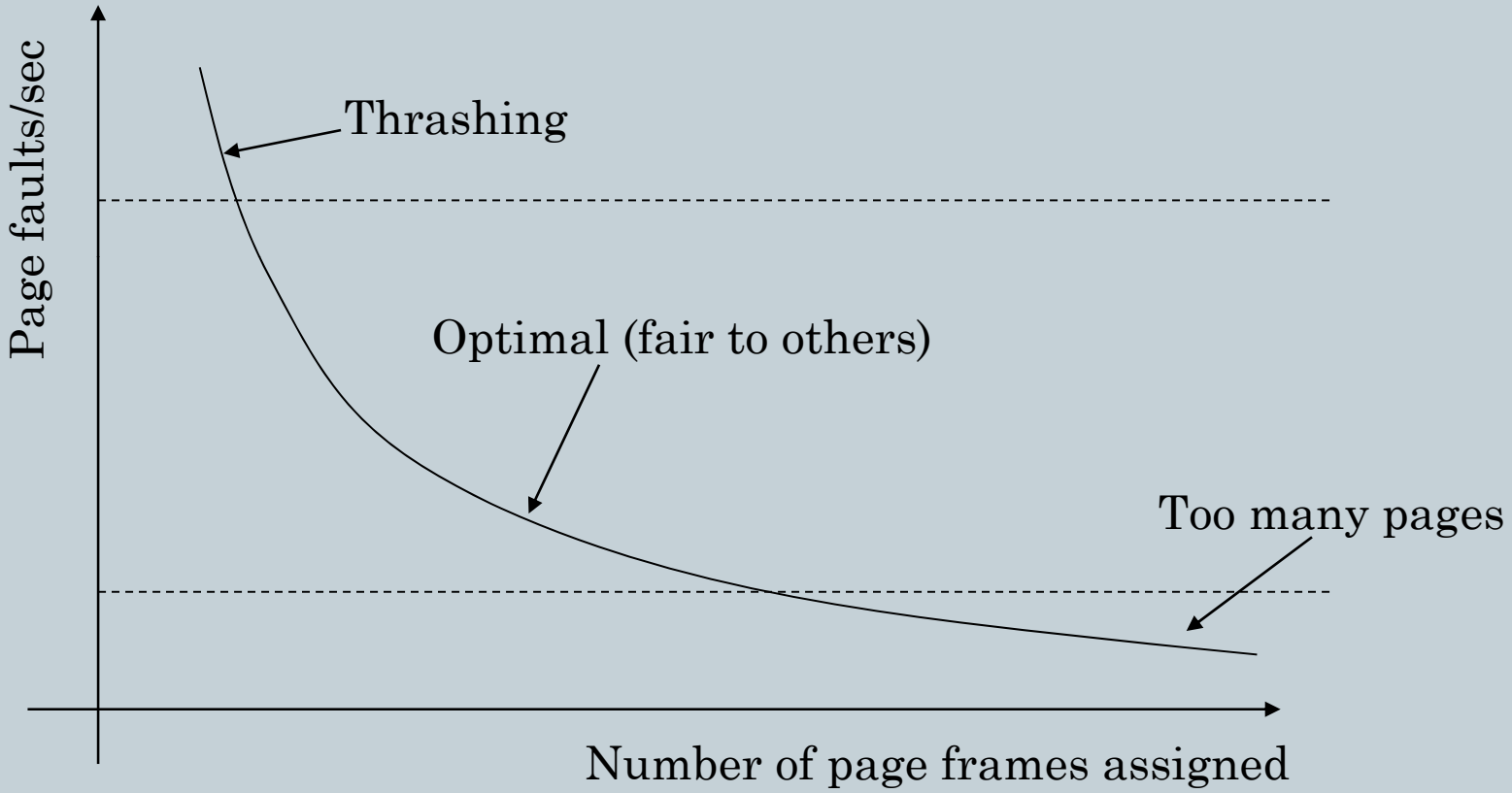
Algorithm	Comment
Optimal	Not implementable, useful for benchmarking
NRU (Not recently used)	Very crude
FIFO	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic (better implementation)
LRU (Least Recently Used)	Excellent but difficult to implement
NFU	Crude approx to LRU
Aging	Efficient in approximating LRU
Working set	Expensive to implement
WSClock	Good and efficient

Design issues

Design issues

- Local vs. global allocation policy
  - When a page fault occurs, whose page should the OS evict?
- Which process should get more or less pages?
  - Monitor the number of page faults for every process (PFF – page fault frequency)
  - For many page replacement algorithms, the more pages the less page faults

Page fault behavior



Load control

- If the WS of all processes > memory, there's *thrashing*
- E.g. the PFF says a process requires more memory but none require less
- Solution: swapping – swap a process out of memory and re-assign its pages to others

Page size

- Page size p, n pages of memory
- Average process size s, in pages s/p
- Each entry in the page table requires e bytes
- On average $p/2$ is lost (fragmentation)
- Internal fragmentation: how much memory is not used within pages
- Wasted memory: $p/2 + se/p$
- Minimizing it yields the optimal page size (under simplifying assumptions)

Two memories

- Separate data and program address spaces
- Two independent spaces, two paging systems
- The linker must know about the two address spaces

Other issues

- Shared pages, handle shared pages (e.g. program code)
  - Sharing data (e.g. shared memory)
- Cleaning policy
  - Paging algorithms work better if there are a lot of free pages available
  - Pages need to be swapped out to disk
  - Paging daemon (write pages to disk during spare time and evict pages if there are to few)

Page fault handling

1. Page fault, the HW traps to the kernel
    1. Perhaps registers are saved (e.g. stack)
2. Save general purpose microprocessor information (registers, PC, PSW, etc.)
3. The OS looks for which page caused the fault (sometimes this information is already somewhere within the MMU)
4. The system checks whether the process has access to the page (otherwise a protection fault is generated, and the process killed)
5. The OS looks for a free page frame, if none is found then the replacement algorithm is run
6. If the selected page is dirty (M=1) a disk write is scheduled (suspending the calling process)
7. When the page frame is clean, the OS schedules another transfer to read in the required page from disk
8. When the load is completed, the page table is updated consequently
9. The faulting instruction is backed up, the situation before the fault is restored, the process resumes execution

Segmentation

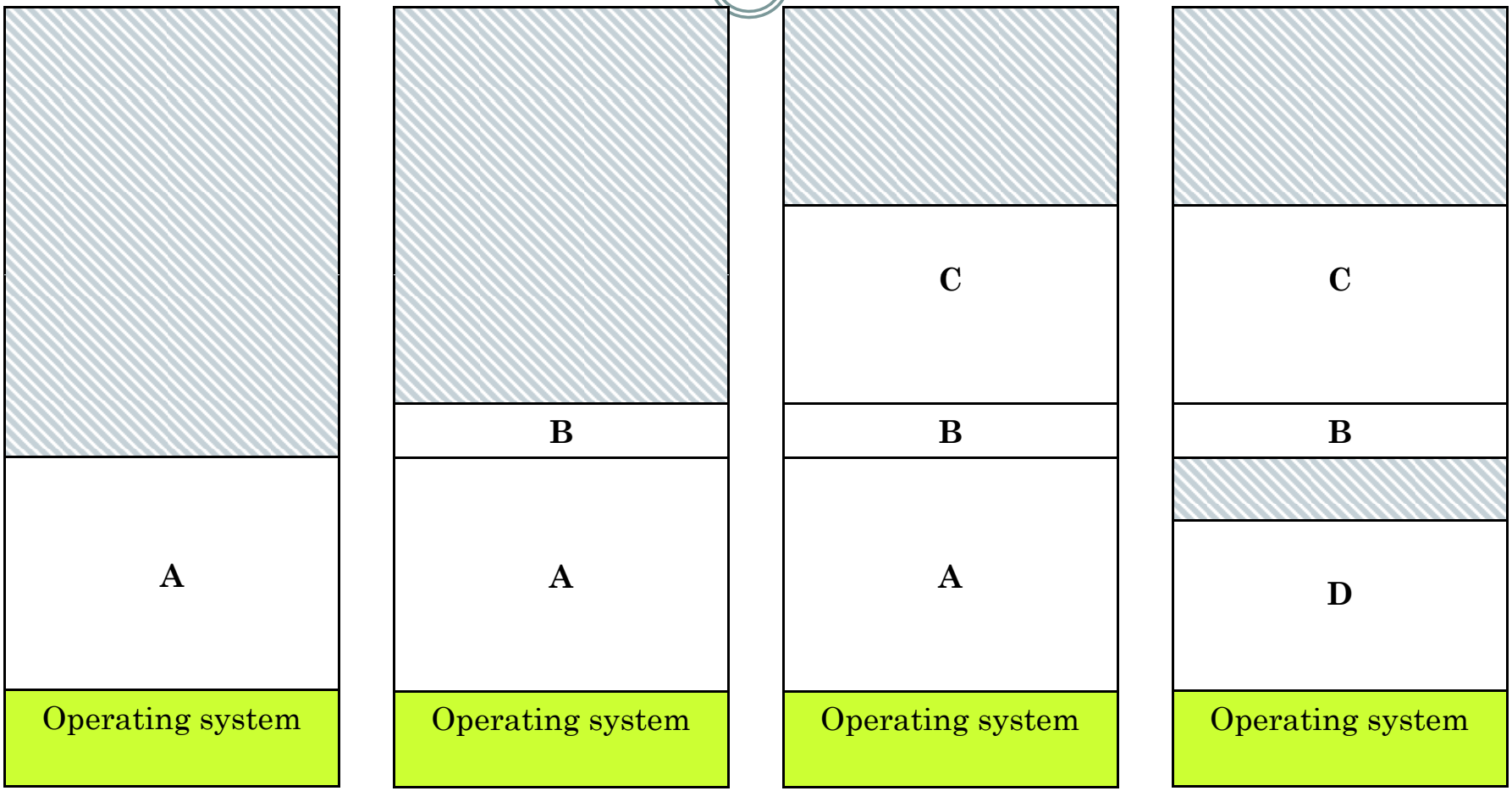
Why?

- Many separate address spaces (segments) (e.g. data, stack, code, and many others if needed)
- Each segment is separate (e.g. addresses from 0 to some MAX)
- Segments might have different lengths
- Segment number + address within segment
- Linking is simplified (libraries within different segments can assume addresses starting from 0) – e.g. if a part of the libraries is recompiled the remainder of the code is unaffected
- Shared library (DLL's) implementation is simpler (the sharing is simpler)

Comparing paging and segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuate be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Pure segmentations



Fragmentation

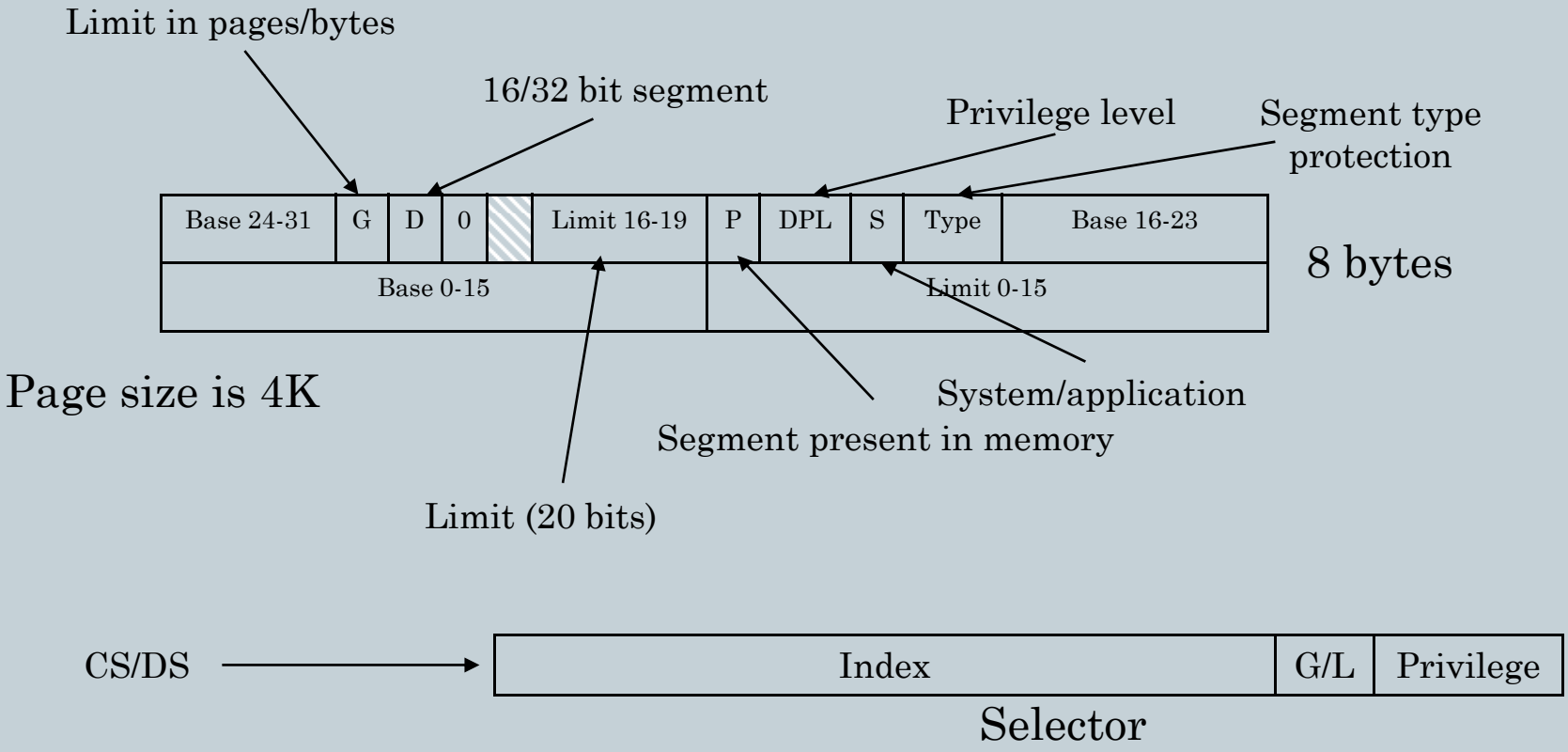
- External fragmentation:
  - Memory fragments not used (we've already seen this)
  - Memory wasted in unused holes

Segmentation + paging (Pentium)

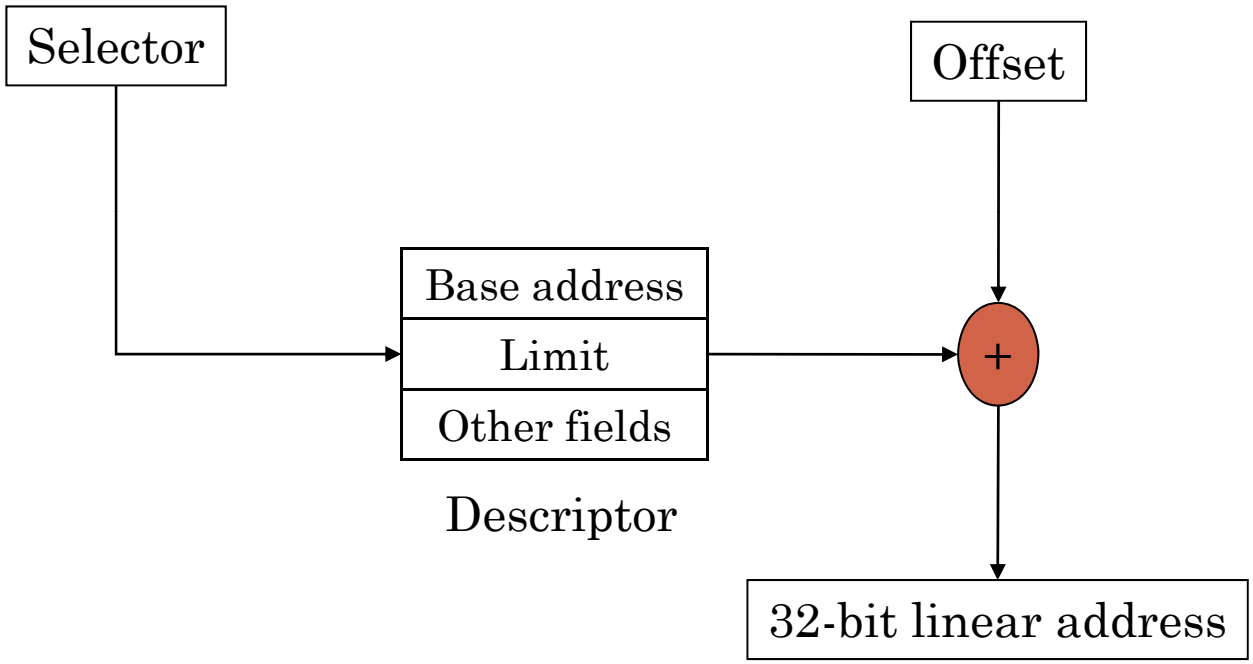
- 16K segments
- 1G 32bit words (DoubleWords)
- Two tables: LDT, GDT – Local (to the process) and global (to the processor) descriptor table
- To work with a segment the machine loads the segment number into a special register (CS, DS, etc.) – CS, DS are 16 bit registers
- The descriptor of the segment (see next slide)

The segment descriptor

- This is used by the microcode within the Pentium to work with segments

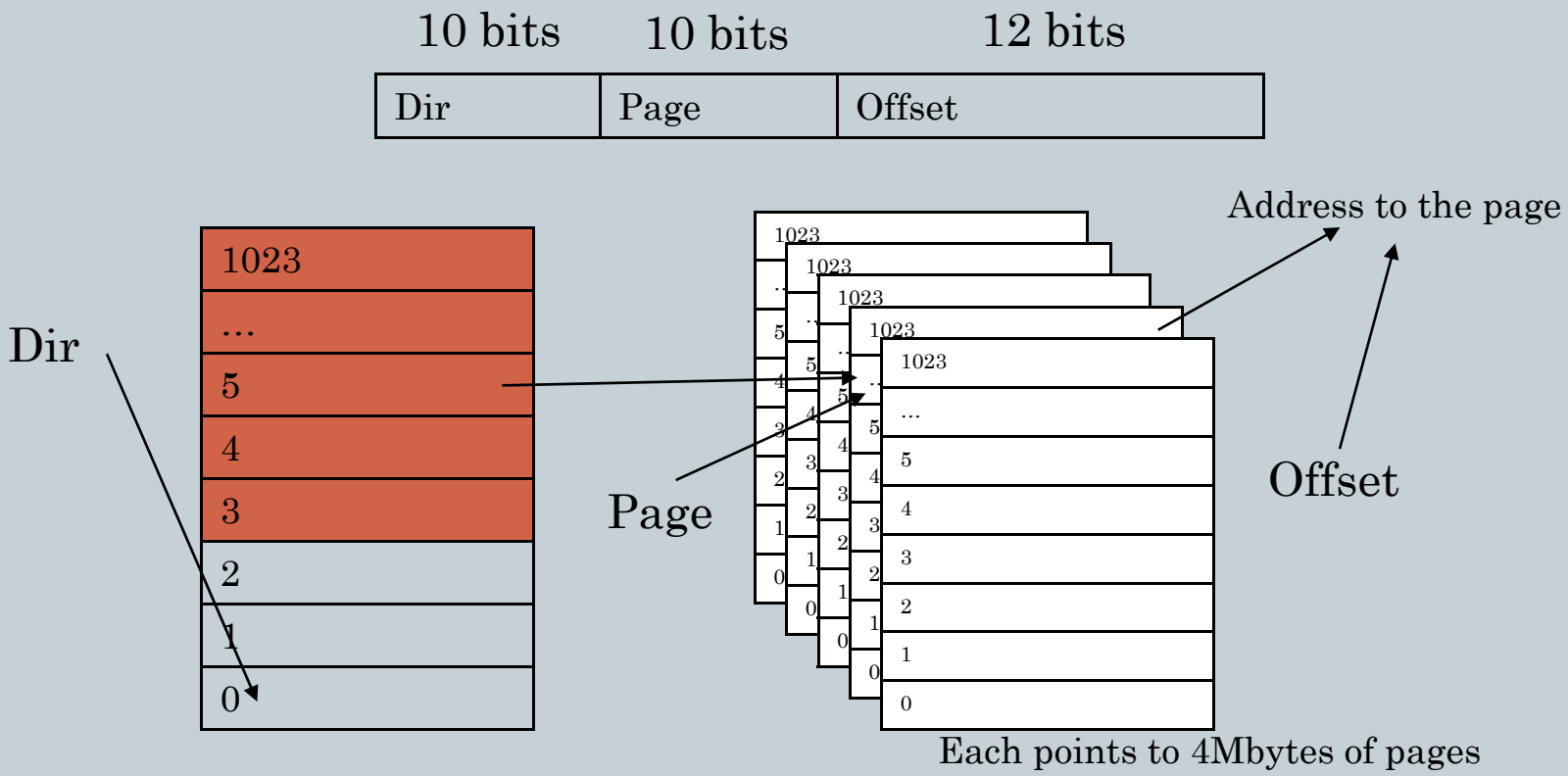


Getting the address



Paging on the Pentium

- 2-level page table in memory



More on the Pentiums

- TLB, to avoid repeated accesses to memory
- The whole thing can be used with just a single segment to obtain a linear 32bit address space
- Set base and limit appropriately
- Protection (a few bits)