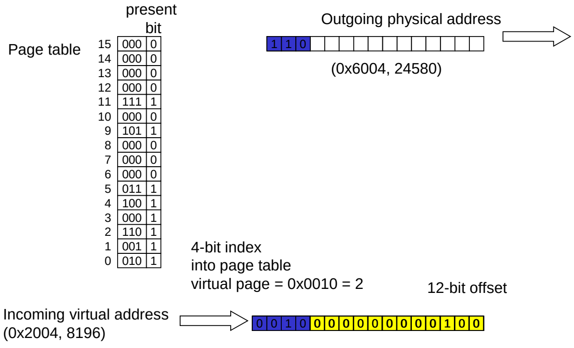
Paging (cont.)

30/10-2003

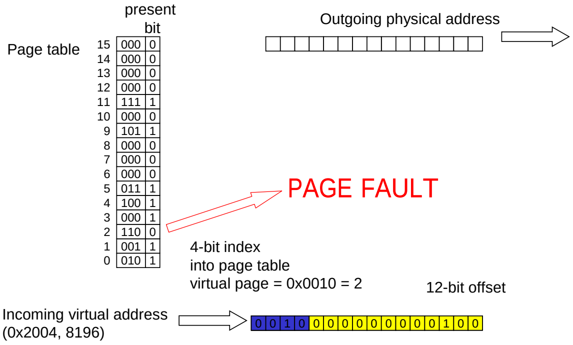
Pål Halvorsen

(including slides from Andrew Tanenbaum)

Memory Lookup



Memory Lookup



Page Fault Handling

1. Hardware traps to the kernel saving program counter and process state information
2. Save general registers and other volatile information
3. OS discover the page fault and tries to determine which virtual page is requested
4. OS checks if the virtual page is valid and if protection is consistent with access
5. Select a page to be replaced
6. Check if selected page frame is "dirty", i.e., updated
7. When selected page frame is ready, the OS finds the disk address where the needed data is located and schedules a disk operation to bring in into memory
8. A disk interrupt is executed indicating that the disk I/O operation is finished, the page tables are updated, and the page frame is marked "normal state"
9. Faulting instruction is backed up and the program counter is reset
10. Faulting process is scheduled, and OS returns to routine that made the trap to the kernel
11. The registers and other volatile information is restored and control is returned to user space to continue execution as no page fault had occured

Page Replacement Algorithms

- Page fault → OS has to select a page for replacement
  - Modified page → write back to disk
  - ***Not*** modified page → just overwrite with new data
- How do we decide which page to replace?
  → determined by the ***page replacement algorithm***
  → several algorithms exist:
  - Random
  - Other algorithms take into acount usage, age, etc.
    (e.g., FIFO, not recently used, least recently used, second chance, clock, ...)
  - which is best???

Optimal

- ***Best possible*** page replacement algorithm:
  - When a page fault occurs, all pages in memory are labeled with the number of instructions that will be executed before this page will be used again
  - The page with ***most*** instructions before reuse is replaced
- Easy to describe, but impossible to implement
  (OS cannot look into the future)
- Estimate by logging page usage on previous runs of process
- Useful to evaluate other page replacement algorithm

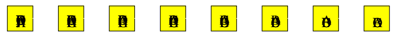
Not Recently Used (NRU)

- Two status bits associated with each page:
  - R → page referenced (read or written)
  - M → page modified (written)
- Pages belong to one of four set of pages according to the status bits:
  - **Class 0**: not referenced, not modified (R=0, M=0)
  - **Class 1**: not referenced, modified (R=0, M=1)
  - **Class 2**: referenced, not modified (R=1, M=0)
  - **Class 3**: referenced, modified (R=1, M=1)
- NRU removes a page at random from the lowest numbered, non-empty class
- Low overhead

First In First Out (FIFO)

- All pages in memory are maintained in a list sorted by age
- FIFO replaces the oldest page, i.e., the first in the list

Reference string: A B C D A E F G H I A J



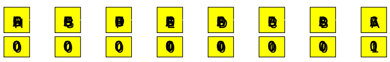
- Low overhead
- FIFO is rearly used in its pure form

Second Chance

- Modification of FIFO
- ***R*** bit: when a page is referenced again, the R bit is set, and the page will be treated as a newly loaded page

Reference string: A B C D A E F G H I

Page I will be inserted, find a page to page out by looking at the first page loaded:

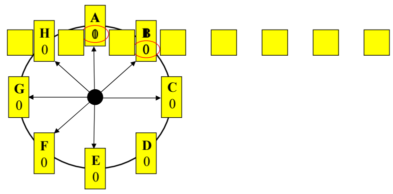


- *Second chance* is a reasonable algorithm, but inefficient because it is moving pages around the list

Clock

- More efficient way to implement ***Second Chance***
- Circular list in form of a clock
- Pointer to the oldest page:
  - R-bit = 0 → replace and advance pointer
  - R-bit = 1 → set R-bit to 0, advance pointer until R-bit = 0, replace and advance pointer

Reference string: A B C D A E F G H I



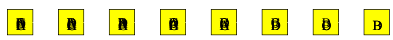
Least Recently Used (LRU)

- Replace the page that has the longest time since last reference
- Based on the observation that
  pages that are heavily used in the last few instructions will probably be used again in the next few instructions
- Several ways to implement this algoithm

Least Recently Used (LRU)

- LRU as a linked list:

Reference string: A B C D A E F G H A C I



- **Expensive** - maintaining an ordered list of all pages in memory:
  - most recently used at front, least at rear
  - update this list <u>every memory reference</u> !!

Least Recently Used (LRU)

- LRU by using *aging*:
  - ”reference counter” for each page
  - after a clock tick:
    - shift bits in the reference counter to the right (rightmost bit is deleted)
    - add a page’s referece bit in front of the reference counter (left)
  - page with *lowest* counter is replaced

Page		Clock tick 0 1 0 1 0 1 1	Clock tick 1 1 1 0 1 0 0	Clock tick 2 1 1 0 1 0 1	Clock tick 3 1 0 0 0 1 0	Clock tick 4 0 1 1 0 0 0
1	00000000	10000000	11000000	11100000	11110000	01111000
2	00000000	00000000	10000000	11000000	01100000	10110000
3	00000000	10000000	01000000	00100000	00010000	10001000
4	00000000	00000000	10000000	11000000	01100000	00110000
5	00000000	10000000	01000000	00100000	10010000	01001000
6	00000000	10000000	01000000	10100000	01010000	00101000

Least Recently Used (LRU)

- LRU as a matrix:
  - *N* pages → *N* x *N* matrix
  - Page *N* is referenced → row *N* is set (1)
    → column *N* is cleared (0)
  - Replace page with *lowest* ***row*** *value*

”Page frame” string: 1 2 3 4 3 2 1 4

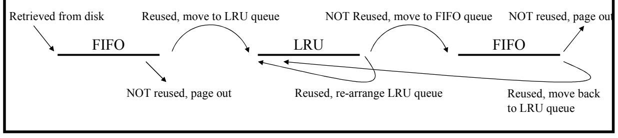
	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	1	1	1	0

Counting Algorithms

- LRU by using a reference counter
  - clear the counter when the page is referenced (counter = 0)
  - increase all counters each clock tick
  - replace the page with the *highest* counter
- *Not/Least Frequently Used* (N/LFU)
  - counter initially 0
  - increase the page’s counter *only if* it has been referenced during this clock tick
  - replace the page with *lowest* counter
- *Most Frequently Used* (MFU)
  - counter as LFU
  - replace the page with the *highest* counter (assuming low counters mean new, fresh pages)

LRU-K & 2Q

- **LRU-K:** bases page replacement in the *last K references* on a page [O’Neil et al. 93]
- **2Q:** uses 3 queues to hold much referenced and popular pages in memory [Johnson et al. 94]
  - 2 FIFO queues for seldom referenced pages
  - 1 LRU queue for much referenced pages

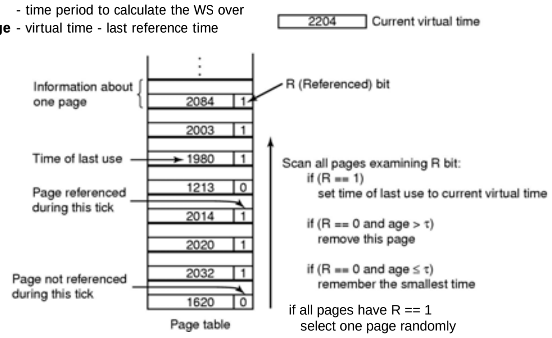


Working Set Model

- Working set:
  set of pages which a process is currently using
- Working set model:
  paging system tries to keep track of each process’ working set and makes sure that these pages is in memory before letting the process run
  → reduces page fault rate (prepaging)
- Defining the working set:
  - set of pages used in the last *k* memory references (must count backwards)
  - approximation is to use all references used in the last XX instructions

Working Set Page Replacement Algorithm

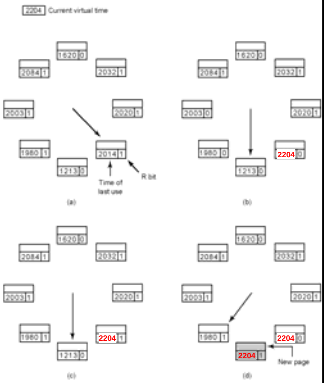
τ - time period to calculate the WS over
age - virtual time - last reference time



- Expensive - must search the whole page table

WSClock Page Replacement Algorithm

- Organize each page table entry as a clock
- As with clock - the page pointed to is examined first
  - R = 1:
    clear bit, set virtual time, continue **(b)**
  - R = 0: **(c)**
    - age < τ : continue to next
    - age > τ :
      - if page clean, replace **(d)**
      - othervice, write to disk and continue to next
- If all pointer comes back to start
  - writes are scheduled to clean pages (find first)
  - no scheduled writes (all in WS), several option
    - remove first clean
    - remove oldest
    - ...



Belady's Anomaly

- **<u>Question:</u>** *the more page frames, the fewer page faults?*
- Belady's anomaly gives a counter example using the FIFO replacement algorithm and buffers of 3 and 4 page frames:

Refernce string:	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**
Youngest page:	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest page:			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

⇒ **9 page faults**

Youngest page:	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Oldest page:				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

⇒ **10 page faults**

Stack Algorithms

- Observation:
  Every process generates a sequence of memory references as it runs where each memory reference corresponds to a virtual page
- Reference string**:** ordered list of page numbers (process' memory accesses)
- A paging system can be charachrized by 3 items:
  1. Reference string of the executing process
  2. Page replacement algorithm
  3. Number of page frames available in memory

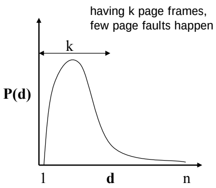
Stack Algorithms

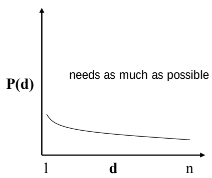
Reference string:	**0**	**2**	**1**	**3**	**5**	**4**	**6**	**3**	**7**	**4**	**7**	**3**	**3**	**5**	**5**	**3**	**1**	**1**	**1**	**7**	**2**	**3**	**4**	**1**
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	2	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	1	7	2	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	3	1	7	2
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	5	5	1	7
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	4	4	5	5
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	6	6	6	6
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults:	P	P	P	P	P	P	P		P					P			P				P		P	P
Distance string:	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	7	4	6	5

Distance string: a page reference is denoted by the distance from the top of the stack, i.e., the number of references since the page was referenced last

Distance String Properties

- The statistical properties of the distance string may show expected performance of the algorithm (probability density function)





Predicting Page Fault Rates

- $F_m = \sum C_k + C_\infty$, $m + 1 \le k \le n$
  F_m – # page faults occuring with a given distance string and *m* page frames
  C_k – # occurrences of *k,* i.e., # times we have a distance *k* since last reference
- Example - computation of the page fault rate:

Distance string: ∞ ∞ ∞ ∞ ∞ ∞ ∞ 4 ∞ 4 2 3 1 5 1 2 6 1 1 4 7 4 6 5

$C_1 = 4$ ← # times 1 occurs in the distance string / # page faults with 1 page frame → $F_1 = 20 \leftarrow C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_\infty$

$C_2 = 2$ ← # times 2 occurs in the distance string / # page faults with 2 page frame → $F_2 = 18 \leftarrow C_3 + C_4 + C_5 + C_6 + C_7 + C_\infty$

$C_3 = 1$ ← # times 3 occurs in the distance string / # page faults with 3 page frame → $F_3 = 17 \leftarrow C_4 + C_5 + C_6 + C_7 + C_\infty$

$C_4 = 4$ — $F_4 = 13 \leftarrow C_5 + C_6 + C_7 + C_\infty$

$C_5 = 2$ — $F_5 = 11 \leftarrow C_6 + C_7 + C_\infty$

$C_6 = 2$ — $F_6 = 9 \leftarrow C_7 + C_\infty$

$C_7 = 1$ — $F_7 = 8 \leftarrow C_\infty$

$C_\infty = 8$ — $F_8 = 8 \leftarrow C_\infty$

Locality

- Reference locality:
  - Time:
    pages that are referenced in the last few instructions will probably be referenced again in the next few instructions
  - Space:
    pages that are located close to the page being referenced will probably also be referenced

Demand Paging Versus Prepaging

- Demand paging:
  pages are loaded on demand, i.e., *after* a process needs it
  - Should be used if *we have no knowledge* about future references
  - Each page is loaded separatly from disk, i.e., results in many disk accesses
- Prepaging:
  prefetching data in advance, i.e., before use
  - Should be used if *we have knowledge* about future references
  - # page faults is reduced, i.e., page in memory when needed by a process
  - # disk accesses can be reduced by loading several pages in *one* I/O-operation

Allocation Policies

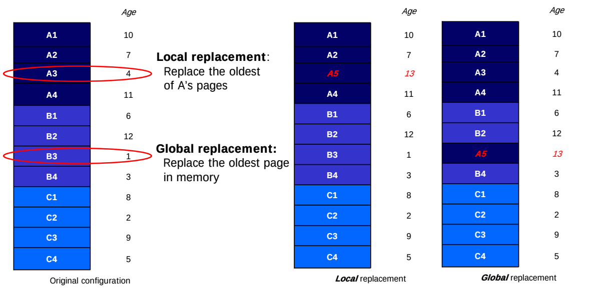
- How should memory be allocated among the competing runnable processes?
- Equal allocation:
  all processes get the same amount of pages
- Proportional allocation:
  amount of pages is depending on process size

Allocation Policies

- Local page replacement:
  consider only pages of own process when replacing a page
  - corresponds to equal allocation
  - can cause thrashing
  - multiple, identical pages in memory
- Global page replacemet:
  consider all pages in memory when replacing a page
  - corresponds to proportional allocation
  - better performance in general
  - monitoring of working set size and aging bits
  - data sharing

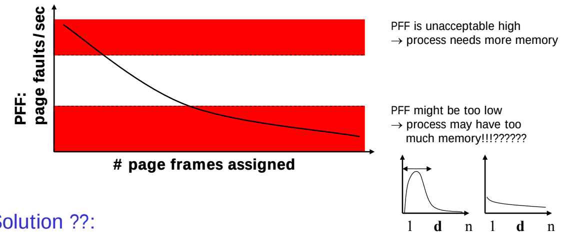
Allocation Policies

- Example: *local* versus *global* replacement
  insert page ***A5*** using age replacement



Allocation Policies

- Page fault frequency (PFF):
  Usually, more page frames → fewer page faults



Solution ??:
Reduce number of processes competing for memory
- reassign a page frame
- swap one or more to disk, divide up pages they held
- reconsider degree of multiprogramming

Page Size

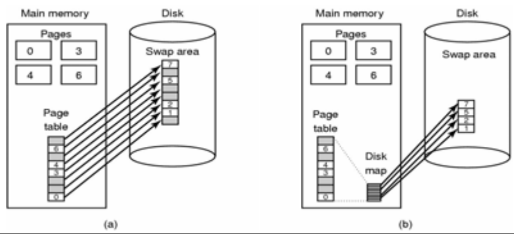
- Determining the optimum page size requires balancing several ***competing*** factors:
  - Data segment size ≠ n x page size → internal fragmentation (small size)
  - Keep in memory only data that is (currently) used (small size)
  - Disk operations (large size)
  - Page table size: access/load time and space requirements (large size)
  - Page replacement algorithm: operations per page (large size)
- Usual page sizes is 4 KB – 8 KB, but up to 64 KB is suggested for systems supporting "new" applications managing high data rate data streams like video and audio

Locking & Sharing

- Locking pages in memory:
  - I/O and context switches
  - Much used pages
  - ...
- Shared pages
  users running the same program at the same time, e.g., editor or compiler
  - Problem 1: not all pages are shareable
  - Problem 2: process swapping or termination
  - ...

Backing Store

- Backing store (disk management):
  Where on disk shall the pages be put when paged out?
  a) Special, in-advance allocated swap area
  (problem: growing processes)
  b) Allocate disk space when needed
  (problem: must hold all disk addresses in memory, time to allocate a new disk block)



Paging Daemons

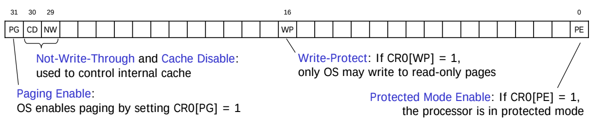
- Paging daemons:
  Background process which sleeps most of the time, but is for example awakened periodically or when the CPU is idle
  - Taking care that enough free page frames are available by writing back modified pages before they are reused
  - Prepaging

Paging on Pentium

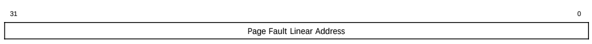
- In protected mode, the currently executing process have a 4 GB address space (2^{32}) – viewed as 1 M 4 KB pages
  - The 4 GB address space is divided into 1 K page groups (1 level – page directory)
  - Each page group has 1 K 4 KB pages (2 level – page table)
- Mass storage space is also divided into 4 KB blocks of information
- Uses control registers for paging information

Control Registers used for Paging on Pentium

- Control register 0 (CR0):

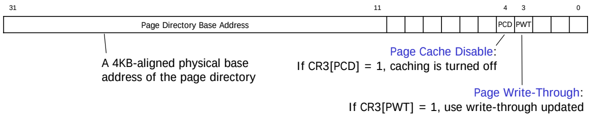


- Control register 1 (CR1) – does not exist, returns only zero
- Control register 2 (CR2)
  - only used if CR0[PG]=1 & CR0[PE]=1

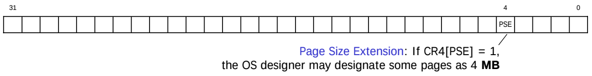


Control Registers used for Paging on Pentium

- Control register 3 (CR3) – page directory base address:
  - only used if CR0[PG]=1 & CR0[PE]=1

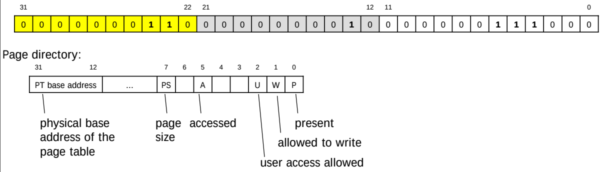


- Control register 4 (CR4):



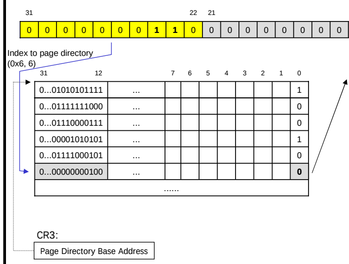
Pentium Memory Lookup

Incoming virtual address
(0x1402038, 20979768)



Pentium Memory Lookup

Incoming virtual address
(0x1402038, 20979768)

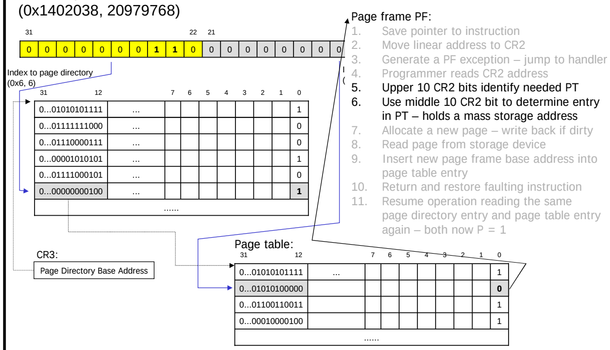


Page table PF:

1. Save pointer to instruction
2. Move linear address to CR2
3. Generate a PF exception – jump to handler
4. Programmer reads CR2 address
5. Upper 10 CR2 bits identify needed PT
6. Page directory entry is really a mass storage address
7. Allocate a new page – write back if dirty
8. Read page from storage device
9. Insert new PT base address into page directory entry
10. Return and restore faulting instruction
11. Resume operation reading the same page directory entry again – now P = 1

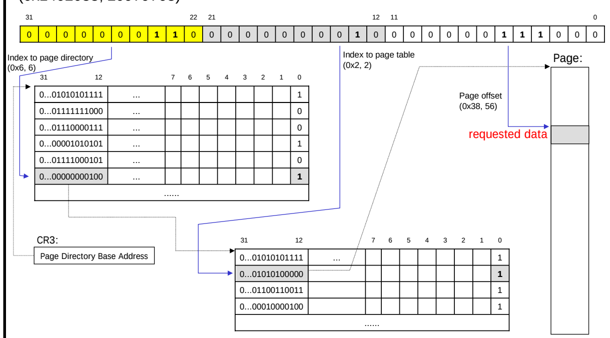
Pentium Memory Lookup

Incoming virtual address
(0x1402038, 20979768)



Pentium Memory Lookup

Incoming virtual address
(0x1402038, 20979768)



Page Fault Causes

- Page directory entry's P-bit = 0:
  page group's directory (page table) not in memory
- Page table entry's P-bit = 0:
  requested page not in memory
- Attempt to write to a read-only page
- Insufficient page-level privilege to access page table or frame
- One of the reserved bits are set in the page directory or table entry