Demand Paging

Demand Paging

- Code pages are stored in a memory-mapped file on the backing store
  - some are currently in memory–most are not
- Data and stack pages are also stored in a memory-mapped file
- OS determines what portion of VAS is mapped in memory
  - physical memory serves as cash for memory-mapped file on backing store

Demand Paging:

Touching Valid but not Present Address

TLB Miss (HW managed)

Page Table walk

Page fault (Present bit P not set in Page Table)

Exception to kernel to run page-fault handler

Convert VA to file offset

Allocate page frame (evict page if needed)

Initiate disk block read into page frame

Disk interrupt when transfer completes

Set P to 1 and update PFN for page's PTE

Resume process at faulting instruction

TLB miss

Page Table walk – success!

TLB updated

Execute instruction

Allocating a Page Frame

- When free frames fall below Low Watermark, do until they climb above High Watermark:
  - Select "victim" page VP to evict (a policy question)
  - Find all PTEs referring to frame VP maps to
    - if page frame was shared
  - Set P bit in each such PTE to 0
  - Remove any TLB entries that included VP's victim frame
    - the PTE they are caching is now invalid!
  - Write changes to page back to disk
- Transferring pages in bulk allows to reduce transfer time

Page Replacement

- Local vs Global replacement
  - Local: victim chosen from frames of process experiencing page fault
    - fixed allocation per process
  - Global: victim chosen from frames allocated to any process
    - variable allocation per process
- Many replacement policies
  - Random, FIFO, LRU, Clock, Working set, etc.
- Goal: minimizing number of page faults

How do we pick a victim?

- We want:
  - low fault-rate for pages
  - page faults as inexpensive as possible
- We need:
  - a way to compare the relative performance of different page replacement algorithms
  - some absolute notion of what a “good” page replacement algorithm should accomplish

Comparing Page Replacement Algorithms

- Record a trace of the pages accessed by a process
  - E.g. 3,1,4,2,5,2,1,2,3,4 (or c,a,d,b,e,b,a,b,c,b)
- Simulate behavior of page replacement algorithm on trace
- Record number of page faults generated

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	d
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	c	c	c	c
	3	d	d	d	d	d	e	e	e	e	e	e
Faults							X					X
Time page needed next						a = 7 b = 6 c = 9 d = 10					a = ∞ b = 11 c = 13 e = 15	

b d c b e

FIFO Replacement

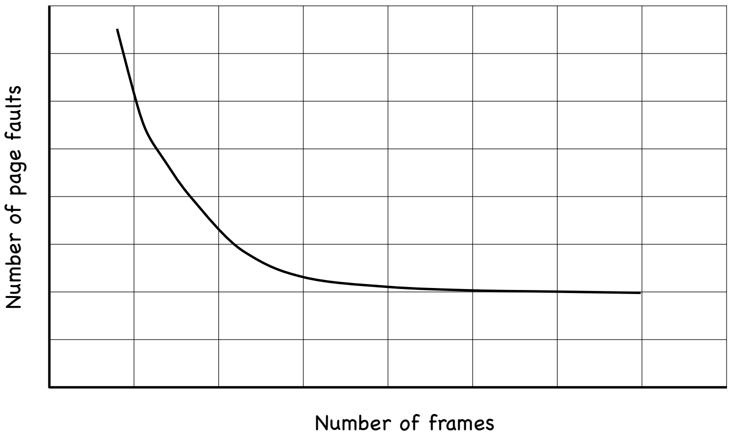
- Replace pages in the order they come into memory

Assume:

a @ -3
b @ -2
c @ -1
d @ 0

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	e	e	e	e	e	d
	1	b	b	b	b	b	b	b	a	a	a	a
	2	c	c	c	c	c	c	c	c	b	b	b
	3	d	d	d	d	d	d	d	d	d	c	c
Faults							X		X	X	X	X

+ Frames
- Page Faults



For example...

FIFO

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Request			a	b	c	d	a	b	e	a	b	c	d	e
Page Frames	0		a	a	a	d	d	d	e	e	e	e	e	e
	1			b	b	b	a	a	a	a	a	c	c	c
	2				c	c	c	b	b	b	b	b	d	d
Faults			X	X	X	X	X	X	X			X	X	

- 3 frames - 9 page faults!

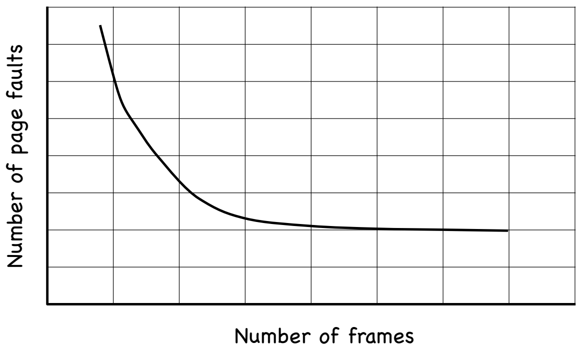
Belady's Anomaly

FIFO

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Request			a	b	c	d	a	b	e	a	b	c	d	e
Page Frames	0		a	a	a	a	a	a	e	e	e	e	d	d
	1			b	b	b	b	b	a	a	a	a	a	e
	2				c	c	c	c	c	b	b	b	b	b
	3					d	d	d	d	d	d	c	c	c
Faults			X	X	X	X			X	X	X	X	X	X

- 4 frames - 10 page faults!

+ Frames
- Page Faults?



- Yes, but only for stack page replacement policies
  - set of pages in memory with n frames is a subset of set of pages in memory with n+1 frames

Locality of Reference

- If a process access a memory location, then it is likely that
  - the same memory location is going to be accessed again in the near future (temporal locality)
  - nearby memory locations are going to be accessed in the future (spatial locality)
- 90% of the execution of a program is sequential
- Most iterative constructs consist of a relatively small number of instructions

LRU: Least Recently Used

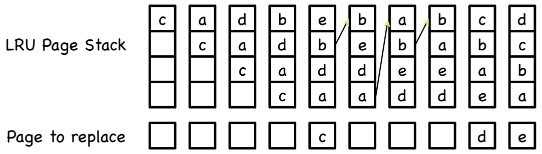
- Replace page not referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	c	c
Faults							X				X	X
Time page last used						a = 2 b = 4 c = 1 d = 3				a = 7 b = 8 e = 5 d = 3	a = 7 b = 8 e = 5 c = 9	

Implementing LRU

- Maintain a "stack" of recently used pages

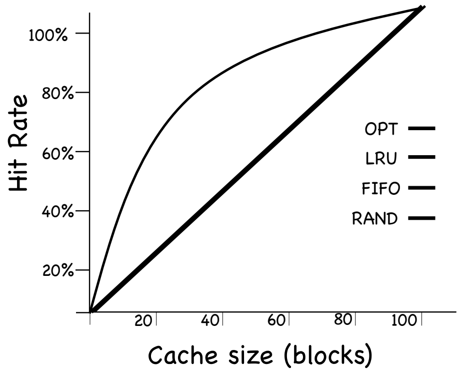
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	c	c
Faults							X				X	X



LRU Page Stack	c	a	d	b	e	b	a	b	c	d
		c	a	d	b	e	b	a	b	c
			c	a	d	d	e	e	a	b
				c	a	a	d	d	e	a
Page to replace					c				d	e

No-Locality Workload

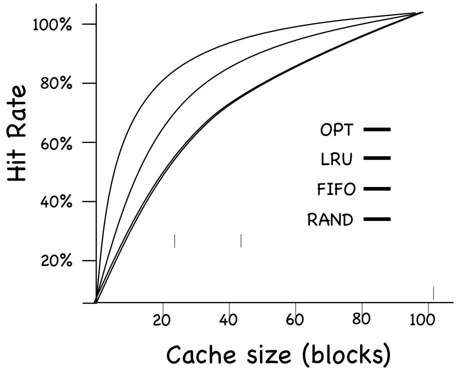
- Workload references 100 unique pages over time
- 10,000 references
- Next page chosen at random



What do you notice?

80%-20% Workload

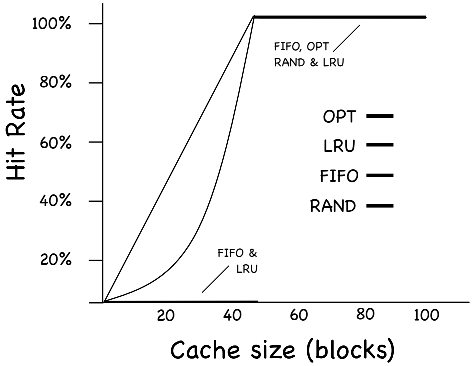
- 10,000 references, but with some locality
  - 80% of references to 20% of the pages
  - 20% of references to the remaining 80% of pages.



What do you notice?

Sequential-in-a-loop Workload

- 10,000 references
- We access 50 pages in sequence, then repeat, in a loop.



What do you notice?

Implementing LRU

- Add a (64-bit) timestamp to each page table entry
  - HW counter incremented on each instruction
  - Page table entry timestamped with counter when referenced
  - Replace page with lowest timestamp

Implementing LRU

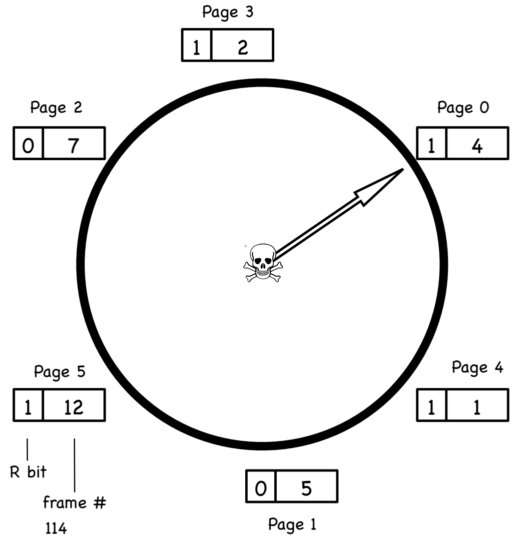
- Add a (64-bit) timestamp to each page table entry
  - HW counter incremented on each instruction
  - Page table entry timestamped with counter when referenced
  - Replace page with lowest timestamp

- Approximate LRU through aging
  - keep a k-bit tag in each table entry
  - at every "tick":
    - Shift tag right one bit
    - Copy Referenced (R) bit in tag
    - Reset Refereced bits to 0
  - If needed, evict page with lowest tag

	R bits at Tick 0	R bits at Tick 1	R bits at Tick 2	R bits at Tick 4	R bits at Tick 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 0	10000000	11000000	11100000	11110000	01111000
Page 1	00000000	10000000	11000000	01100000	10110000
Page 2	10000000	01000000	00100000	00100000	10001000
Page 3	00000000	00000000	10000000	01000000	00100000
Page 4	10000000	11000000	01100000	10110000	01011000
Page 5	10000000	01000000	10100000	01010000	00101000

The Clock Algorithm

- Organize pages in memory as a circular list
- When page is referenced, set its reference bit R to 1
- On page fault, look at page the hand points:
  - if R = 0:
    - evict the page
    - set R bit of newly loaded page to 1
  - else (R = 1): clear R
  - advance hand



Clock Page Replacement

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a	d	b	e	b	a	b	c	d
Page Frames 0	a	a	a	a	a	e	e	e	e	e	d
1	b	b	b	b	b	b	b	b	b	b	b
2	c	c	c	c	c	c	c	a	c	a	a
3	d	d	d	d	d	d	d	d	d	c	c
Faults						X		X		X	X
Page table entries for resident pages			1 a 1 b 1 c 1 d			1 e 0 b 0 c 0 d	1 e 1 b 0 c 0 d	1 e 0 b 1 a 0 d	1 e 1 b 1 a 0 d	1 e 1 b 1 a 1 c	1 d 0 b 0 a 0 c

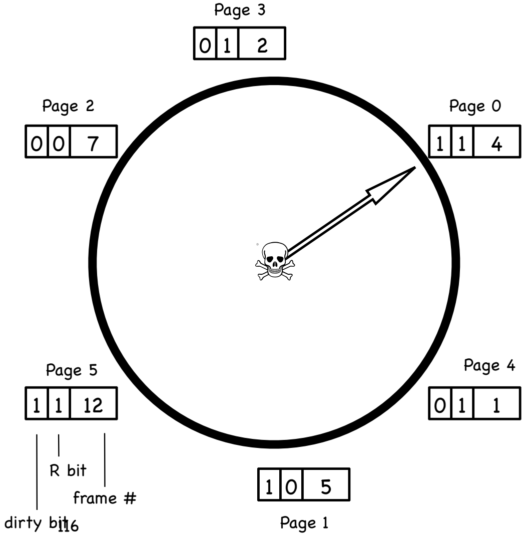
Hand clock: ☐

The Second Chance Algorithm

- Dirty pages get "second chance" before eviction
  - synchronously replacing dirty pages is expensive!

If clock's hand points at P and this is P's state…		…this is what happens	
dirty	R	dirty	R
0	0	replace page	
0	1	0	0
1	0	0	0
1	1	1	0

[Start asynchronous transfer of dirty page to disk]



Second Chance
Page Replacement

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames 0	a	a	a	a	a	a	a	a	a	a	a
1	b	b	b	b	b	b	b	b	b	b	d
2	c	c	c	c	c	e	e	e	e	e	e
3	d	d	d	d	d	d	d	d	d	c	c
Faults						X				X	X

Page table entries for resident pages	01 a				11 a	00 a	00 a	11 a		11 a	00 a
	01 b				11 b	00 b	01 b	01 b		01 b	01 d
	01 c				01 c	01 e	01 e	01 e		01 e	00 e
	01 d				01 d	00 d	00 d	00 d		01 c	00 c

Hand clock: ☐

Async copy: ☐