Virtual Memory

Today

- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms

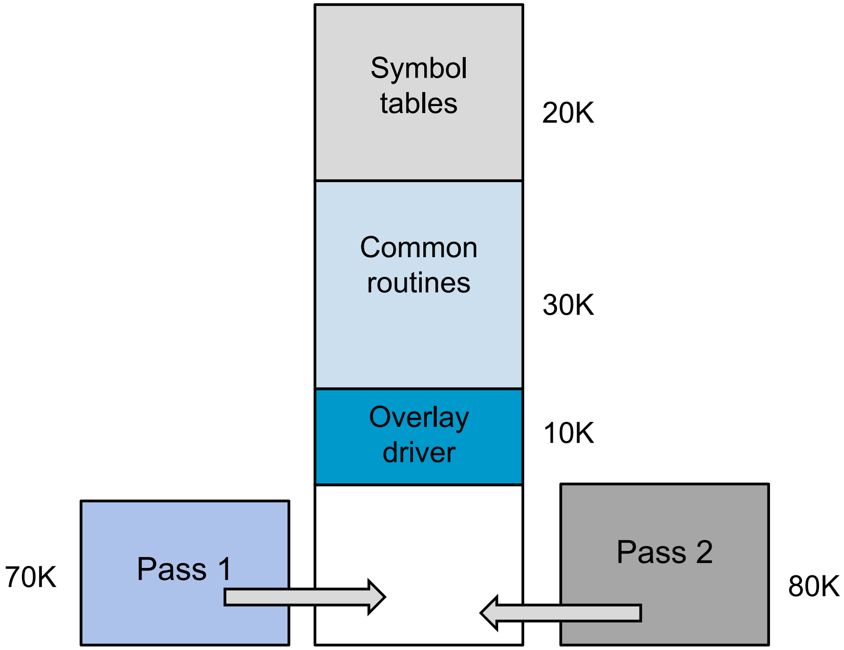
Before virtual memory

- Handling processes >> than allocated memory
- Keep in memory only what's needed
- Overlay approach: implemented by user
  - Easy on the OS
  - Hard on the programmer

Overlay for a two-pass assembler:

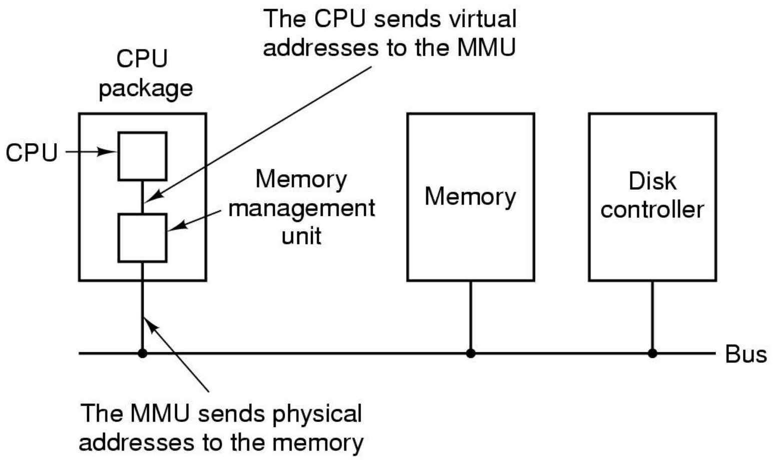
Pass 1	70KB
Pass 2	80KB
Symbol Table	20KB
Common Routines	30KB
Total	200KB

Two overlays: 120 + 130KB



Virtual memory

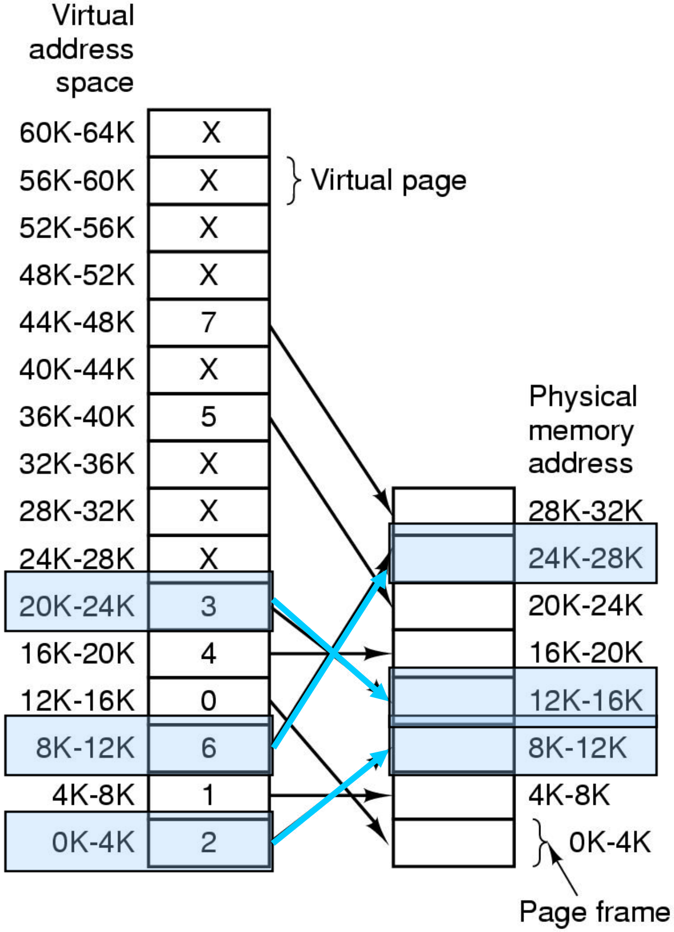
- Hiding the complexity
  - The combined size of program, data and stack >> physical memory available for it
  - OS keeps parts of program in use in memory, rest in disk
  - Set of addresses a program can generate – virtual address space
  - Translate that to physical limitation – physical address
  - Doing the translation – MMU
- Most common approach –
  - Virtual address space split into pages
  - Physical memory into page frames
  - Page & page frames = size (512B … 64KB)



Pages, page frames and tables

With

- 64KB virtual address space
- 4KB pages
- 32KB physical address space
- 16 pages and 8 page frames



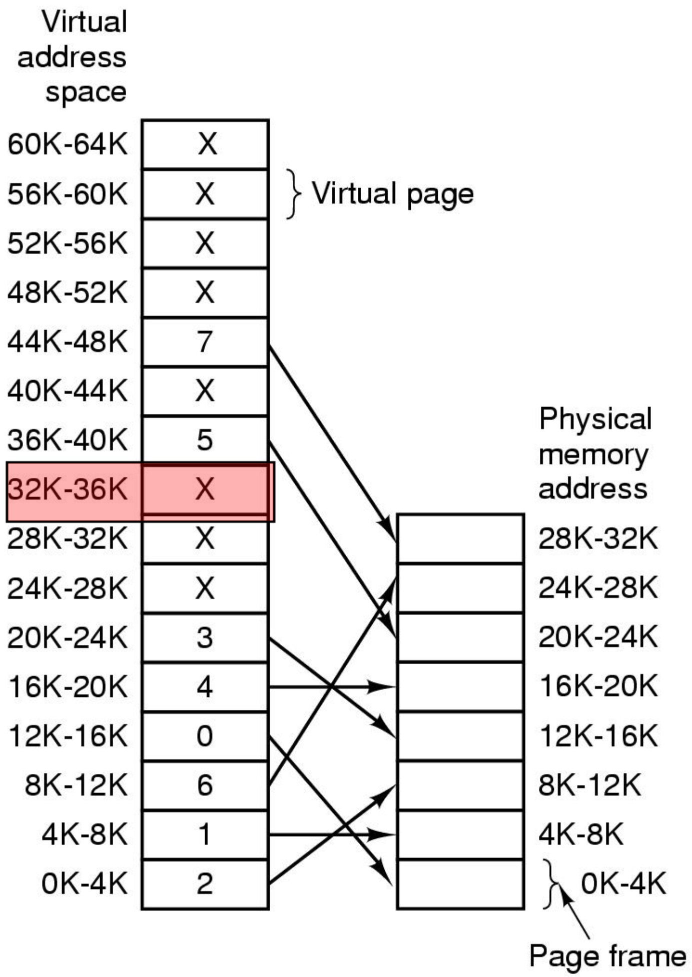
Try to access :

- **MOV REG, 0**
  Virtual address 0
  Page frame 2
  Physical address 8192
- **MOV REG, 8192**
  Virtual address 8192
  Page frame 6
  Physical address 24576
- **MOV REG, 20500**
  Virtual address 20500 (20480 + 20)
  Page frame 3
  Physical address 20+12288

Since virtual memory >> physical memory

- Use a present/absent bit
- MMU checks –
  - If not there, “page fault” to the OS (trap)
  - OS picks a victim (?)
  - … sends victim to disk
  - … brings new one
  - … updates page table

MOVE REG, 32780
Virtual address 32780
Virtual page 8, byte 12 (32768+12)
Page is unmapped – page fault!



Page replacement algorithms

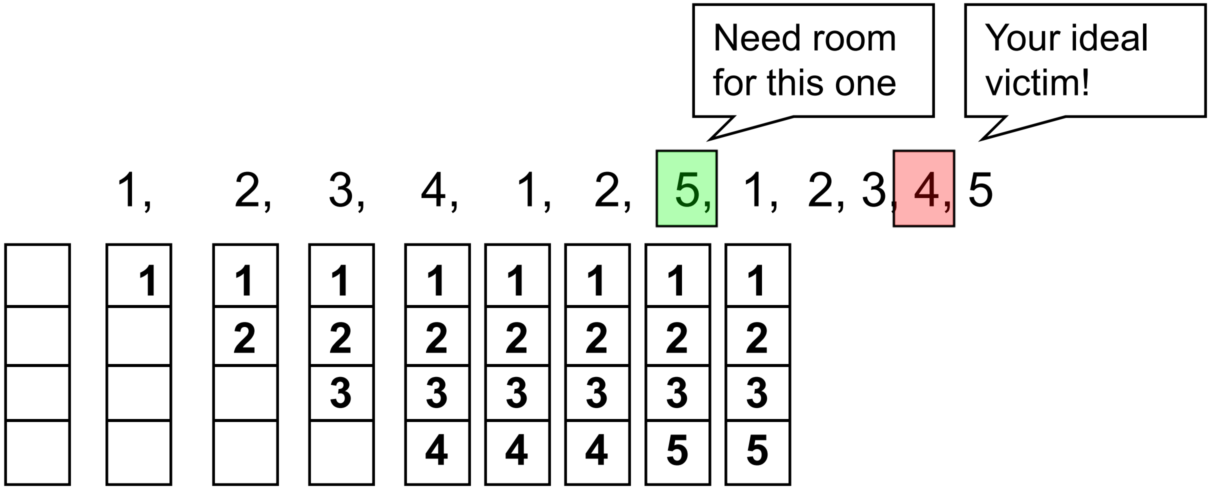
- Virtual address space >> physical one
- OS uses main mem as (page) cache – demand paging
- Page fault – cache miss
  - Need room for new page? Page replacement algorithm
  - What's your best candidate for removal?
- What do you do with victim page?
  - Modified page must first be saved
  - Unmodified one just overwritten
  - Better not to choose an often used page
    - It will probably need to be brought back in soon
- Try to avoid thrashing
  - OS wastes most of the time moving pages around
  - Fix the algorithm, swap out somebody, get more memory

Why does demand paging work?

- Locality
  - Temporal locality – location recently referenced tend to be referenced again soon
  - Spatial locality – locations near recently referenced are more likely to be referenced soon
- Locality means paging could be infrequent
  - Once you brought a page in, you'll use it many times
  - Some issues that may play against you
    - Degree of locality of application
    - Page replacement policy and application reference pattern
    - Amount of physical memory and application footprint

Optimal algorithm (Belady's algorithm)

- The best page to replace is the one you'll never need again
  - Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - Logging page use on previous runs of process
  - Although impractical, useful for comparison



FIFO algorithm

- Maintain a linked list of all pages – in order of arrival
- Victim is first page of list
  – Maybe the oldest page will not be used again …
- Disadvantage
  – But maybe it will – the fact is, you have no idea!
  – Increasing physical memory *might* increase page faults (Belady’s anomaly, we’ll come back to this)

A, B, C, D, A, B, E, A, B, C, D, E

	A	B	C	D	A	B	E	E	E	C	D	D
		A	B	C	D	A	B	B	B	E	C	C
			A	B	C	D	A	A	A	B	E	E

Not recently used (NRU) algorithm

- Each page has *Reference* and *Modified* bits
  - Set when page is referenced, modified
  - R bit set means recently referenced, so you must clear it every now and then
- Pages are classified

How can this occur?

R	M	Class
0	0	Not referenced, not modified (0,0 → 0)
0	1	Not referenced, modified (0,1 → 1)
1	0	Referenced, but not modified (1,0 → 2)
1	1	Referenced and modified (1,1 → 3)

- NRU removes page at random
  - from lowest numbered, non-empty class
- Easy to understand, relatively efficient to implement and sort-of OK performance

Second chance algorithm

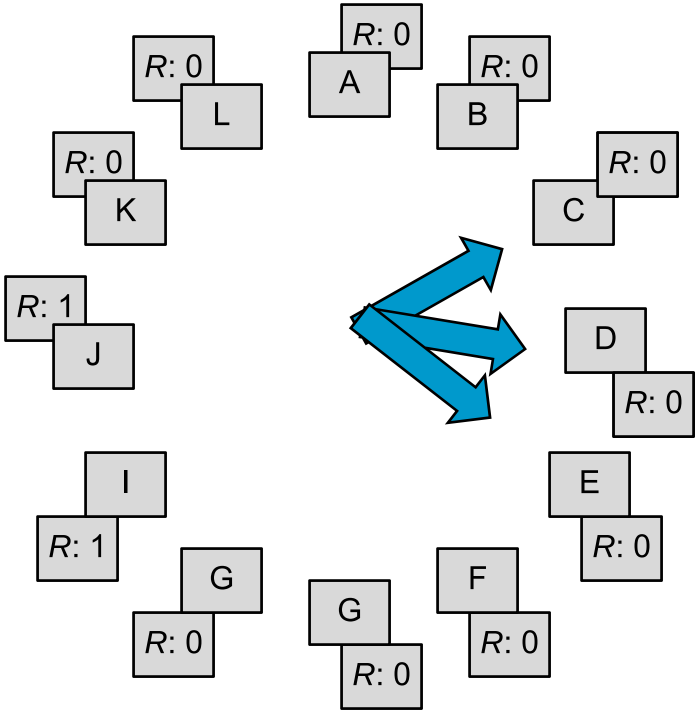
- Simple modification of FIFO – look at the R bit
- Operation of second chance
  - Pages sorted in FIFO order
  - Page list if fault occurs at time 20, A has R bit set (time is loading time)

	Page	Time	R
Most recently loaded	H	18	X
	G	15	X
	F	14	X
	E	12	X
	D	8	X
	C	7	X
	B	3	0
Oldest page	**A**	**0**	**1**

Page	Time	R
A	**20**	**0**
H	18	X
G	15	X
F	14	X
E	12	X
D	8	X
C	7	X
B	3	0

Clock algorithm

- Quit moving pages around – move a pointer?
- Same as Second chance but for implementation
  - When page fault
  - Look at page pointed at by hand
    - If R = 0, evict page
    - If R = 1. clear R & move hand



Least recently used (LRU) algorithm

- Pages used recently will used again soon
  – Throw out page unused for longest time
- Must keep a linked list of pages
  – Most recently used at front, least at rear
  – Update this list every memory reference !!
- Alternatively keep counter in page table entry
  – Choose page with lowest value counter
  – Periodically zero the counter

A second HW LRU implementation

- Use a matrix – *n* page frames – *n x n* matrix
- Page *k* is reference
  - Set all bits of row *k* to 1
  - Set all bits of column *k* to 0
- Page of lowest row is LRU

0,1,2,3,2,1,0,3,2

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

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

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

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

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

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

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

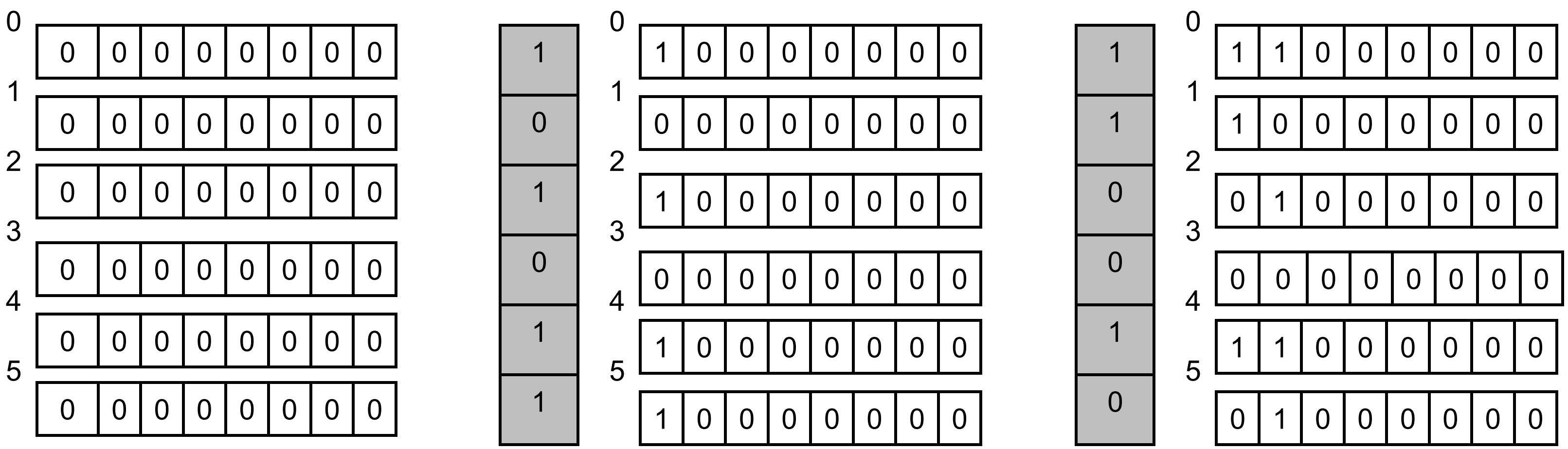
0	1	2	3	
0	1	1	1	0
0	0	1	1	1
0	0	0	1	2
0	0	0	0	3

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

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

Simulating LRU in software

- Not Frequently Used
  - Software counter per page
  - At clock interrupt – add R to counter for each page
  - Problem - it never forgets!
- Better – Aging
  - Push R from the left, drop bit on the right
  - How is this *not* LRU? One bit per tick & a finite number of bits per counter

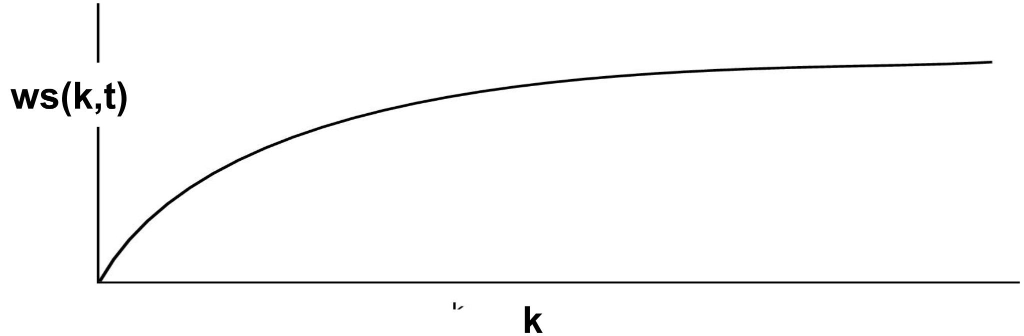


Working set algorithm

- Most programs exhibit *locality of reference* – over a short time, just a few common pages
- Working set
  - Set of pages used by the *k* most recent memory references
  - *ws(k, t)* – size of the working set at time *t* (*k* is the working set window size)
  - *What bounds ws(k, t) as you increase k?*
  - *How could you use this knowledge to reduce turnaround time?*

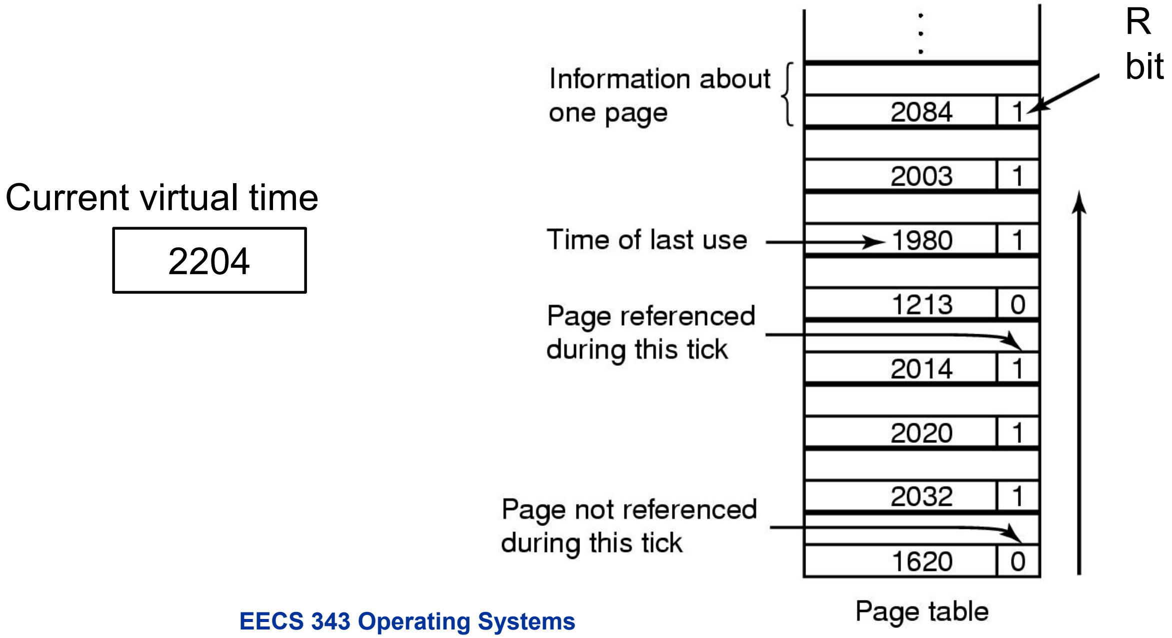
Clearly $ws(k_i,t) \le ws(k_j,t)$

for $i < j$



Working set algorithm

- Working set and page replacement
  - Victim – a page *not* in the working set
- At each clock interrupt – scan the page table
  - R = 1? Write Current Virtual Time (CVT) into *Time of Last Use*
  - R = 0? CVT – *Time of Last Use* > *Threshold ? out!* else see if there's someone and evict oldest (w/ R=0)
  - If all are in the working set (all R = 1) random

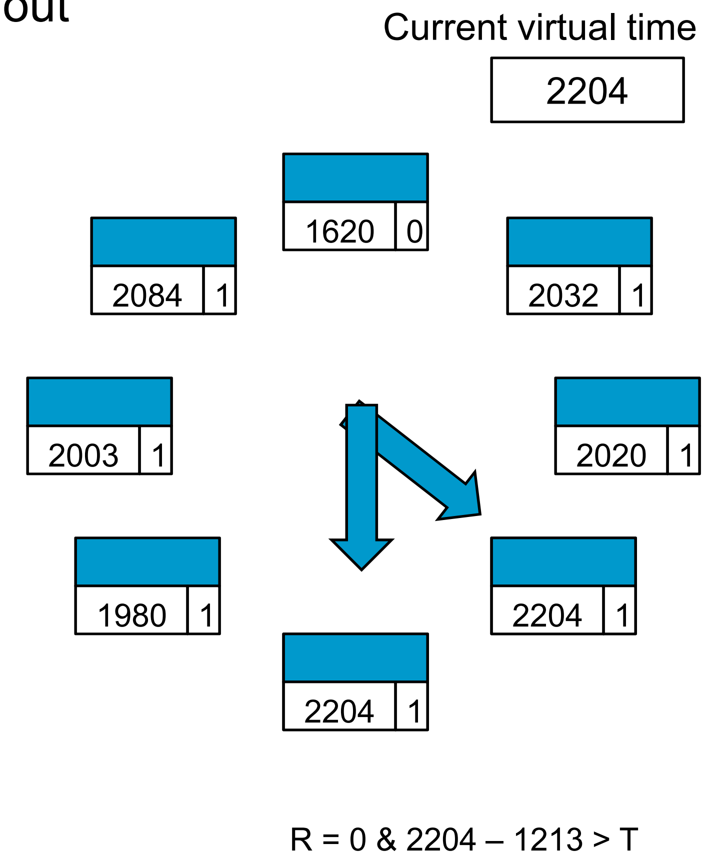


WSClock algorithm

- Problem with WS algorithm – Scans the whole table
- Combine clock & working set
  - If R = 1, same as working set
  - If R = 0, if age > T and page clean, out
  - If dirty, schedule write and check next one
  - If loop around,

    There's 1+ write scheduled – you'll have a clean page soon

    There's none, pick any one



Belady's anomaly

- The more page frames the fewer page faults, right?
  - FIFO with 3 page frames
  - FIFO with 4 page frames

All page frames initially empty

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	0	1	4					
		0	1	2	3	0	1					
			0	1	2	3	0					
	P	**P**	**P**	**P**	**P**	**P**	**P**					

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	3	3	4					
		0	1	2	2	2	3					
			0	1	1	1	2					
				0	0	0	1					
	P	**P**	**P**	**P**								

Belady's anomaly

- The more page frames the fewer page faults, right?
  - FIFO with 3 page frames
  - FIFO with 4 page frames

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

9 page faults

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

10 page faults

Modeling page replacement algorithms

- Paging system can be characterized by
  - Page replacement algorithm
  - a reference string
  - # page frames
- Abstract interpreter with
  - Internal array, M, to keep track of memory state
    - Size of (M) = # virtual pages, n
  - Split in two parts
    - Top m entries, for m pages frame
    - The bottom part (n – m) for pages that have been referenced but eventually paged out
  - Initially M is empty

An example using LRU

Reference to a page (5) out of the blue box → page fault

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
Pages in page frames	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
Pages paged out to disk					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									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	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Stack algorithms

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									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	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

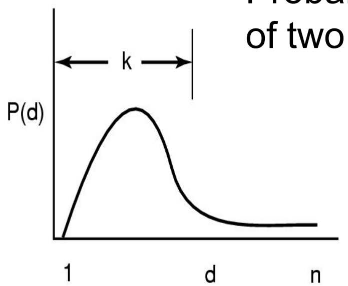
Model works well with other algorithms. Particularly interesting …

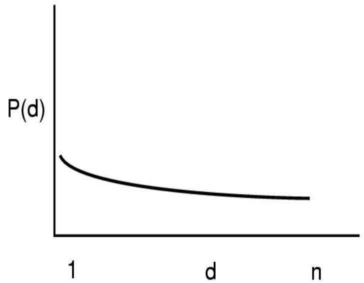
Stack algorithm: $M(m,r) \subseteq M(m+1,r)$

Pages in memory with *m* pages frames and after *r* memory references

Distance string – each page reference denoted by the distance from top of the stack where the page was located (if not yet referenced: ∞)

Probability density function of two distance strings





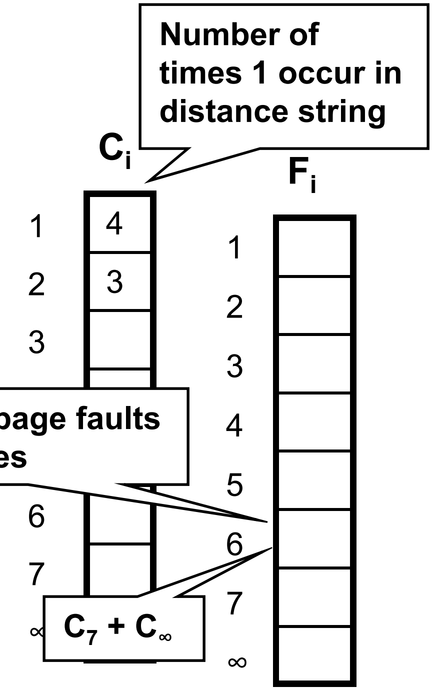
Distance string & page faults

Computation of page fault rate from distance string

C_i – number of occurrences of i in distance string

F_m – number of page faults with m frames

Reference string		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
			0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
				0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
					0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
						0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
							0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
								0	0	2	2	2	2	2	2	2	2	2	2	2	2	2			
										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	
Distance string		∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3



$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

Distance string & page faults

Computation of page fault rate from distance string

C_i – number of occurrences of i in distance string

F_m – number of page faults with m frames

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									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	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

i	C_i
1	4
2	3
3	3
4	3
5	2
6	1
7	0
∞	8

i	F_i
1	20
2	17
3	14
4	11
5	9
6	8
7	8
∞	8

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

Next time ...

- You now understand how things work, i.e. the mechanism …
- Next time we'll consider design and implementation issues for paging systems – or things you want/need to pay attention for good performance