Virtual Memory Design and Implementation

Today

- Page replacement algorithms
- Some design and implementation issues

Next

- Last on virtualization – VMMs

How can any of this 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

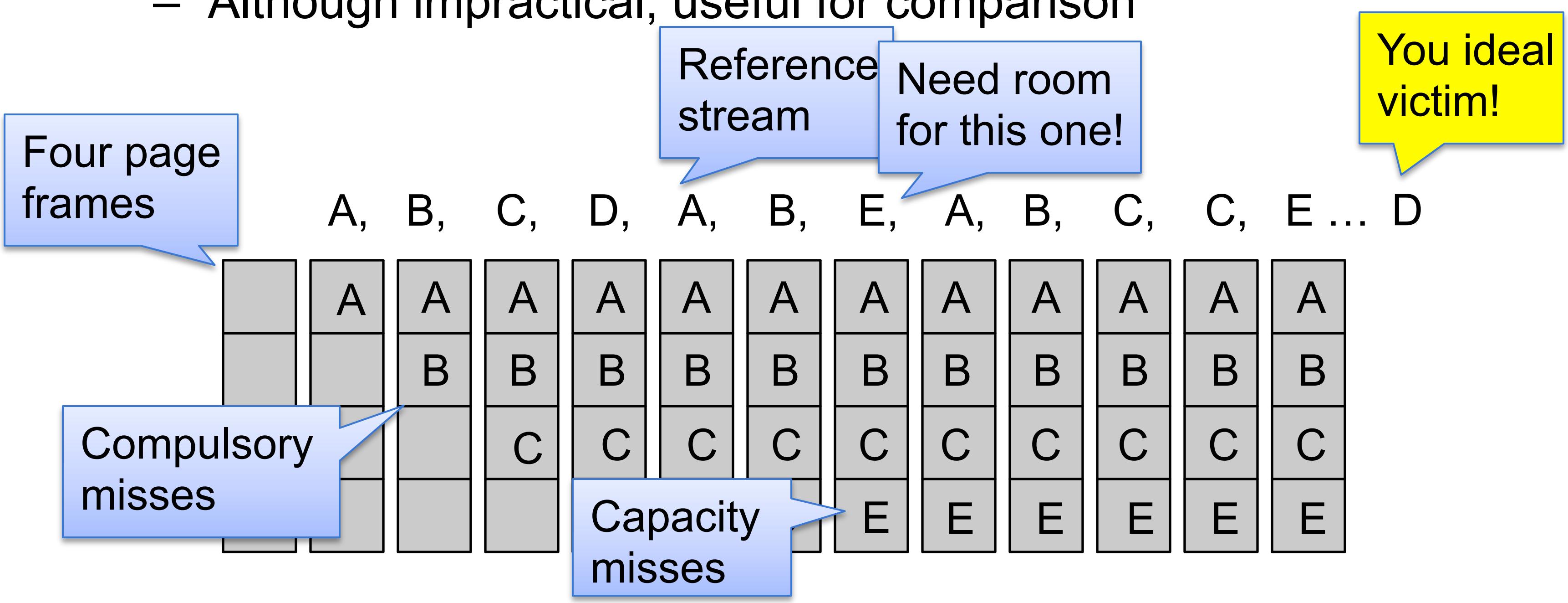
Page replacement algorithms

- OS uses main memory as (page) cache
  - If only load *when* reference – demand paging
- Page fault – cache miss
  - Need room for new page? Page replacement algorithm
  - What’s your best candidate for removal?
    - The one you will never touch again – duh!
- What do you do with victim page?
  - If modified, must be saved, otherwise just overwritten
  - Better not to choose an often used page

- Let’s look at some
  - For now, assume a process pages against itself, using a fixed number of page frames

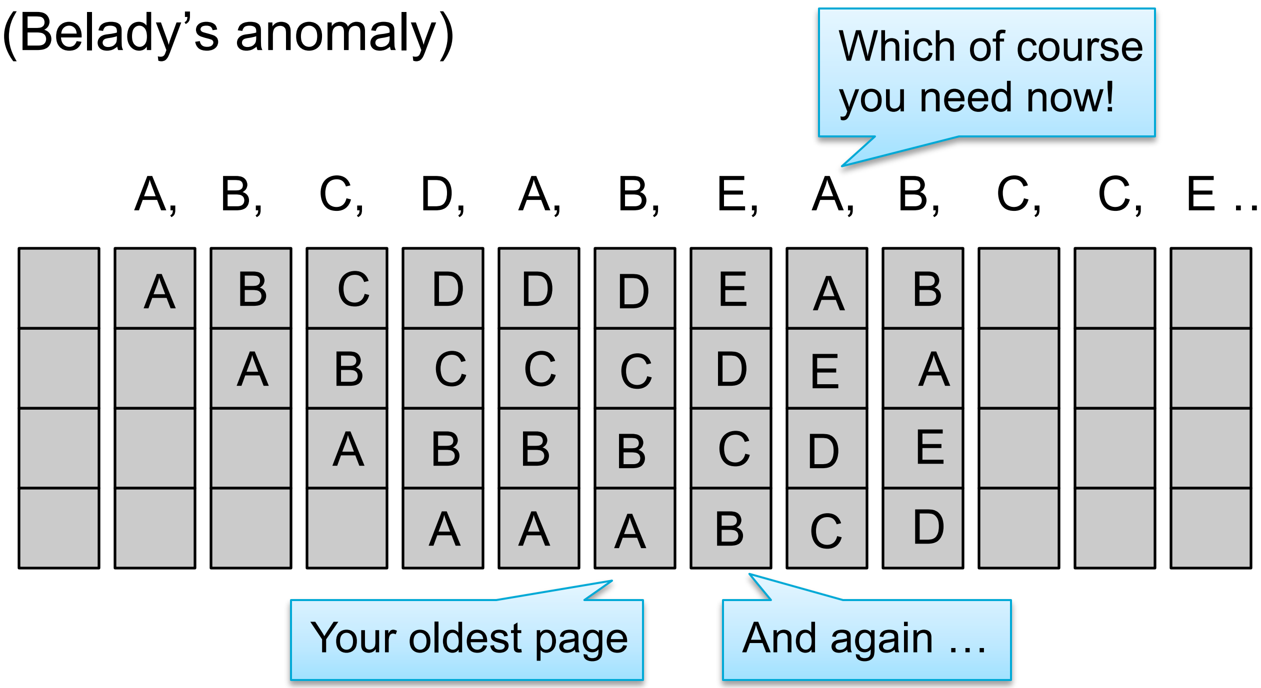
Optimal algorithm (Belady's algorithm)

- *If you could only tell!* – best page to replace, 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)



Least recently used (LRU) algorithm

- Pages used recently will be used again soon
  - Throw out page unused for longest time
  - Idea: past experience is a decent predictor of future behavior
    - LRU looks at the past, Belady's wants to look at the future
    - *How is LRU different from FIFO?*
- Must keep a linked list of pages
  - Most recently used at front, least at rear
  - Update this list every memory reference!!
    - Too expensive in mem. bandwidth, algorithm execution time, etc

Second chance algorithm

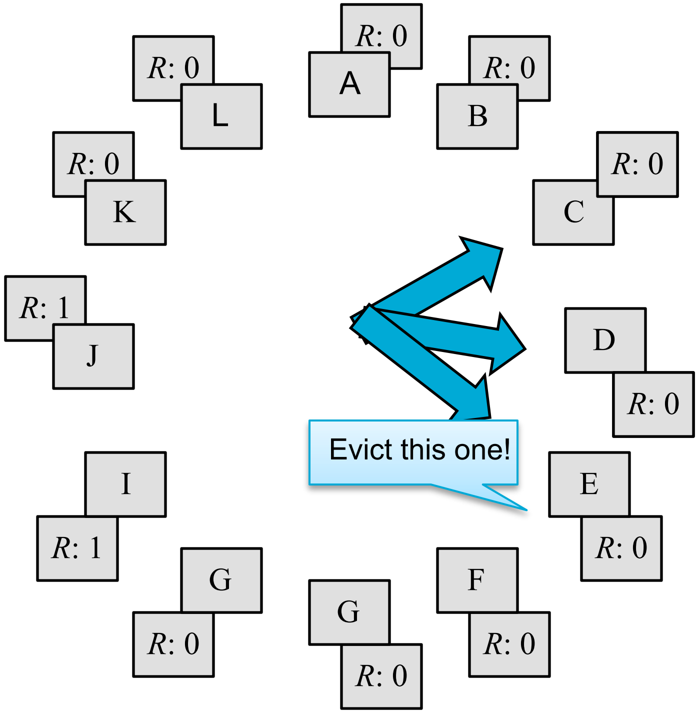
- Simple modification of FIFO
  - Avoid throwing out a heavily used page – look at the R bit
- Operation of second chance
  - Pages sorted in FIFO order
  - If it has been used, gets another chance – move it to the end of the list of pages, clear R and update timestamp
  - Page list if fault occurs at 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

- Second chance is reasonable but inefficient
  - Quit moving pages around – move a pointer?
- Same as Second chance but for implementation
  - Keep all pages in a circular list, as a clock, with the hand pointing to the oldest page
  - When page fault
    - Look at page pointed at by hand
      - If R = 0, evict page
      - If R = 1. clear R & move hand



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)

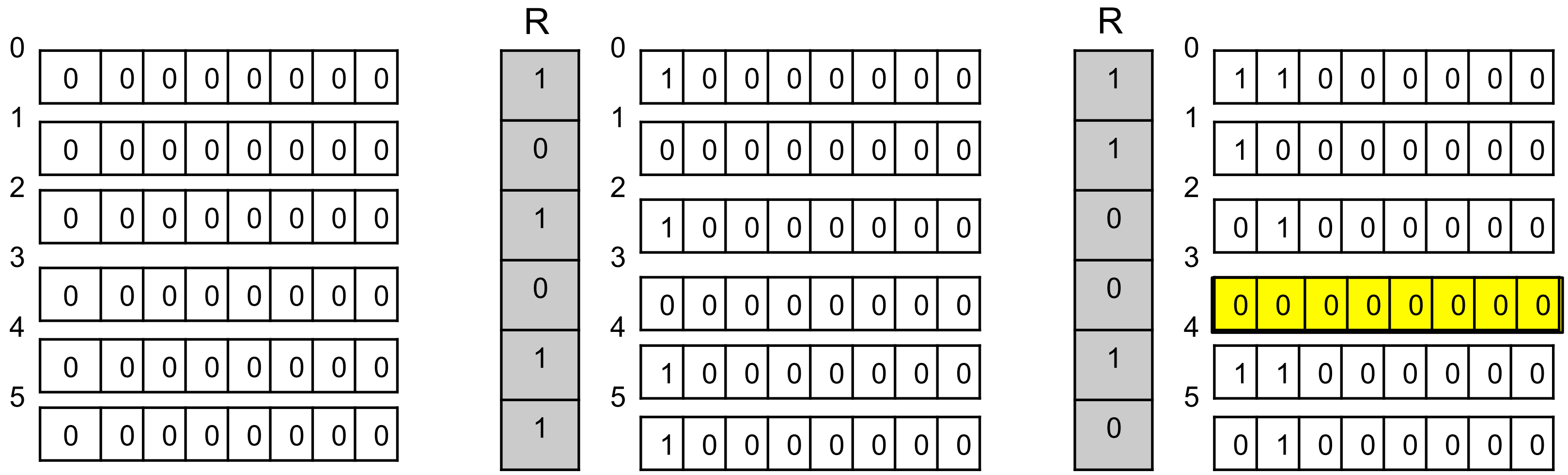
- NRU removes page at random
  - from lowest numbered, non-empty class
- Easy to understand, relatively efficient to implement and sort-of OK performance

Approximating LRU

- With some extra help from hardware
  - Keep a counter in PTE
  - Equipped hardware with a counter, ++ after each instruction
  - After each reference, update PTE counter for the referenced with hardware counter
  - Choose page with lowest value counter
- In software, Not Frequently Used
  - Software counter associated with each page
  - At clock interrupt – add R to counter for each page
  - Problem - it never forgets!

Approximating LRU

- 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



And now a short break …

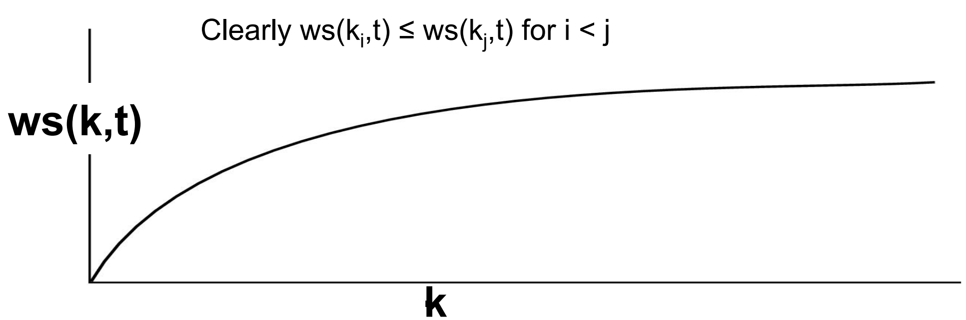
Working set

- Most programs show *locality of reference*
  - *Over a short time, just a few common pages*
- Working set
  - Models the dynamic locality of a process' memory usage
  - i.e. the set of pages currently needed by a process
- Intuitively, working set must be in memory, otherwise you'll experience heavy faulting (thrashing)
  - What does it mean 'how much memory does program x need?" – what is program x average/worst-case working set size?

Working set

- Demand paging
  - Simplest strategy, load page when needed
- Can you do better knowing a process WS?
  - How could you use this to reduce turnaround time? *Prepaging*
- Working set definition
  - *ws(k,t) = {pages p such that p was referenced in the k most recent memory references}* (*k* is WS window size)

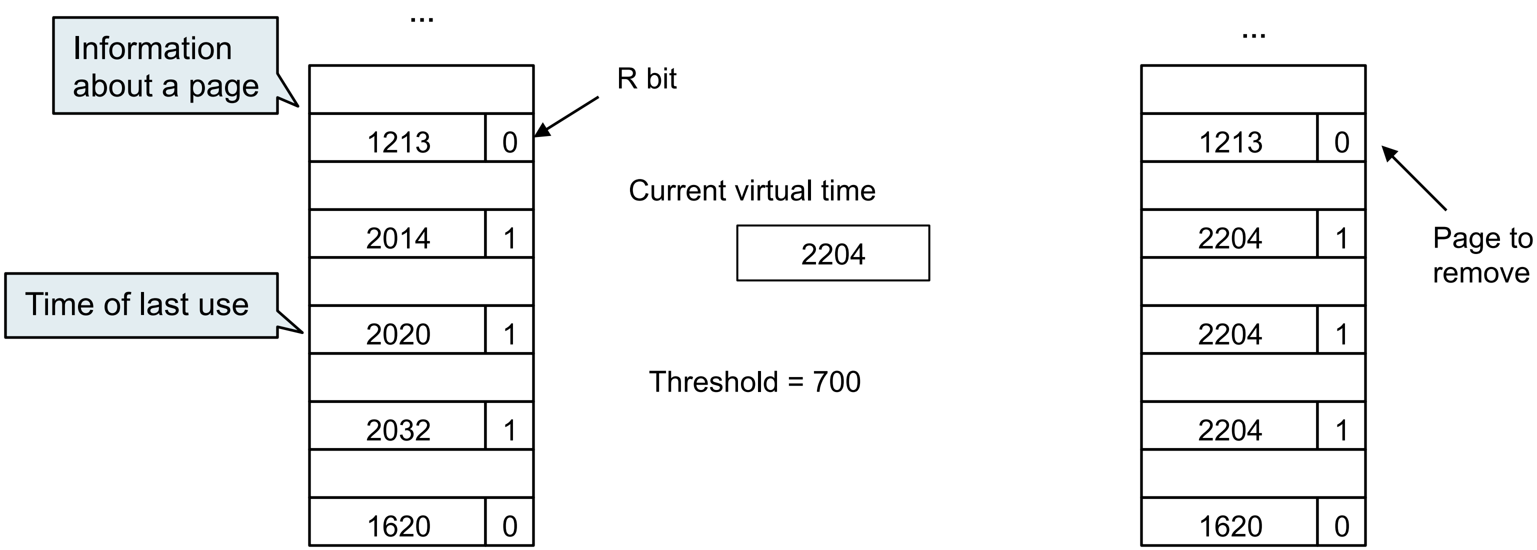
What bounds ws(k, t) as you increase k?



  - A more practical definition – instead of *k* reference pages, *t* msec of execution time

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 some other page and evict oldest (w/ R=0)
  - If all are in the WS (all *R* = *1*), random, preferably clean

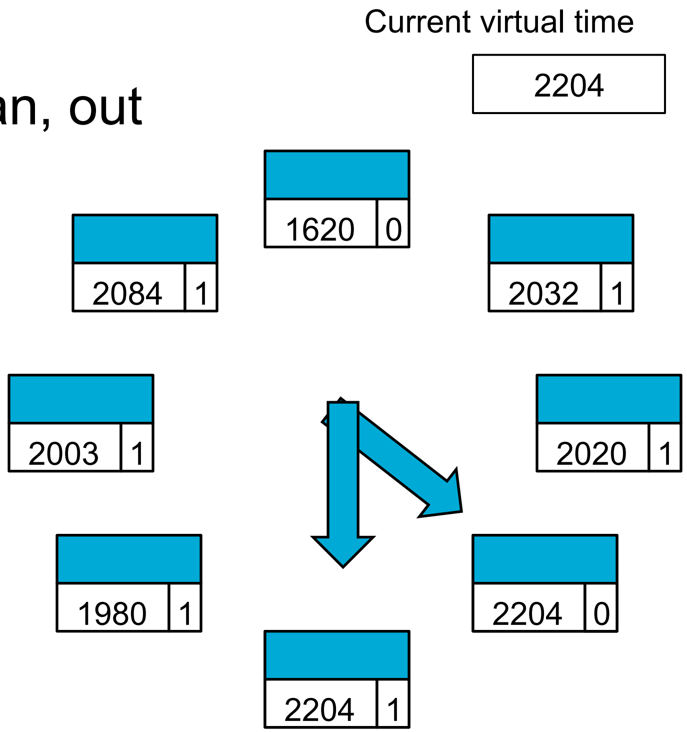


WSClock algorithm

- Problem with WS algorithm – Scans the whole table
- Instead, scan only what you need to find a victim
- Combine clock & working set
  - If *R = 1*, unset it
  - 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

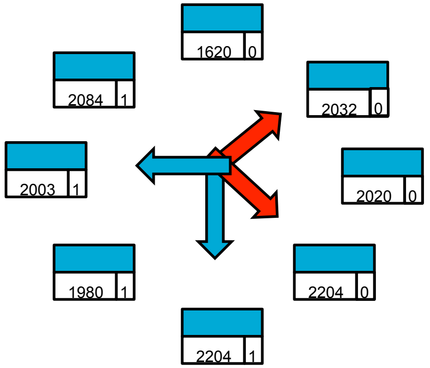
    There's none, pick any one



R = 0 & 2204 – 1213 > T

Cleaning policy

- To avoid having to write pages out when needed – paging daemon
  - Periodically inspects state of memory
  - Keep enough pages free
  - If we need the page before it's overwritten – reclaim it!
- Two hands for better performance (BSD)
  - First one clears R, second checks it
  - If hands are close, only heavily used pages have a chance
  - If back is just ahead of front hand (359°), original clock
  - Two key parameters, adjusted at runtime
    - Scanrate – rate at which hands move through the list
    - Handspread – gap between them



Design issues – global vs. local policy

- When you need a page frame, pick a victim from
  - Among your own resident pages – Local
  - Among all pages – Global
- Local algorithms
  - Basically every process gets a fixed % of memory
- Global algorithms
  - Dynamically allocate frames among processes
  - Better, especially if working set size changes at runtime
  - How many page frames per process?
    - Start with basic set & react to Page Fault Frequency (PFF)
- Most replacement algorithms can work both ways except for those based on working set

Why not working set based algorithms?

Load control

- Despite good designs, system may still thrash
  - Sum of working sets > physical memory
- Page Fault Frequency (PFF) indicates that
  - Some processes need more memory
  - but no process needs less
- Way out: Swapping
  - So yes, even with paging you still need swapping
  - Reduce number of processes competing for memory
  - ~ two-level scheduling – careful with which process to swap out (there's more than just paging to worry about!)

  What would you like of the remaining processes?

Backing store

- How do we manage swap area?
  - Allocate space to process when started
  - Keep offset to process swap area in PCB
  - Process can be brought entirely when started or as needed
- Some problems
  - Size – process can grow … split text/data/stack segments in swap area
  - Do not allocate anything … you may need extra memory to keep track of pages in swap!

Page fault handling

- Hardware traps to kernel
- General registers saved by assembler routine, OS called
- OS find which virtual page cause the fault
- OS checks address is valid, seeks page frame
- If selected frame is dirty, write it to disk *(CS)*
- Get new page *(CS)*, update page table
- Back up instruction where interrupted
- Schedule faulting process
- Routine load registers & other state and return to user space

Instruction backup

- With a page fault, the current instruction is stopped part way through … harder than you think!
  - Consider instruction: MOV.L #6(A1), 2(A0)

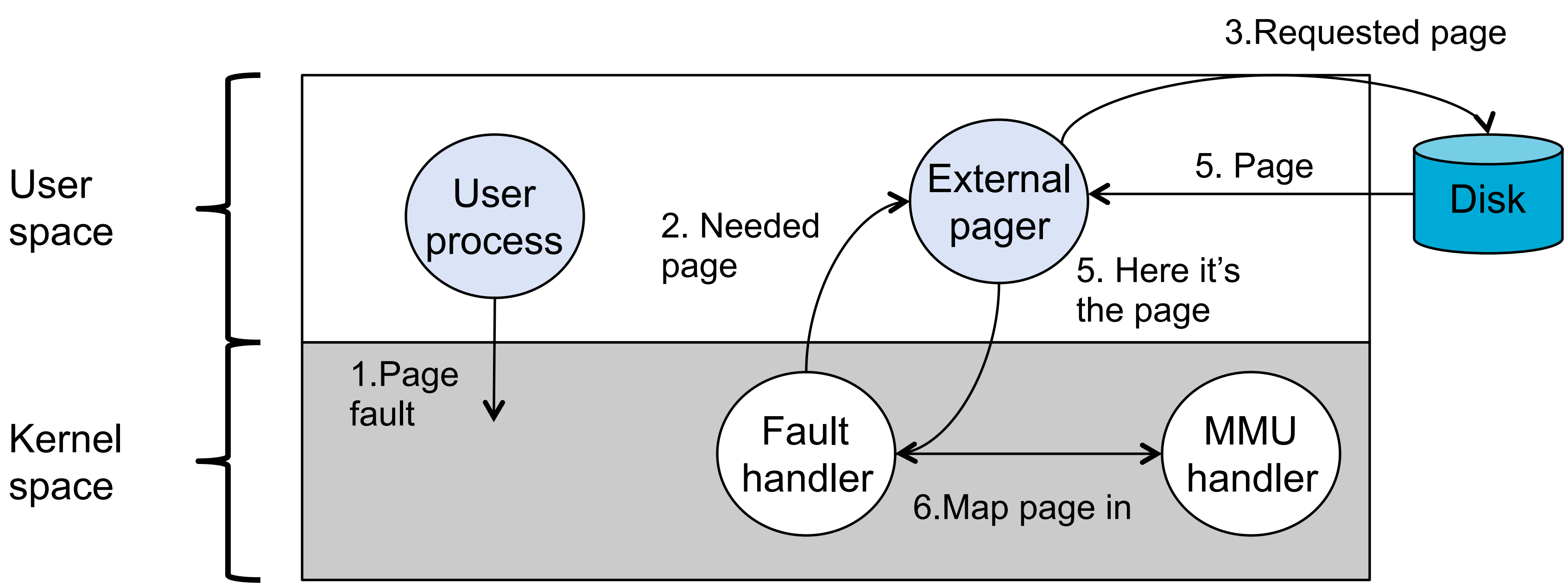
One instruction, three memory references (instruction word itself, two offsets for operands

1000	**MOVE**
1002	6
1004	2

  - Which one caused the page fault? What's the PC then?
  - Worse – autodecr/incr as a side-effect of execution?
- Some CPU design include hidden registers to store
  - Beginning of instruction
  - Indicate autodecr./autoincr. and amount

Separation of policy & mechanism

- How to structure the memory management system for easy separation? (based on Mach)
  1. Low-level MMU handler – machine dependent
  2. Page-fault handler in kernel – machine independent, most of paging mechanism
  3. External pager running in user space – policy is here



Separation of policy & mechanism

- *Where do you put the page replacement algorithm?*
  - In external pager? No access to R and M bits
    - Either pass it to the pager or
    - Fault handler informs external pager which page is victim
- Pros and cons
  - More modular, flexible
  - Overhead of crossing user-kernel boundary and msg exchange
  - As computers get faster and software more complex …

Next time

- Virtualize the CPU, virtualize memory, …
- Let’s virtualize the whole machine