

---

# CSC 369

Week 8: Page Replacement  
Reading: Text, Chapter 4.4-4.7

# Announcements

- A2 is out
- You need to make two minor changes to the main.c file in your a2-branch
  - New vm\_bootstrap returns physical memory size
  - swap\_bootstrap needs to be called with memsize
  - Why not call swap\_bootstrap in vm\_bootstrap?
    - Ordering constraints during boot
    - Swap needs to use vfs layer, disk device, both of which must be bootstrapped first
    - But vfs and devices need memory bootstrapped first

# Memory Management

Last 2 weeks on memory management:

- **Goals of memory management**
  - To provide a convenient abstraction for programming
  - To allocate scarce memory resources among competing processes to maximize performance with minimal overhead
- **Mechanisms**
  - Physical and virtual (logical) addressing (1)
  - Techniques: Partitioning, paging, segmentation (1)
  - Page table management, TLBs, VM tricks (2)
- **Policies**
  - Page fetch policy
  - Page placement policy
  - Page replacement algorithms (3) - today

# Locality

- All paging schemes depend on locality
  - Processes reference pages in localized patterns
- **Temporal locality**
  - Locations referenced recently likely to be referenced again
- **Spatial locality**
  - Locations near recently referenced locations are likely to be referenced soon
- Although the cost of paging is high, if it is infrequent enough it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging possible
  - All caching strategies depend on locality to be effective

# Policy Decisions

- Page tables, MMU, TLB, etc. are **mechanisms** that make virtual memory possible
- Today, we'll look at **policies** for virtual memory management:
  - Fetch Policy - when to fetch a page
    - Demand paging vs. Prepaging
  - Placement Policy - where to put the page
    - Are some physical pages preferable to others?
  - Replacement Policy - what page to evict to make room?
    - Lots and lots of possible algorithms!

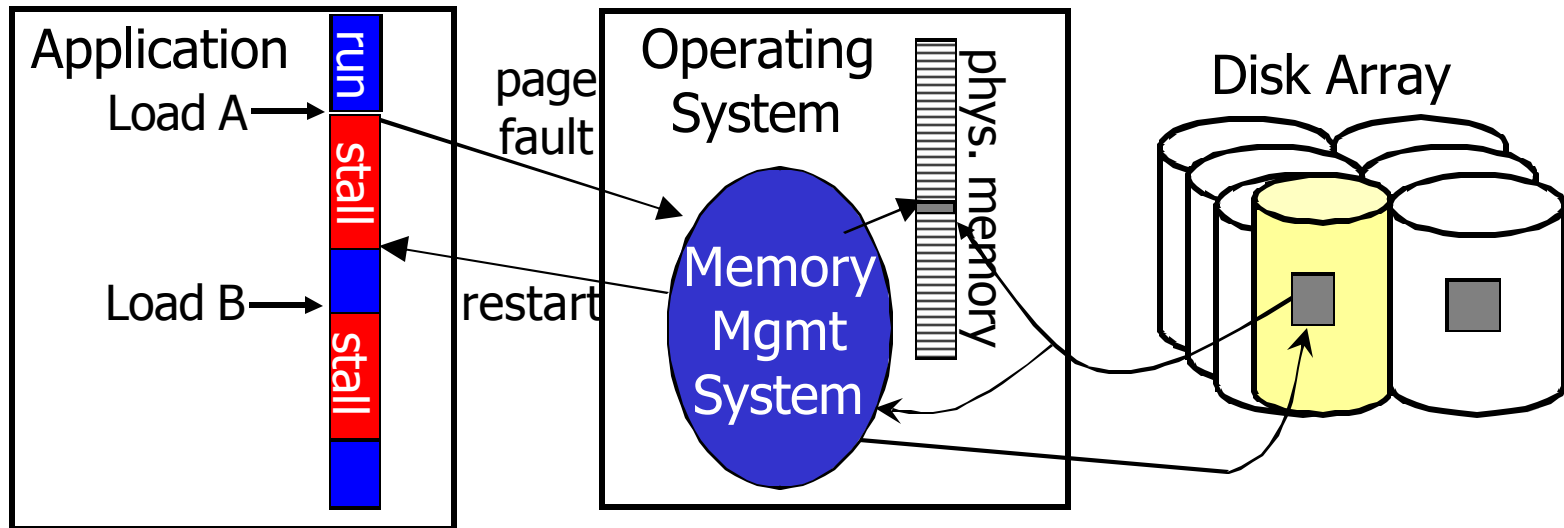
# Demand Paging (OS)

- Demand paging from the OS perspective:
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
  - References to evicted pages cause a TLB miss
    - PTE was invalid, causes fault
  - OS allocates a page frame, reads page from disk
  - When I/O completes, the OS fills in PTE, marks it valid, and restarts faulting process
- Dirty vs. clean pages
  - Actually, only dirty pages (modified) need to be written to disk
  - Clean pages do not - but you need to know where on disk to read them from again
    - In OS/161, each lpage has a location in swap area

# Demand Paging (Process)

- Demand paging is also used when a process first starts up
- When a process is created, it has:
  - A brand new page table with all valid bits off
  - No pages in memory
- When the process starts executing
  - Instructions fault on code and data pages
  - Faulting stops when all necessary code and data pages are in memory
  - Only code and data needed by a process needs to be loaded
  - This, of course, changes over time...
- How does OS161 do this?

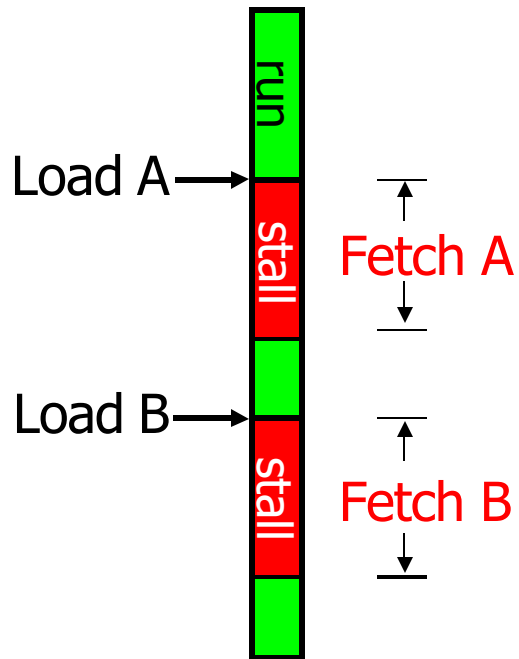
# Costs of Demand Paging



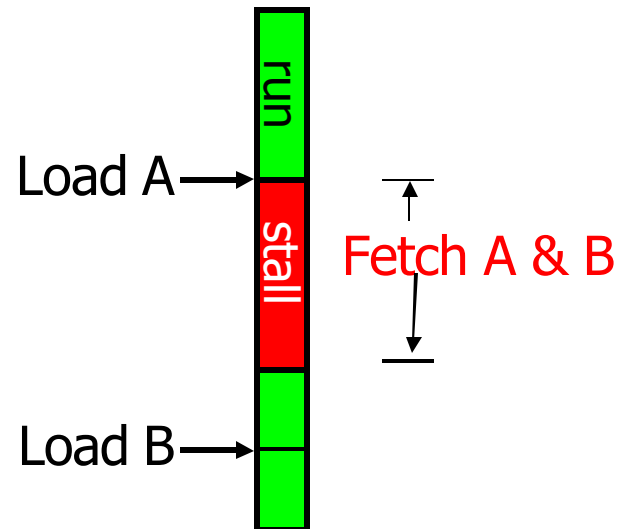
- Timing: Disk read is initiated **when the process needs the page**
- Request size: Process can only page fault on one page at a time, disk sees single page-sized read
- What alternative do we have?

# Prepaging (aka Prefetching)

Without Prepaging



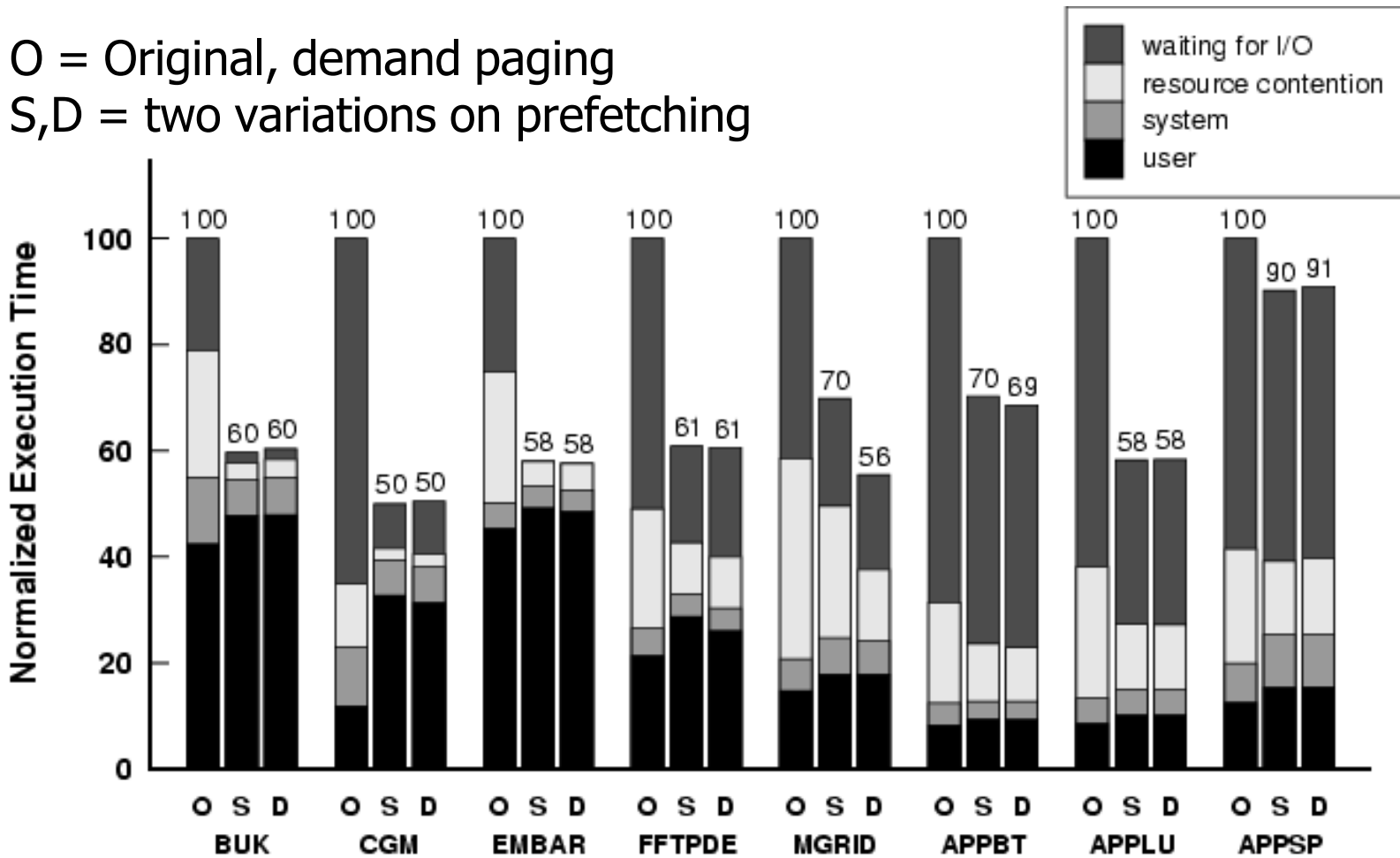
With Prepaging



- Predict future page use at time of current fault
  - On what should we base the prediction? What if it's wrong?

# Effect of Prefetching

O = Original, demand paging  
 S,D = two variations on prefetching



# Placement Policy

- In paging systems, memory management hardware can translate any virtual-to-physical mapping equally well
- Why would we prefer some mappings over others?
  - NUMA (non-uniform memory access) multiprocessors
    - any processor can access entire memory, but local memory is faster
  - Cache performance
    - Choose physical pages to minimize cache conflicts
- These are active research areas!

# Page Replacement Policy

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory
- There may be no free frames available for use
- When this happens, the OS must **replace** a page for each page faulted in
  - It must evict a page (called the **victim**) to free up a frame
  - If the victim has been modified since the last page-out, it must be written back to disk on eviction
    - Modify or dirty bit in PTE
    - LPF\_DIRTY flag in lp\_paddr field of lpage in OS161
- The **page replacement algorithm** determines how a victim is chosen

# Evicting the Best Page

- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
- Replacement algorithms are evaluated on a **reference string** by counting the number of page faults
- The best page to evict is the one never used again
  - Will never fault on it
- Never is a long time, so picking the page closest to "never" is the next best thing
  - Evicting the page that won't be used for the longest period of time minimizes the number of page faults
  - Proved by Belady, 1966

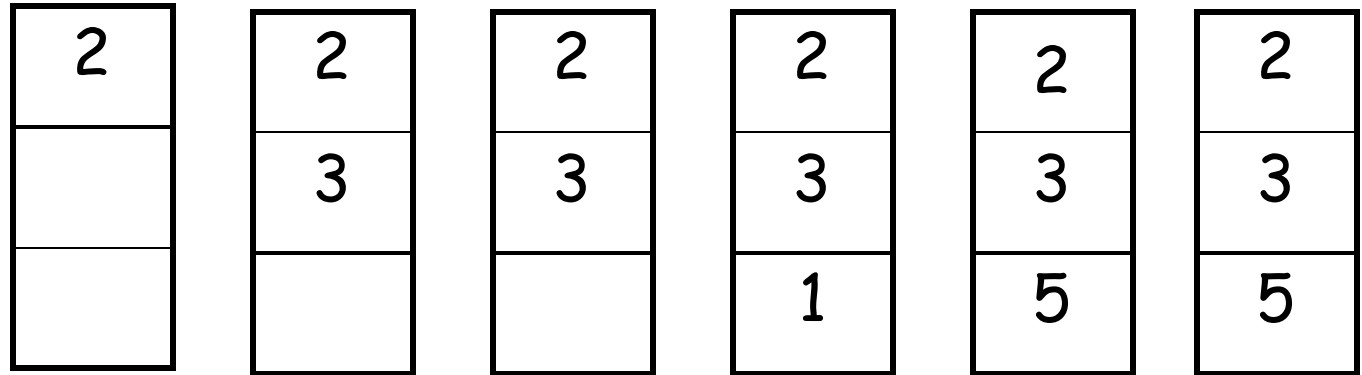
# Belady's Algorithm

- Belady's algorithm is known as the **optimal** page replacement algorithm because it has the lowest fault rate for any page reference stream (aka OPT or MIN)
  - Idea: Replace the page that will not be used for the longest period of time
  - Problem: Have to know the future perfectly
- Why is Belady's useful then? Use it as a yardstick
  - Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could use some work
    - Random replacement is often the lower bound

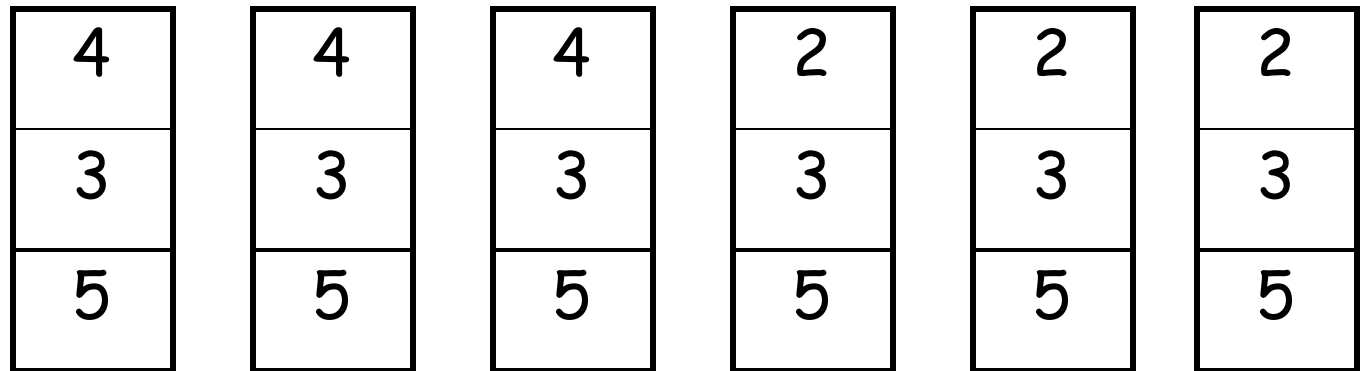
# Modelling Belady's Algorithm

- Page address list: 2,3,2,1,5,2,4,5,3,2,5,2

Cold misses:  
first access  
to a page  
(unavoidable)



Capacity  
misses:  
caused by  
replacement  
due to  
limited size  
of memory



# First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Then again, maybe it's not
  - We don't have any info to say one way or the other
- FIFO suffers from "Belady's Anomaly"
  - The fault rate might actually **increase** when the algorithm is given more memory (**very bad**)

# Modelling FIFO

- Page Address List: 0,1,2,3,0,1,4,0,1,2,3,4

	0	1	2	3	0	1	4	0	1	2	3	4	3 frames, 9 faults
Youngest	0	1	2	3	0	1	4	4	4	2	3	3	
		0	1	2	3	0	1	1	1	4	2	2	
Oldest			0	1	2	3	0	0	0	1	4	4	

Anomaly	0	1	2	3	0	1	4	0	1	2	3	4	4 frames, 10 faults
Youngest	0	1	2	3	3	3	4	0	1	2	3	4	
		0	1	2	2	2	3	4	0	1	2	3	
Oldest			0	1	1	1	2	3	4	0	1	2	

# Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - Idea: We can't predict the future, but we can make a guess based upon past experience
  - On replacement, evict the page that has not been used for the longest time in the **past** (Belady's: **future**)
  - When does LRU do well? When does LRU do poorly?
- Exact implementation is costly
  - So we need to approximate it

# Implementing Exact LRU

- Option 1:
  - Time stamp every reference
  - Evict page with oldest time stamp
  - Problems:
    - Need to make PTE large enough to hold meaningful time stamp (may double size of page tables, TLBs)
    - Need to examine every page on eviction to find one with oldest time stamp
- Option 2:
  - Keep pages in a stack. On reference, move the page to the top of the stack. On eviction, replace page at bottom.
  - Problems:
    - Need costly software operation to manipulate stack on EVERY memory reference!

# Modelling Exact LRU

- Page Address List: 0,1,2,3,0,1,4,0,1,2,3,4

3 Frames	0	1	2	3	0	1	4	0	1	2	3	4	10 faults
MRU page	0	1	2	3	0	1	4	0	1	2	3	4	
		0	1	2	3	0	1	4	0	1	2	3	
LRU page			0	1	2	3	0	1	4	0	1	2	

4 Frames	0	1	2	3	0	1	4	0	1	2	3	4	8 faults
MRU page	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	2	3	0	1	4	0	1	2	
LRU page				0	1	2	3	3	3	4	0	1	

# Approximating LRU

- Exact LRU is too costly to implement
- LRU approximations use the PTE **reference** bit
- Basic Idea:
  - Initially, all R bits are zero; as processes execute, bits are set to 1 for pages that are used
  - Periodically examine the R bits - we do not know order of use, but we know pages that were (or were not) used
- Additional-Reference-Bits Algorithm
  - Keep a counter for each page
  - At regular intervals, for every page do:
    - Shift R bit into high bit of counter register
    - Shift other bits to the right
    - Pages with “larger” counters were used more recently

# Second Chance Algorithm

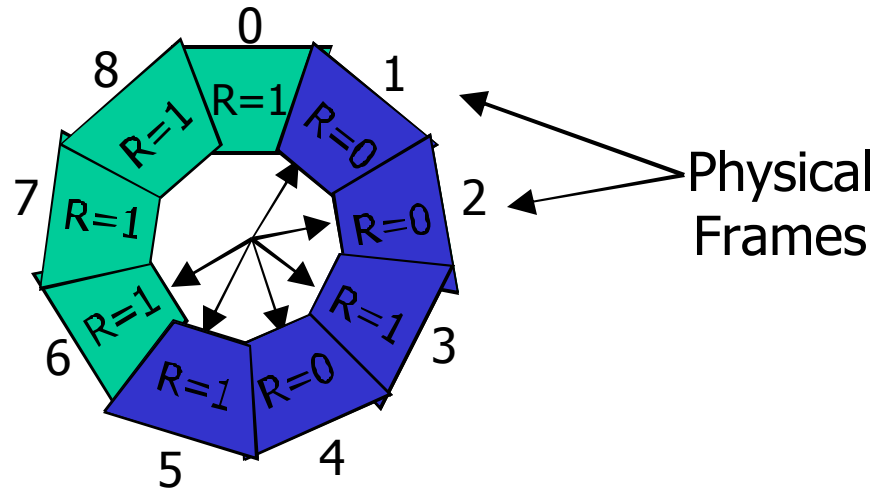
- FIFO, but inspect reference bit when page is selected
  - If ref bit is 0, replace the page
  - If ref bit is 1, clear ref bit, reset arrival time of page to current time
  - Pages that are used often enough to keep reference bits set will not be replaced
- Can combine with Modify bit to create 4 classes of pages
  - Called "Not Recently Used" in text

# Implementing Second Chance (clock)

## Replace page that is "old enough"

- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the ref bit (aka use bit) is off, it hasn't been used recently
    - What is the minimum "age" if ref bit is off?
  - If the ref bit is on, turn it off and go to next page
- Arm moves quickly when pages are needed
- Low overhead when plenty of memory
- If memory is large, "accuracy" of information degrades

# Modelling Clock



- 1st page fault:
  - Advance hand to frame 4, use frame 3
- 2nd page fault (assume none of these pages are referenced)
  - Advance hand to frame 6, use frame 5

# Counting-based Replacement

- Count number of uses of a page
- Least-Frequently-Used (LFU)
  - Replace the page used least often
  - Pages that are heavily used at one time tend to stick around even when not needed anymore
  - Newly allocated pages haven't had a chance to be used much
- Most-Frequently-Used (MFU)
  - Favours new pages
- Neither is common, both are poor approximations of OPT

# Page Buffering

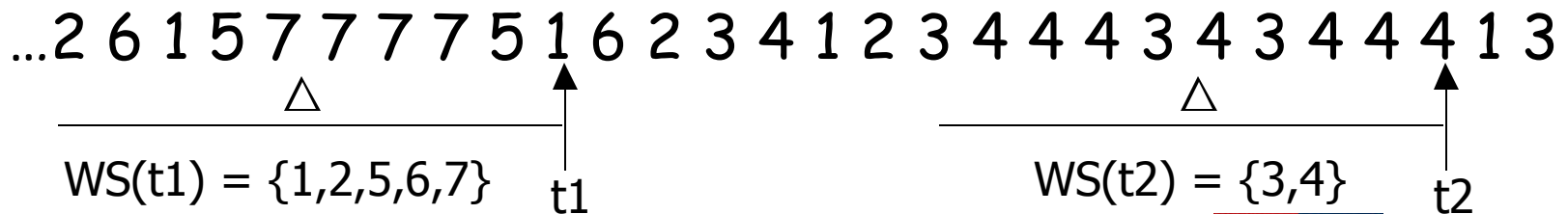
- Preceding discussion assumed the replacement algorithm is run and a victim page selected when a new page needs to be brought in
- Most of these algorithms are too costly to run on every page fault
  - Maintain a pool of free pages
  - Run replacement algorithm when pool becomes too small ("low water mark"), free enough pages to at once replenish pool ("high water mark")
  - On page fault, grab a frame from the free list
  - Frames on free list still hold previous contents, can be "rescued" if virtual page is referenced before reallocation

# Fixed vs. Variable Space

- In a multiprogramming system, we need a way to allocate memory to competing processes
- Problem: How to determine how much memory to give to each process?
  - Fixed space algorithms
    - Each process is given a limit of pages it can use
    - When it reaches the limit, it replaces from its own pages
    - Local replacement
      - Some processes may do well while others suffer
  - Variable space algorithms
    - Process' set of pages grows and shrinks dynamically
    - Global replacement - one process can ruin it for the rest
    - Local replacement - replacement and set size are separate

# Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60s
- Definition
  - $WS(t, \Delta) = \{\text{pages } P \text{ such that } P \text{ was referenced in the time interval } (t, t-\Delta)\}$
  - $t = \text{time}$ ,  $\Delta = \text{working set window (measured in page refs)}$
- A page is in the working set (WS) only if it was referenced in the last  $\Delta$  references



# Working Set Size

- The working set size is the number of pages in the working set
  - The number of pages referenced in the interval  $(t, t-\Delta)$
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter  $\Delta$  that determines a working set with few faults
  - Denning: Don't run a process unless working set is in memory

# Working Set Problems

- Problems
  - How do we determine  $\Delta$ ?
  - How do we know when the working set changes?
- Too hard to answer
  - So, working set is not used in practice as a page replacement algorithm
- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, "How much memory does Netscape need?", they are in effect asking for the size of Netscape's working set

# Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    - So that it faults less
    - But not always (FIFO, Belady's Anomaly)
  - If the fault rate is below a low threshold, take away memory
    - Should fault more
    - But not always
- Hard to use PFF to distinguish between changes in locality and changes in size of working set

# Thrashing

- Page replacement algorithms avoid **thrashing**
  - When more time is spent by the OS in paging data back and forth from disk than executing user programs
  - No time spent doing useful work (making progress)
  - In this situation, the system is **overcommitted**
    - No idea which pages should be in memory to reduce faults
    - Could just be that there isn't enough physical memory for all of the processes in the system
    - Ex: Running Window95 with 4 MB of memory...
  - Possible solutions
    - Swapping - write out all pages of a process and suspend it
    - Buy more memory
  - Bad idea: CPU utilization is low, so we can increase the degree of multiprogramming, right?

# Windows XP Paging Policy

- Local page replacement
  - Per-process FIFO
  - Pages are stolen from processes using more than their minimum working set
  - Processes start with a default of 50 pages
  - XP monitors page fault rate and adjusts working-set size accordingly
  - On page fault, cluster of pages around the missing page are brought into memory

# Linux Paging

- Global replacement, like most Unix
- Modified second-chance clock algorithm
  - Pages age with each pass of the clock hand
  - Pages that are not used for a long time will eventually have a value of zero
- Continually under development... we'll look at some details of the current version later

# Summary

- Page replacement algorithms
  - Belady's - optimal replacement (minimum # of faults)
  - FIFO - replace page loaded furthest in past
  - LRU - replace page referenced furthest in past
    - Approximate using PTE reference bit
  - LRU Clock - replace page that is "old enough"
  - Working Set - keep the set of pages in memory that has minimal fault rate (the "working set")
  - Page Fault Frequency - grow/shrink page set as a function of fault rate
- Multiprogramming
  - Should a process replace its own page, or that of another?