CS162
Operating Systems and Systems Programming
Lecture 11

Page Allocation and Replacement

October 9, 2013
Anthony D. Joseph and John Canny
http://inst.eecs.berkeley.edu/~cs162

Post Project 1 Class Format

- Mini quizzes after each topic
  - Not graded!
  - Simple True/False
  - Immediate feedback for you (and me)
- Separate from pop quizzes

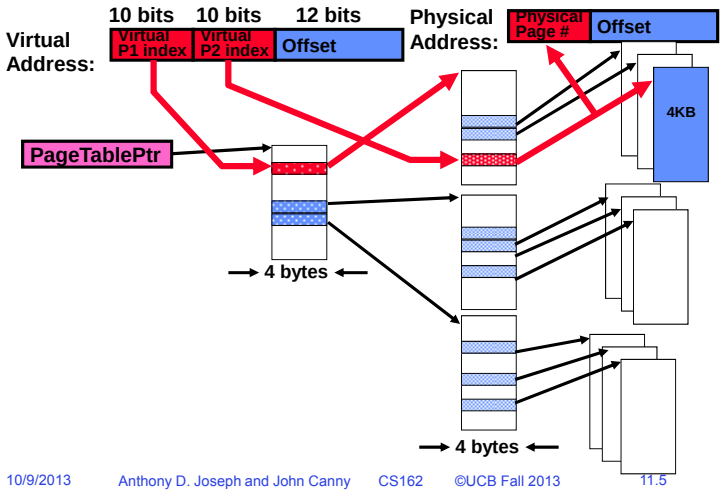
Quiz 11.1: Caches & TLBs

- Q1: True _ False _ Associative caches have fewer compulsory misses than direct mapped caches
- Q2: True _ False _ Two-way set associative caches can cache two addresses with same cache index
- Q3: True _ False _ With write-through caches, a read miss can result in a write
- Q5: True _ False _ A TLB caches translations to virtual addresses

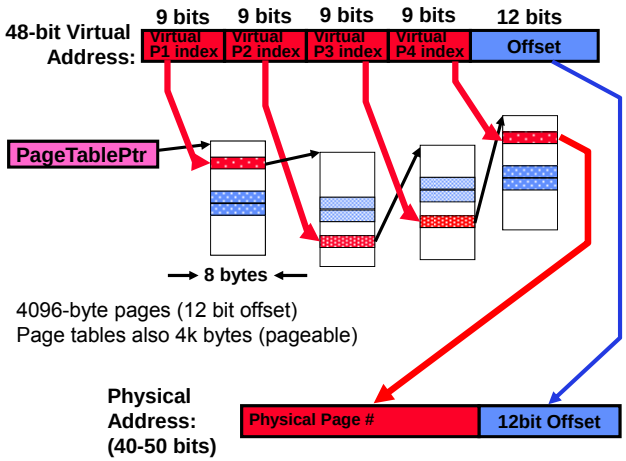
Quiz 11.1: Caches & TLBs

- Q1: True _ False x Associative caches have fewer compulsory misses than direct mapped caches
- Q2: True x False _ Two-way set associative caches can cache two addresses with same cache index
- Q3: True _ False x With write-through caches, a read miss can result in a write
- Q5: True _ False x A TLB caches translations to virtual addresses

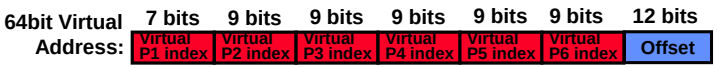
Review: Two-level page table



X86_64: Four-level page table!



IA64: 64bit addresses: Six-level page table?!?

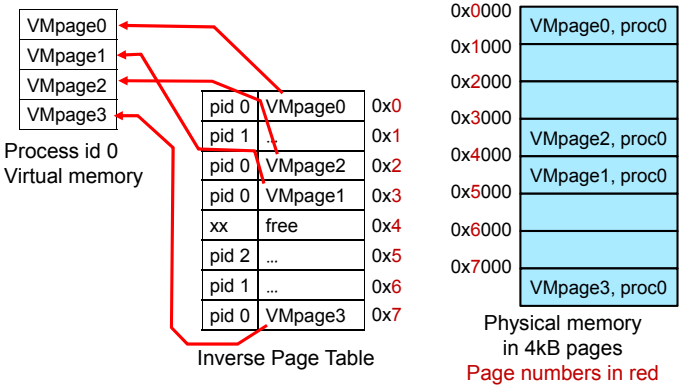


No!

Too slow
Too many almost-empty tables

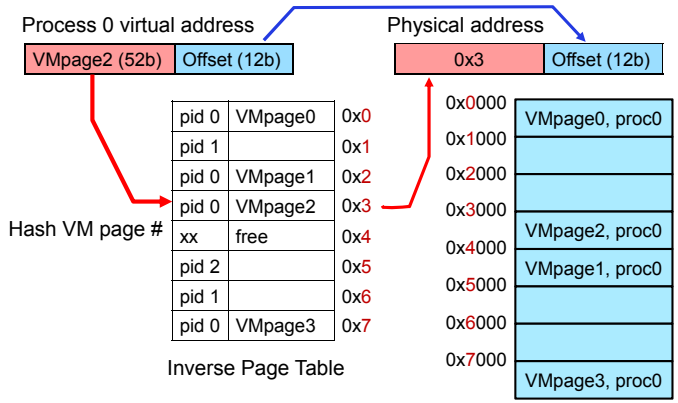
IA64: Inverse Page Table (IPT)

Idea: index the page table by physical pages instead of VM



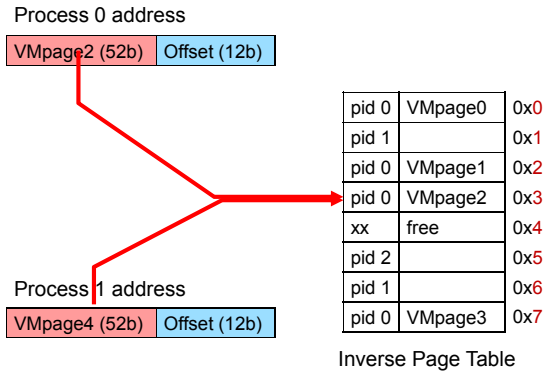
IPT address translation

- Need an associative map from VM page to IPT address: **Use a hash map**.



IPT address translation

Note: can't share memory: only one hashed entry will match.



IA64: Inverse Page Table (IPT)

Pros:

- Page table size naturally linked to physical memory size.
- Only two memory accesses (most of the time).
- Shouldn't need to page out the page table.
- Hash function can be very fast if implemented in hardware.

Cons:

- Can't (easily) share pages.
- Have to manage collisions, e.g. by chaining, which adds memory accesses.

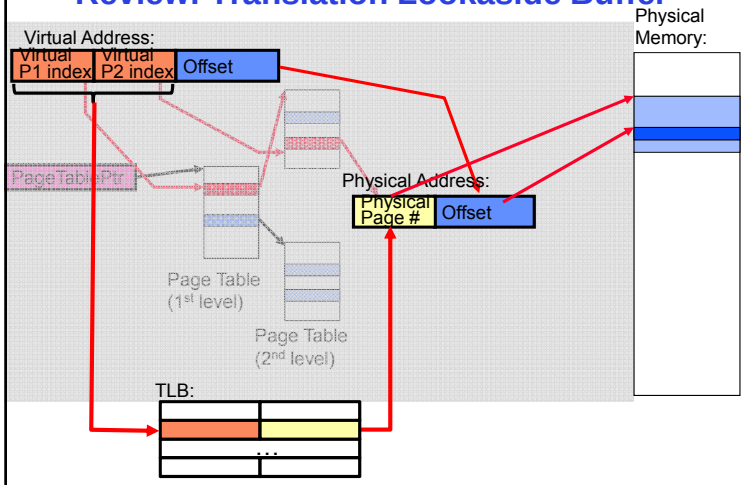
Quiz 11.2: Address Translation

- Q1: True _ False _ Paging does not suffer from external fragmentation
- Q2: True _ False _ The segment offset can be larger than the segment size
- Q3: True _ False _ Paging: to compute the physical address, add physical page # and offset
- Q4: True _ False _ Uni-programming doesn't provide address protection
- Q5: True _ False _ Virtual address space is always larger than physical address space
- Q6: True _ False _ Inverted page tables keeps fewer entries than multi-level page tables

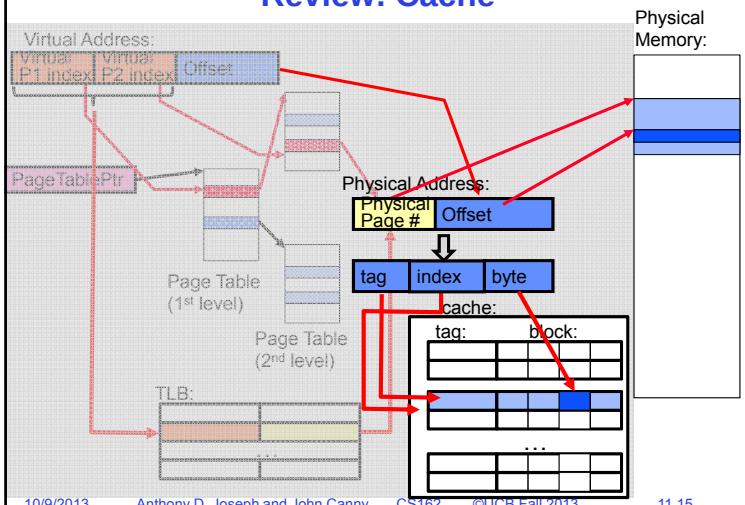
Quiz 11.2: Address Translation

- Q1: True X False _ Paging does not suffer from external fragmentation
- Q2: True _ False X The segment offset can be larger than the segment size
- Q3: True _ False X Paging: to compute the physical address, add physical page # and offset
- Q4: True X False _ Uni-programming doesn't provide address protection
- Q5: True _ False X Virtual address space is always larger than physical address space
- Q6: True _ False X Inverted page tables keeps fewer entries than multi-level page tables

Review: Translation Lookaside Buffer



Review: Cache



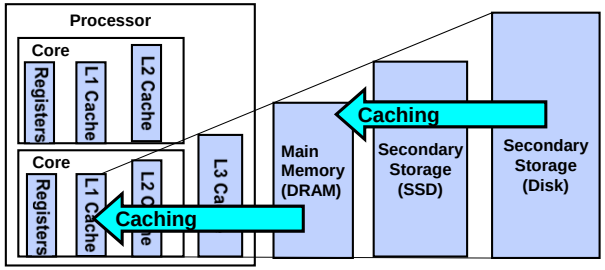
Goals for Today

- Page Replacement Policies
  - FIFO
  - LRU
  - Clock Algorithm

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as cache for disk/SSD



Demand Paging is Caching

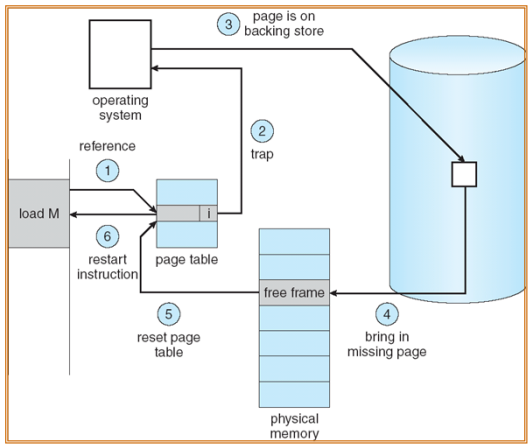
- Since Demand Paging is Caching, we must ask:

Question	Choice
What is the block size?	
What is the organization of this cache (i.e., direct-mapped, set-associative, fully-associative)?	
How do we find a page in the cache?	
What is page replacement policy? (i.e., LRU, Random, …)	
What happens on a miss?	
What happens on a write? (i.e., write-through, write-back)	

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Steps in Handling a Page Fault



Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

    EAT = (1 – p) x 200ns + p x 8 ms
    = (1 – p) x 200ns + p x 8,000,000ns
    = 200ns + p x 7,999,800ns
- If one access out of 1,000 causes a page fault, then EAT = 8.2 µs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - EAT < 200ns x 1.1 $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400,000 !

What Factors Lead to Misses?

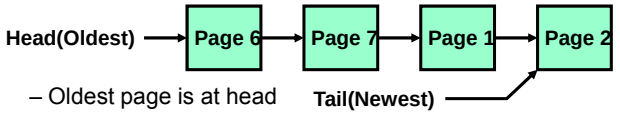
- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - » Prefetching: loading them into memory before needed
    - » Need to predict future somehow! More later.
- Capacity Misses:
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - » One option: Increase amount of DRAM (not quick fix!)
    - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses:
  - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Unpredictable

Replacement Policies (Con't)

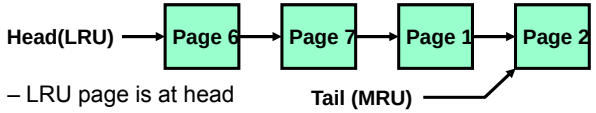
- FIFO:
  - Replace page that has been in for the longest time.
  - Be "fair" to pages and give them equal time.
  - Bad idea because page use is not even. We want to give more time to heavily used pages.
- How to implement FIFO? It's a queue (can use a linked list)



  - Oldest page is at head
  - When a page is brought in, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

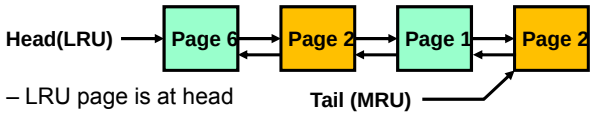
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list?



  - LRU page is at head
  - When a page is used for the first time, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

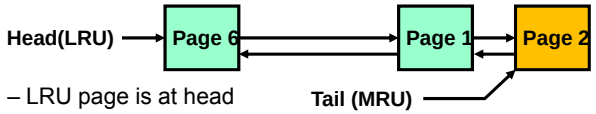
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity
- Problems with this scheme for paging?
  - Updates are happening on page **use**, not just swapping
  - List structure requires extra pointers c.f. FIFO, more updates
- In practice, people approximate LRU (more later)

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

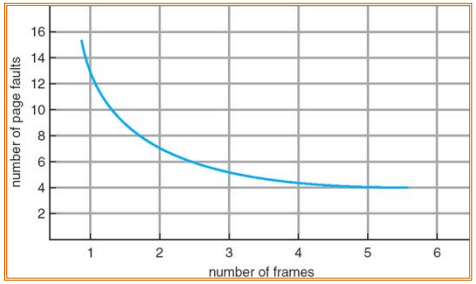
- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- MIN Does much better:

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: Belady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Administrivia

- Project #1:
  - Design doc (`submit proj1-final-design`) and group evals (Google Docs form) due today at 11:59PM
  - » Group evals are anonymous to your group
- Midterm #1 is Monday Oct 21 5:30-7pm in 145 Dwinelle (A-L) and 2060 Valley LSB (M-Z)
  - – Closed book, double-sided ***handwritten*** page of notes, no calculators, smartphones, Google glass etc.
  - – Covers lectures #1-13 (Disks/SSDs, Filesystems), readings, handouts, and projects 1 and 2
  - – Review session **390 Hearst Mining, Fri October 18, 5-7 PM**
- Class feedback is always welcome!

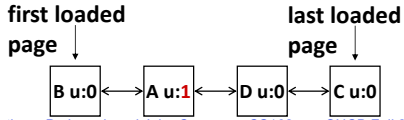
5min Break

Implementing LRU & Second Chance

- Perfect:
  - – Timestamp page on each reference
  - – Keep list of pages ordered by time of reference
  - – Too expensive to implement in reality for many reasons
- Second Chance Algorithm:
  - – Approximate LRU
    - » Replace an old page, not the oldest page
  - – FIFO with "use" bit
- Details
  - – A "use" bit per physical page
    - » set when page accessed
  - – On page fault check page at head of queue
    - » If use bit=1 → clear bit, and move page to tail (give the page second chance!)
    - » If use bit=0 → replace page
  - – Moving pages to tail still complex

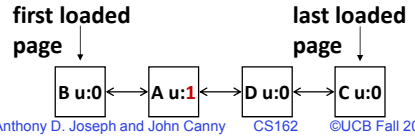
Second Chance Illustration

- Max page table size 4
  - – Page B arrives
  - – Page A arrives
  - – Access page A
  - – Page D arrives
  - – Page C arrives



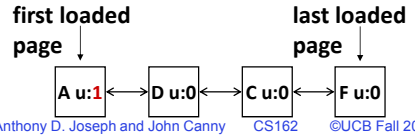
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



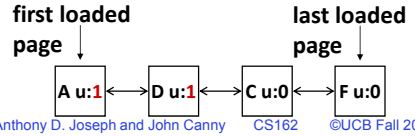
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



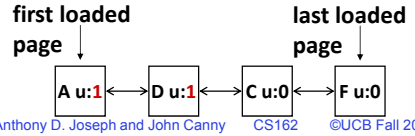
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D



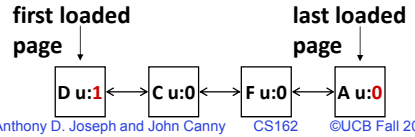
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



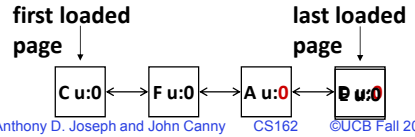
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Clock Algorithm

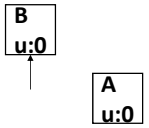
- Clock Algorithm: more efficient implementation of second chance algorithm
  - Arrange physical pages in circle with single clock hand
- Details:
  - On page fault:
    - » Check use bit: 1→used recently; clear and leave it alone
      0→selected candidate for replacement
    - » Advance clock hand (not real time)
  - Will always find a page or loop forever?

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives

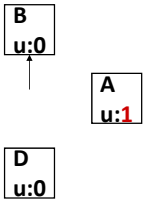
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A



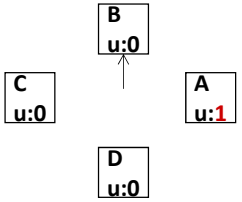
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives



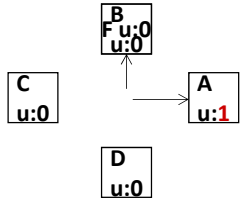
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



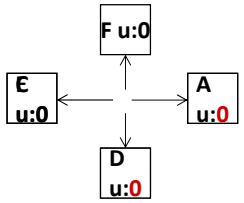
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D



Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Clock Algorithm: Discussion

- What if hand moving slowly?
  - Good sign or bad sign?
    - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set

Nth Chance version of Clock Algorithm

- Nth chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Quiz 11.3: Demand Paging

- Q1: True _ False _ Demand paging incurs conflict misses
- Q2: True _ False _ LRU can never achieve higher hit rate than MIN
- Q3: True _ False _ The LRU miss rate may increase as the cache size increases
- Q4: True _ False _ The Clock algorithm is a simpler implementation of the Second Chance algorithm
- Q5: Assume a cache with 100 pages. The number of pages that the Second Chance algorithm may need to check before finding a page to evict is at most ___

Quiz 11.3: Demand Paging

- Q1: True _ False x Demand paging incurs conflict misses
- Q2: True x False _ LRU can never achieve higher hit rate than MIN
- Q3: True _ False x The LRU miss rate may increase as the cache size increases
- Q4: True x False _ The Clock algorithm is a simpler implementation of the Second Chance algorithm
- Q5: Assume a cache with 100 pages. The number of pages that the Second Chance algorithm may need to check before finding a page to evict is at most **101**

Summary (1/2)

- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss ⇒ find free page, get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Replacement policies
  - FIFO: Place pages on queue, replace page at head
    - » Fair but can eject in-use pages, suffers from Belady's anomaly
  - MIN: Replace page that will be used farthest in future
    - » Benchmark for comparisons, can't implement in practice
  - LRU: Replace page used farthest in past
    - » For efficiency, use approximation

Summary (2/2)

- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace