Virtual Memory

- A generalization of what demand paging allows
- A form of memory where the system provides a useful abstraction
  - A very large quantity of memory
  - For each process
  - All directly accessible via normal addressing
  - At a speed approaching that of actual RAM
- The state of the art in modern memory abstractions

The Basic Concept

- Give each process an address space of immense size
  - Perhaps as big as your hardware's word size allows
- Allow processes to request segments within that space
- Use dynamic paging and swapping to support the abstraction
- The key issue is how to create the abstraction when you don't have that much real memory

The Key VM Technology: Replacement Algorithms

- The goal is to have each page already in memory when a process accesses it
- We can't know ahead of time what pages will be accessed
- We rely on locality of access
  - In particular, to determine what pages to move out of memory and onto disk
- If we make wise choices, the pages we need in memory will still be there

The Basics of Page Replacement

- We keep some set of all possible pages in memory
  - Perhaps not all belonging to the current process
- Under some circumstances, we need to replace one of them with another page that's on disk
  - E.g., when we have a page fault
- Paging hardware and MMU translation allows us to choose any page for ejection to disk
- Which one of them should go?

The Optimal Replacement Algorithm

- Replace the page that will be next referenced furthest in the future
- Why is this the right page?
  - It delays the next page fault as long as possible
  - Fewer page faults per unit time = lower overhead
- A slight problem:
  - We would need an oracle to know which page this algorithm calls for
  - And we don’t have one

Do We Require Optimal Algorithms?

- Not absolutely
- What's the consequence of the algorithm being wrong?
  - We take an extra page fault that we shouldn't have
  - Which is a performance penalty, not a program correctness penalty
  - Often an acceptable tradeoff
- The more often we're right, the fewer page faults we take

Approximating the Optimal

- Rely on locality of reference
- Note which pages have recently been used
  - Perhaps with extra bits in the page tables
  - Updated when the page is accessed
- Use this data to predict future behavior
- If locality of reference holds, the pages we accessed recently will be accessed again soon

Candidate Replacement Algorithms

- Random, FIFO
  - These are dogs, forget ‘em
- Least Frequently Used
  - Sounds better, but it really isn’t
- Least Recently Used
  - Assert that near future will be like the recent past
  - If we haven’t used a page recently, we probably won’t use it soon
  - The computer science equivalent to the “*unseen hand*”

Naïve LRU

- Each time a page is accessed, record the time
- When you need to eject a page, look at all timestamps for pages in memory
- Choose the one with the oldest timestamp
- Will require us to store timestamps somewhere
- And to search all timestamps every time we need to eject a page

True LRU Page Replacement

Reference stream

a	b	c	d	a	b	d	e	f	a	b	c	d	a	e	d

Page table using true LRU

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frame 0	a				!				f				d			!
frame 1		b				!				a				!		
frame 2			c					e				c				
frame 3				d			!				b				e	

Loads 4

Replacements 7

Maintaining Information for LRU

- Can we keep it in the MMU?
  - MMU notes the time whenever a page is referenced
  - MMU translation must be blindingly fast
    - Getting/storing time on every fetch would be very expensive
  - At best they will maintain a *read* and a *written* bit per page
- Can we maintain this information in software?
  - Mark all pages invalid, even if they are in memory
  - Take a fault first time each page is referenced, note the time
  - Then mark this page valid for the rest of the time slice
  - Causing page faults to reduce the number of page faults???
- We need a <u>cheap</u> software surrogate for LRU
  - No extra page faults
  - Can't scan entire list each time, since it's big

Clock Algorithms

- A surrogate for LRU
- Organize all pages in a circular list
- MMU sets a reference bit for the page on access
- Scan whenever we need another page
  - For each page, ask MMU if page has been referenced
  - If so, reset the reference bit in the MMU & skip this page
  - If not, consider this page to be the least recently used
  - Next search starts from this position, not head of list
- Use position in the scan as a surrogate for age
- No extra page faults, usually scan only a few pages

Clock Algorithm Page Replacement

Reference Stream

a	b	c	d	a	b	d	e	f	a	b	c	d	a	e	d

LRU clock

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frame 0	a				!	!	!	!	f				d			!
frame 1		b				!	!	!		a				!	!	
frame 2			c					e			b				e	
frame 3				d			!	!	!			c				
clock pos	0	1	2	3	0	0	0	0/2	3/0	1	2	3	0	1	1/2	3

loads 4, replacements 7

True LRU

frame 0	a				a											d
frame 1		b												a		
frame 2			c					e				c				
frame 3				d			d				b				e	

Loads 4

Replacements 7

Comparing True LRU To Clock Algorithm

- Same number of loads and replacements
  - But didn't replace the same pages
- What, if anything, does that mean?
- Both are just approximations to the optimal
- If LRU clock's decisions are 98% as good as true LRU
  - And can be done for 1% of the cost (in hardware and cycles)
  - It is a bargain!

Page Replacement and Multiprogramming

- We don't want to clear out all the page frames on each context switch
- How do we deal with sharing page frames?
- Possible choices:
  - Single global pool
  - Fixed allocation of page frames per process
  - Working set-based page frame allocations

Single Global Page Frame Pool

- Treat the entire set of page frames as a shared resource
- Approximate LRU for the entire set
- Replace whichever process' page is LRU
- Probably a mistake
  - Bad interaction with round-robin scheduling
  - The guy who was last in the scheduling queue will find all his pages swapped out
  - And not because he isn't using them
  - When he gets in, lots of page faults

Per-Process Page Frame Pools

- Set aside some number of page frames for each running process
  - Use an LRU approximation separately for each
- How many page frames per process?
- Fixed number of pages per process is bad
  - Different processes exhibit different locality
    - Which pages are needed changes over time
    - Number of pages needed changes over time
  - Much like different natural scheduling intervals
- We need a dynamic customized allocation