LRU, Optimal, and Caching

44-550: Operating Systems

Least Recently Used

- Replace the page in memory that has not been used for the longest amount of time

Needed Page:	0	1	2	1	4	2	3	7
Page 1:								
Page 2:								
Page 3:								
Hit/Fault								

LRU Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*							
Page 2:								
Page 3:								
Hit/Fault	F							

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*						
Page 2:		1						
Page 3:								
Hit/Fault	F	F						

LRU Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*					
Page 2:		1	1					
Page 3:			2					
Hit/Fault	F	F	F					

LRU Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*	0*				
Page 2:		1	1	1				
Page 3:			2	2				
Hit/Fault	F	F	F	H				

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*	0*	4			
Page 2:		1	1	1	1			
Page 3:			2	2	2*			
Hit/Fault	F	F	F	H	F			

LRU Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*	0*	4	4		
Page 2:		1	1	1	1	1*		
Page 3:			2	2	2*	2		
Hit/Fault	F	F	F	H	F	H		

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*	0*	4	4	4*	
Page 2:		1	1	1	1	1*	3	
Page 3:			2	2	2*	2	2	
Hit/Fault	F	F	F	H	F	H	F	

LRU Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0*	0*	0*	0*	4	4	4*	7
Page 2:		1	1	1	1	1*	3	3
Page 3:			2	2	2*	2	2	2*
Hit/Fault	F	F	F	H	F	H	F	F

$p = \frac{6}{8} = .75$

Note that LRU frequently (but not always) performs better than FIFO. There can and will be examples where FIFO will outperform LRU (Free Lunch Theorem)

Optimal

- Replace the page in memory that will not be used for the longest amount of time
- Requires the entire stream of needed pages
  - Unless you have an amazing branch predictor, this will never work all of the time

Needed Page:	0	1	2	1	4	2	3	7
Page 1:								
Page 2:								
Page 3:								
Hit/Fault								

Optimal Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0(∞)							
Page 2:								
Page 3:								
Hit/Fault	F							

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0(∞)						
Page 2:		1(3)						
Page 3:								
Hit/Fault	F	F						

Optimal Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0(∞)					
Page 2:		1	1(3)					
Page 3:			2(5)					
Hit/Fault	F	F	F					

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0	0(∞)				
Page 2:		1	1	1(∞)				
Page 3:			2	2(5)				
Hit/Fault	F	F	F	H				

Optimal Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0	0	$4(\infty)$			
Page 2:		1	1	1	$1(\infty)$			
Page 3:			2	2	2(5)			
Hit/Fault	F	F	F	H	F			

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0	0	4	$4(\infty)$		
Page 2:		1	1	1	1	$1(\infty)$		
Page 3:			2	2	2	$2(\infty)$		
Hit/Fault	F	F	F	H	F	H		

Optimal Example

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0	0	4	4	3(∞)	
Page 2:		1	1	1	1	1	1(∞)	
Page 3:			2	2	2	2	2(∞)	
Hit/Fault	F	F	F	H	F	H	F	

Needed Page:	0	1	2	1	4	2	3	7
Page 1:	0	0	0	0	4	4	3	3(∞)
Page 2:		1	1	1	1	1	1	7(∞)
Page 3:			2	2	2	2	2	2(∞)
Hit/Fault	F	F	F	H	F	H	F	F

$p = \frac{6}{8} = .75$

CPU Caching

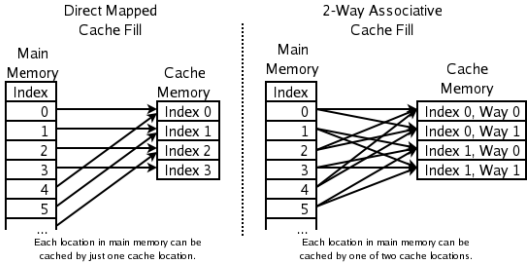
- Cache is on chip memory used by the CPU to decrease the average time to access memory
  - Super fast super expensive memory
- Modern CPUs have 3 kinds of cache:
  - Data cache
  - Instruction Cache
  - Translation Look aside Buffer (TLB): Speeds up virtual to physical memory access for both instructions and data
- Cache hit: processor immediately reads data from or writes to the cache
- Cache miss: cache allocates a new entry reading in data from memory
  - Think page faults, just a lot less painful
- Cache lines: representation of a block of memory in cache

Cache Entry

- Has 3 parts:
  - Tag: identifier for the cache line and part of the memory address
  - Data block: data for the cache line
  - flag bits: valid and dirty bits
    - valid: indicates whether the block has been loaded with avlid data
    - dirty: indicates whether the block has been changed since it was read from memory. Dirty means we need to save it to main memory.
- In general: If you hit the cache, the data lookup is free!

Associative Cache

- Decides where in cache a memory entry will go
- Fully associative: replacement policy may choose any entry
- Set association: memory entries are mapped to specific locations in cache
- Important in programming for High Performance systems: interleaving data in loops may allow for fewer cache misses



Translation lookaside Buffer (TLB)

- Memory management hardware for cache
- Improves speed to translate virtual addresses
- Includes basic memory information such as references to pages, frames, and variables.

Memory Access

- Uniform Memory Access
  - Memory location accessed is independent of the processor
- Non-uniform memory access (NUMA)
  - Systems where memory access times vary wildly (based on memory location relative to the processor)
  - Memory location accessed depends on the processor
    - Every processor has it's own memory
    - Much like an exteremly tightly coupled cluster on one machine
  - x86 Processors have supported NUMA since 2007

Cache Coherency

- Consistency of data on the cache shared by local CPUs.
- Three levels:
  - Write operations appear to occur instantly
  - Processor see the same chages of values for each operand
  - Different processor might see an op and assume different values (non-coherence)
- Three implementations
  - Snooping: individaul caches monitor other caches
  - Snarfing: cache controller watches addresses and memory; updates its data copies once other data is updated in memory
  - Concurrency protocol: how do we implement coherence?
    - Hint: It's really hard
- ccnuma
  - allows for consistent memory image across sockets
  - IPC used to keep a consistent memory image when more than one cache stores the same memory location

Thrashing

- Virtual memory is constantly switching data between cache and memory, and between memory an disk
- Constant paging (memory to disk) can be resolved by:
  - MORE RAM!
  - Rewriting programs to be smarter
  - Reducing memory usage
  - Increasing spatial locality
- Spatial locality is part of the programmer's responsibilities

Example

- Download `examples/cache/cache_thrasher.c` from the course website
- Inspect the file and note the difference between the two functions being timed.
- Modify, compile, and run (cygwin should work for this) the program for the following values of `ARRAY_DIM`:
  - 10
  - 100
  - 1000
  - 5000
  - 10000
- Record the times and speedups that are reported in a table and answer the following questions:
  - What do you notice as the size of the 2D array grows?
  - Based on what you know about pointers and memory allocation and caching, what do you think is happening? Why is one faster than the other?