Page Fault

The act of accessing a page that is not in physical memory is referred to as a page fault

- Present bit in page table entry: indicates if a page of a process resides in memory or not
- When translating VA to PA, MMU reads present bit
- If page present in memory, directly accessed
- If page not in memory, MMU raises a trap to the OS –page fault

Page Fault Handling

- Page fault traps OS and moves CPU to kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address (say, in page table)
- OS context switches to another process
  - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process scheduled again, OS restarts the instruction that caused page fault

Summary: what happens on memory access

- CPU issues load to a VA for code or data
  - –Checks CPU cache first
  - –Goes to main memory in case of cache miss
- MMU looks up TLB for VA
  - –If TLB hit, obtains PA, fetches memory location and returns to CPU (via CPU caches)
  - –If TLB miss, MMU accesses memory, walks page table, and obtains page table entry
    - If present bit set in PTE, accesses memory
    - If not present but valid, raises page fault. OS handles page fault and restarts the CPU load instruction
    - If invalid page access, trap to OS for illegal access

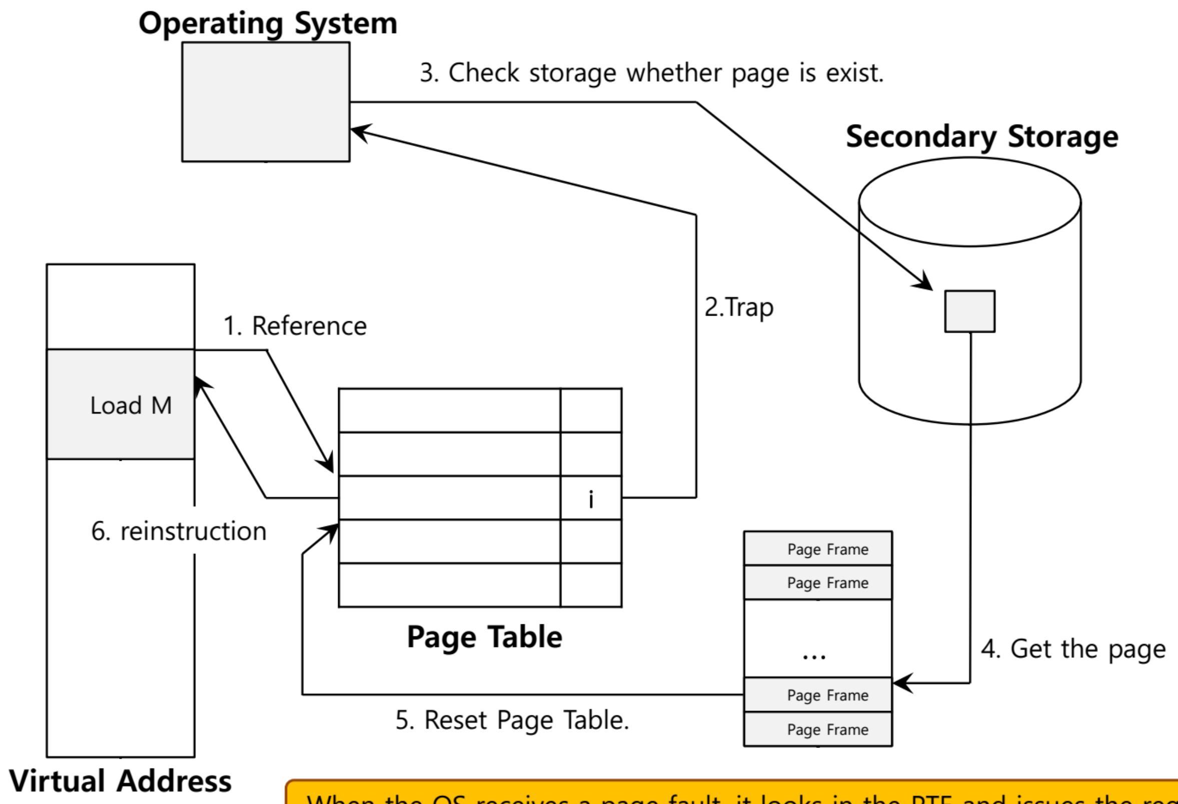
Subtle point

While the I/O is in flight, the process will be in the blocked state.

Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another process

Another way a multiprogrammed system can make the most effective use of its hardware

Page Fault Control Flow



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page fault control flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

```
PFN = FindFreePhysicalPage()
if (PFN == -1)                // no free page found
    PFN = EvictPage()         // run replacement algorithm
DiskRead(PTE.DiskAddr, PFN)   // sleep (waiting for I/O)
PTE.present = True            // update page table with present
PTE.PFN     = PFN             // bit and translation (PFN)
RetryInstruction()            // retry instruction
```

More Complications in a Page fault

- When servicing page fault, what if OS finds that there is no free page to swap in the faulting page?
- OS must swap out an existing page (if it has been modified, i.e., dirty) and then swap in the faulting page–too much work!
- OS may proactively swap out pages to keep list of free pages handy
- Which pages to swap out? Decided by page replacement policy.

- The OS must find a physical frame for the soon-be-faulted-in page to reside within.
- If there is no such page, waiting for the replacement algorithm to run and kick some pages out of memory.

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page
  - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.

Swap Daemon, Page Daemon

To keep a small amount of memory free, most operating systems have some kind of high watermark(HW) and low watermark(LW) to help decide when to start evicting pages from memory.

How this works :

When the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs.

The thread evicts pages until there are HW pages available.

The background thread, sometimes called the swap daemon or page daemon, then goes to sleep, after it has freed some memory for running processes and the OS to use.

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Minimize the number of times that we have to fetch a page from disk

or

Maximize the number of times a page that is accessed is found in memory

Effective Access Time (EAT) = $H \times \text{Access}_{C} + (1\text{-}H) \times \text{Access}_{MM}$

EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.

Virtual Memory Policies

Goal: Minimize number of page faults

Page faults require milliseconds to handle (reading from disk)

Implication: Plenty of time for OS to make good decision

OS has two decisions

Page selection

When should a page (or pages) on disk be brought into memory?

Page replacement

Which resident page (or pages) in memory should be thrown out to disk?

Page Selection

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems?

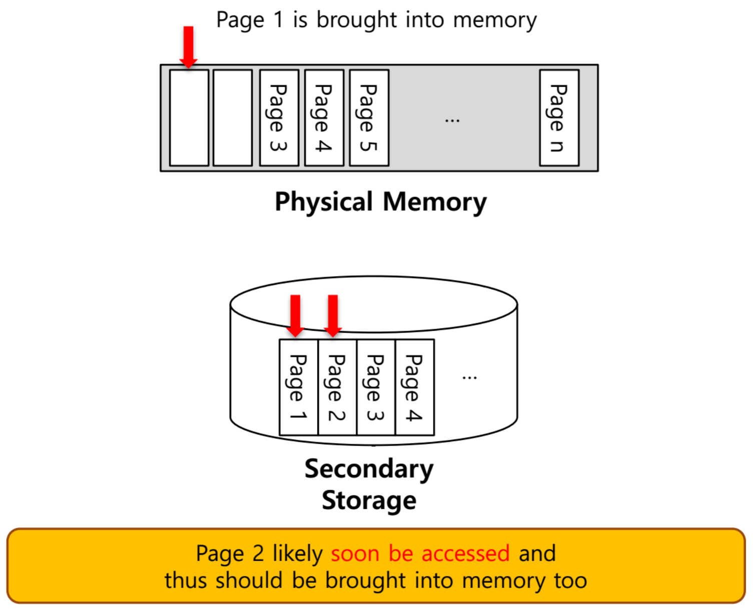
Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don’t need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix

When should a page be brought from disk into memory?

Prefecthing/Prepaging

- The OS guess that a page is about to be used, and thus bring it in ahead of time.



Page 1 is brought into memory
Page 3
Page 4
Page 5
…
Page n
Physical Memory
Page 1
Page 2
Page 3
Page 4
…
Secondary Storage
Page 2 likely soon be accessed and thus should be brought into memory too

Page Replacement

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

FIFO: Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
- Disadvantage: Some pages may always be needed

Page replacement policies

- **Optimal:** replace page not needed for longest time in future (not practical!)
- **FIFO:** replace page that was brought into memory earliest (may be a popular page!)
- **LRU/LFU:** replace the page that was least recently (or frequently) used in the past

LRU: Least-recently-used: Replace page not used for longest time in past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed
  - Does not handle all workloads well

Tracing the Optimal Policy

Assume a program accesses the following stream of virtual pages:

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Assuming a cache that fits three pages

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

Tracing the FIFO Policy

Assume a program accesses the following stream of virtual pages:

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Assuming a cache that fits three pages

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

Even though page 0 had been accessed a number of times, FIFO still kicks it out.

Belady's Anomaly

Interesting reference stream that behaved a little unexpectedly

The memory-reference stream: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**.

The replacement policy they were studying was FIFO.

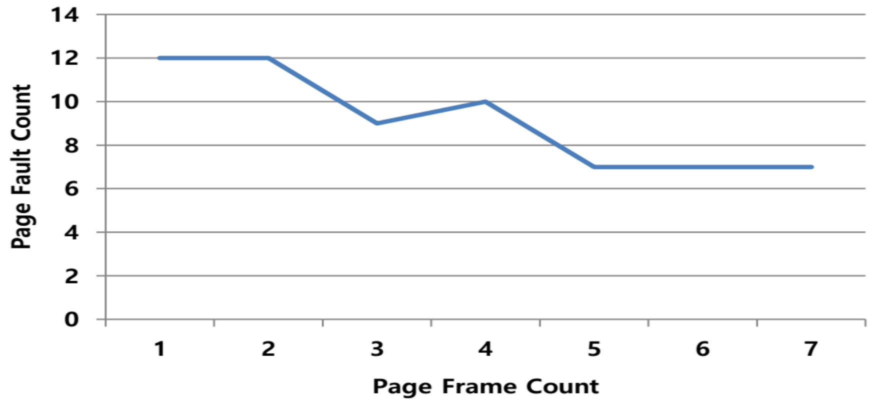
The interesting part: how the cache hit rate changed when moving from a cache size of 3 to 4 pages.

Belady's Anomaly

- We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO, it gets worse.

Reference Row

1 2 3 4 1 2 5 1 2 3 4 5

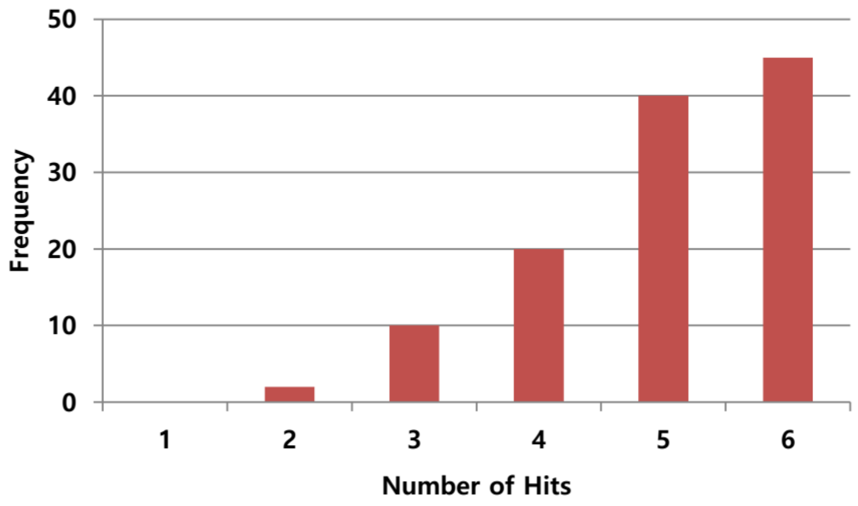


Random Policy

- Picks a random page to replace under memory pressure.
  - It doesn't really try to be too intelligent in picking which blocks to evict.
  - Random does depends entirely upon how lucky Random gets in its choice.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

Need for better ?

FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again.

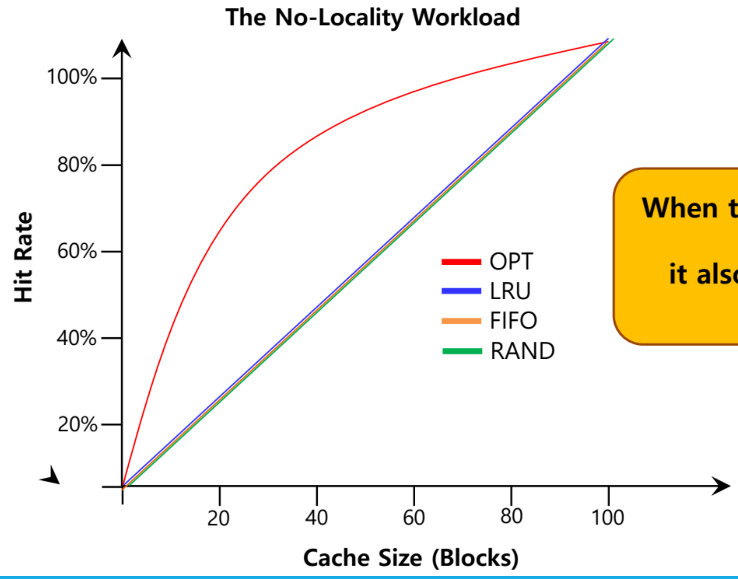
FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon bepaged back in.

Thus, FIFO, Random, and similar policies are not likely to approach optimal;

Something smarter is needed

The No-locality workload

- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

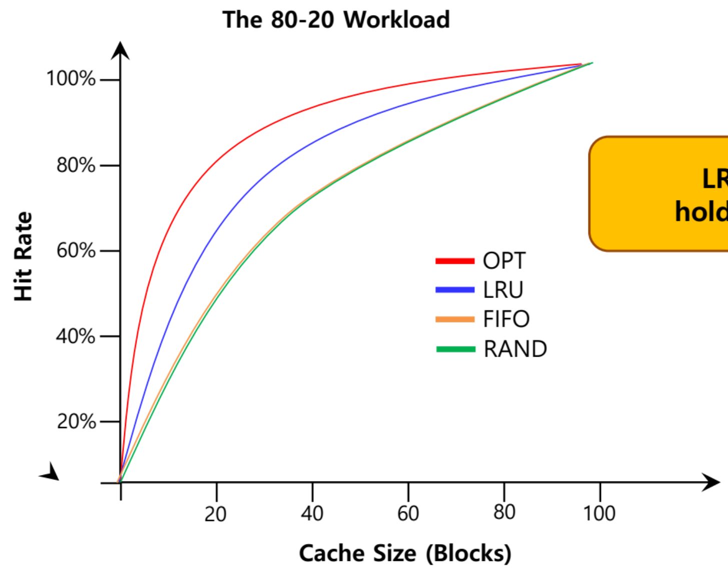
Overall, 10,000 pages are accessed.

Cache size varied from very small (1 page) to enough to hold all the unique pages (100 page)

It doesn't matter much which realistic policy you are using; LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache.

80-20 Workload

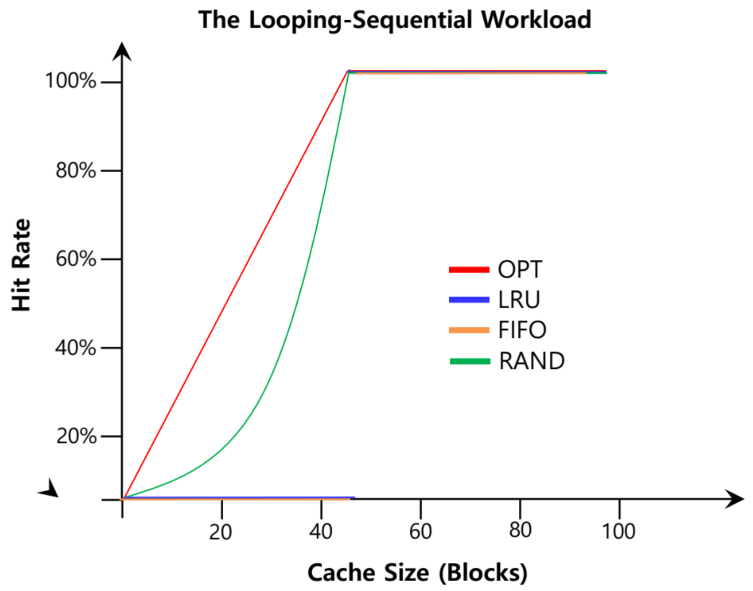
- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.



LRU is more likely to hold onto the hot pages.

Looping Sequential

- Refer to 50 pages in sequence.
  - Starting at 0, then 1, … up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



FIFO & LRU under a looping-sequential workload, kick out older pages; unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache

Using History

- Lean on the past and use <u>**history**</u>.
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replcaed as it clearly has some value	LFU

Principle of locality

An observation about programs and their behavior.

What this principle says --

Programs tend to access certain code sequences (e.g., in a loop) and datastructures (e.g., an array accessed by the loop) quite frequently;

Spatial locality -- states that if a page P is accessed,it is likely the pages around it (say P−1 or P+ 1) will also likely be accessed.

Temporal locality -- states that pages that have been accessed in the near past are likely to be accessed again in the near future.

Use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

Using History: LRU

- Replaces the least-recently-used page.

Reference Row

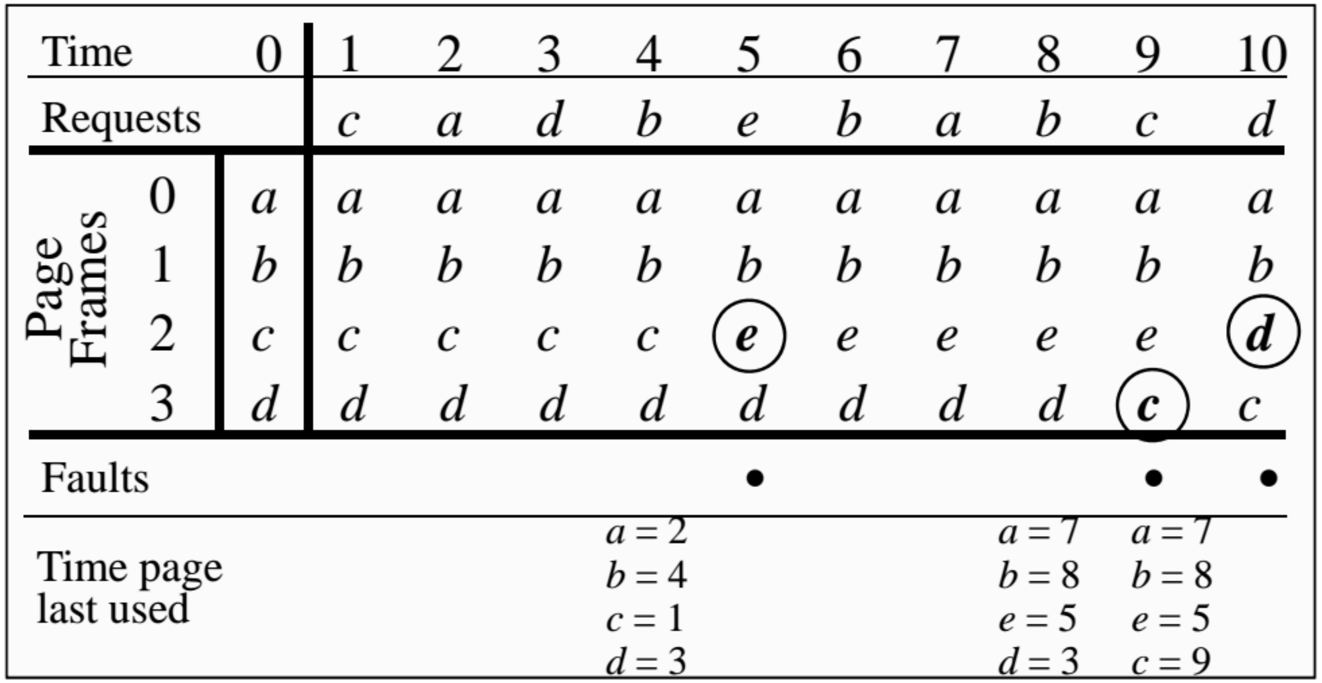
0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- Replace the page that hasn't been referenced for the longest time



Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							●				●	●
Time page last used						$a = 2$ $b = 4$ $c = 1$ $d = 3$				$a = 7$ $b = 8$ $e = 5$ $d = 3$	$a = 7$ $b = 8$ $e = 5$ $c = 9$	

How is LRU implemented ?

- OS is not involved in every memory access
  - –how does it know which page is LRU? [To keep track of which pages have been least-and most-recently used, the system has to do some accounting work on every memory reference]
- Hardware help and some approximations
- MMU sets a bit in PTE (“accessed” bit) when a page is accessed
- OS periodically looks at this bit to estimate pages that are active and inactive
- To replace, OS tries to find a page that does not have access bit set –May also look for page with dirty bit not set (to avoid swapping out to disk)

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

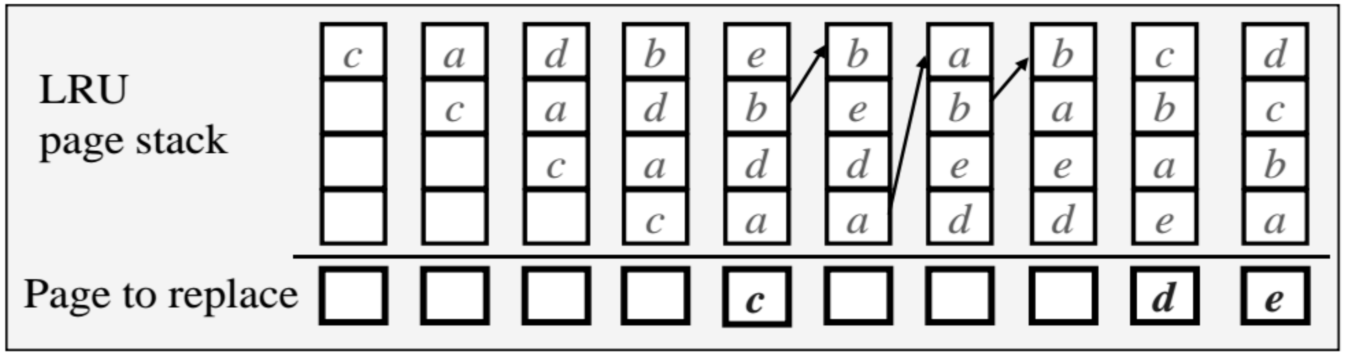
- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, Perfect LRU is seldom implemented -- LRU is an approximation anyway, so approximate more Goal: Find an old page, but not necessarily the very oldest

Least Recently Used Page Replacement
Implementation

- Maintain a “stack” of recently used pages

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•



LRU page stack	*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
		c	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*
			c	*a*	*d*	*d*	*e*	*e*	*a*	*b*
				c	*a*	*a*	*d*	*d*	*e*	*a*
Page to replace					***c***				***d***	***e***

Historical Algorithms

- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on ***every memory reference.***
  - Add a little bit of hardware support.

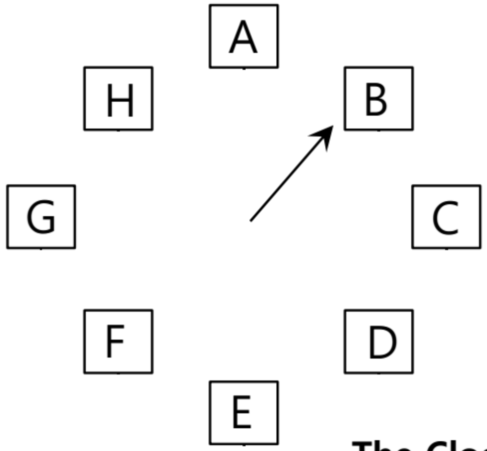
Approximating LRU

- Require some hardware support, in the form of a **<u>use bit</u>**
  - Whenever a page is referenced, the use bit is set by hardware to 1.
  - Hardware never clears the bit, though; that is the responsibility of the OS

- Clock Algorithm
  - All pages of the system arranges in a circular list.
  - A clock hand points to some particular page to begin with.

Clock Algorithm

- The algorithm continues until it finds a use bit that is set to 0.



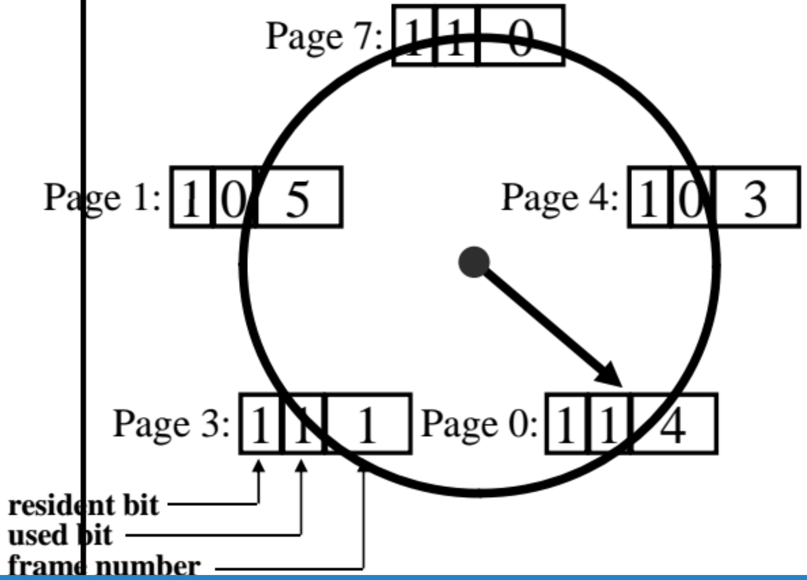
Use bit	Meaning
0	Evict the page
1	Clear **<u>Use bit</u>** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Approximate LRU Page Replacement
The *Clock* algorithm

- Maintain a circular list of pages resident in memory
  - Use a *clock* (or *used/referenced*) bit to track how often a page is accessed
  - The bit is set whenever a page is referenced
- Clock hand sweeps over pages looking for one with *used* bit = 0
  - Replace pages that haven't been referenced for one complete revolution of the clock



```
func Clock_Replacement
begin
  while (victim page not found) do
    if (used bit for current page = 0) then
      replace current page
    else
      reset used bit
    end if
    advance clock pointer
  end while
end Clock_Replacement
```

Clock algorithm

Hardware

Keep use (or reference) bit for each page frame

When page is referenced: set use bit

Operating System

Page replacement: Look for page with use bit cleared (has not been referenced for awhile)

Implementation:

Keep pointer to last examined page frame

Traverse pages in circular buffer

Clear use bits as search

Stop when find page with already cleared use bit, replace this page

Clock Page Replacement

Example

	Time	0	1	2	3	4	5	6	7	8	9	10
	Requests		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*						
	1	*b*	*b*	*b*	*b*	*b*						
	2	*c*	*c*	*c*	*c*	*c*						
	3	*d*	*d*	*d*	*d*	*d*						
	Faults											

Page table entries for resident pages:

1	*a*
1	*b*
1	*c*
1	*d*

Clock Page Replacement
Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	***e***	*e*	*e*	*e*	*e*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	***a***	*a*	*a*	*a*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•		•		•	•

Page table entries for resident pages:	0	1	2	3	4	5	6	7	8	9	10
	1 *a*					1 *e*	1 *e*	1 *e*	1 *e*	1 *e*	1 *d*
	1 *b*					0 *b*	1 *b*	0 *b*	1 *b*	1 *b*	0 *b*
	1 *c*					0 *c*	0 *c*	1 *a*	1 *a*	1 *a*	0 *a*
	1 *d*					0 *d*	0 *d*	0 *d*	0 *d*	1 *c*	0 *c*