빈 페이지 리스트에서 비어 있는 페이지를 찾아서 폴트된 [illegible] 페이지에 할당을 해주면 된다.

Beyond Physical Memory: Policies

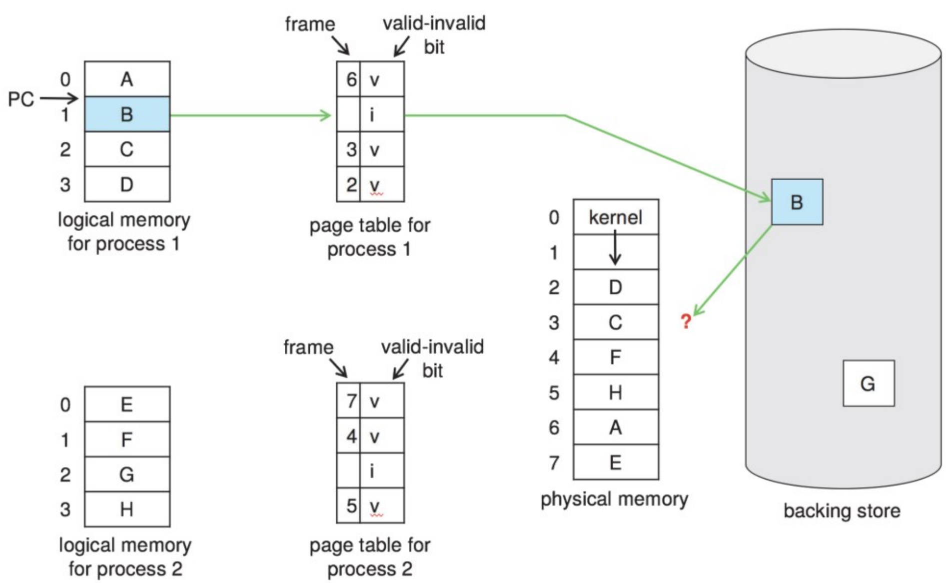
Prof. Yongtae Kim

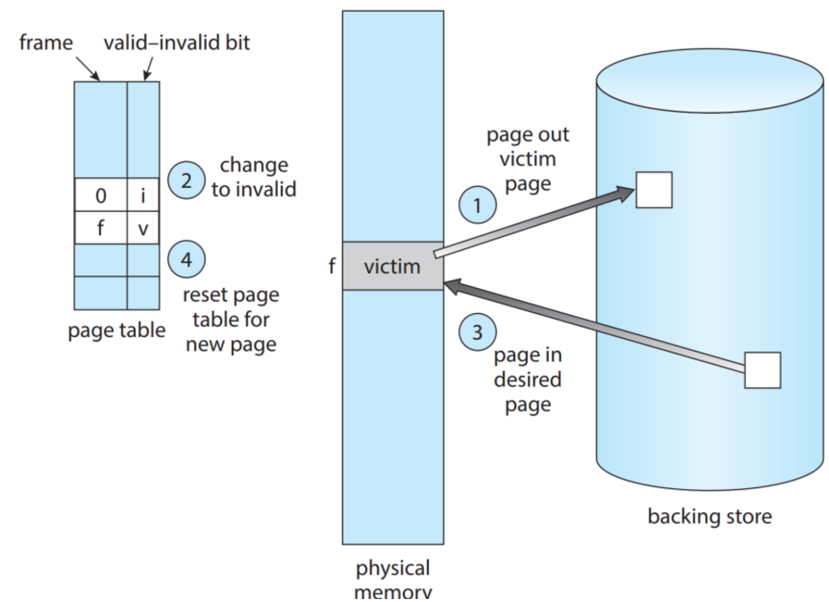
Computer Science and Engineering
Kyungpook National University

Page Replacement: Concept

메모리는 일종의 cache 라고 생각할 수 있다.

- **The memory pressure forces the OS to start paging out pages to make room for actively-used pages**
  - Deciding which page(s) to evict is encapsulated within OS's replacement policy
- **Since main memory holds a subset of all pages in system, it can be viewed as a cache for virtual memory pages in the system**
  - Then, our primary goal in picking a replacement policy for this cache is to minimize the number of cache misses (i.e. maximize the cache hit rate)





Cache Management

→ cache miss를 최소화 하는 게 목표.

AMAT = $T_m + (P_{Miss} \times T_D)$

- **Considering "average memory access time" (AMAT) for a program**

$$AMAT = T_M + (P_{Miss} \cdot T_D)$$

중요한 요소

Parameter	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Miss}	The probability of not finding the data in the cache (0 ~ 1.0)

memory access

- **AMAT examples**

disk access

$100ns + (0.1 \times 10ms)$ $10 \times 10^6 ns$

$= 10^2 ns + 10^6 ns$

$10^{-4} ms + 1 ms = 1.0001 ms.$

  - T_M = 100 ns, T_D = 10 ms = 10,000,000 ns
  - P_{Miss} = 0.1 (90% hit): AMAT = 100 ns + (0.1 × 10 ms) = 1.0001 ms ≈ 1 ms ≒ 1 ms
  - P_{Miss} = 0.001 (99.9% hit): AMAT = 100 ns + (0.001 × 10 ms) = 10.1 μs
  - Hit rate 90%→99.9% (9.99%p difference) results in AMAT 100× faster

- **The cost of disk access is so high in modern systems**
  - Therefore, even a tiny miss rate will quickly dominate the overall AMAT of running programs

The Optimal Replacement Policy (MIN)

Belady (optimal 정책)

- **The optimal replacement policy leads to the fewest number of misses overall**
  - The optimal policy is to replace the page that will be accessed furthest in the future, resulting in the fewest-possible cache misses

가장 나중에 사용될 page는 교체하는 방법.

but 미래에 어떤 page가 사용될 지는 알 수 x

- **Example with virtual page access of 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1 in a row and a cache with capacity of 3 pages**

그래서 다른 optimal policy와의 비교대상 고정.

  - Not surprisingly, the first three accesses are misses due to empty cache and such a miss is sometimes called cold-start miss (or compulsory miss)
  - When we reach another miss (page 3), the page 2 will be accessed furthest in the future → OS evicts page 2
  - We can't build the optimal policy for general purpose OS and use it as reference for comparison as future is usually not known

반드시 발생하는 miss

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

알 수 없음 (안다 가정)

$$hit\ rate = \frac{hits}{hits + misses} = \frac{6}{6+5} = 54.5\%$$

optimal 정책이면 미래의 가장 나중에 사용될 page를 교체.

A Simple Policy: FIFO

→ 가장 먼저 사용된 page를 miss가 발생했을 때 교체하겠다는 의미.

- **FIFO (first-in, first-out) replacement was used early systems**
  - Pages were simply placed in a queue when they enter the system
  - When a replacement occurs, the page on the front of the queue is evicted
  - FIFO has one great strength: quite simple to implement
  - Cons: hit rate may not increase when the cache gets larger (Belady's Anomaly)

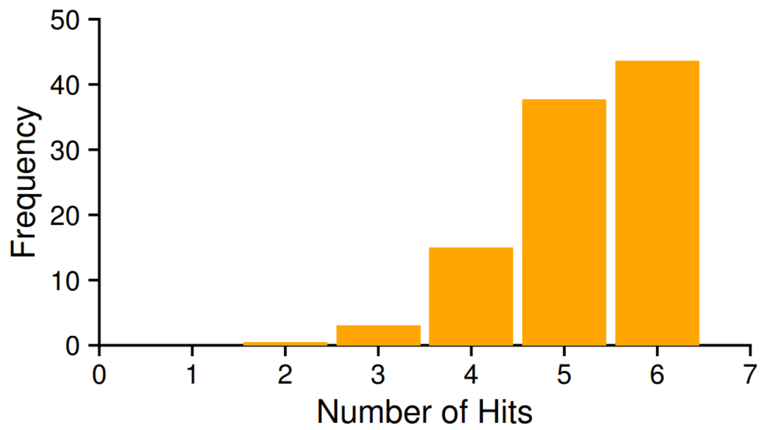
    cache가 커지더라도 hit-rate가 증가하지 않는다.
- **Example with the same access sequence and cache**
  - Three compulsory misses to pages 0, 1, and 2, and then hit on both 0 and 1
  - When page 3 is referenced, causing a miss, OS evicts page 0 that was the first one in order

$$hit\ rate = \frac{hits}{hits + misses} = \frac{4}{4+7} = 36.4\%$$

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Another Simple Policy: Random

- **Random replacement policy simply picks a random page to replace under memory pressure**
  - Random has properties similar to FIFO; simple to implement
  - Cons: unpredictable → performance varies
- **Example with the same access sequence and cache**
  - Three compulsory misses and OS randomly evicts the page upon a miss



Random performance over 10,000 trials:
sometimes it is as good as optimal (6 hits)
sometimes it does much worse (2 hits or fewer)

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

$$hit\ rate = \frac{hits}{hits + misses} = \frac{5}{5+6} = 45.5\%$$

Using History: LRU

- **FIFO or random policy might kick out an important page, one that is about to be referenced again**
  - We lean on the past and use history to improve our guess at the future (locality)
- **A family of simple historically-based algorithms are born**
  - Least-Frequently-Used (LFU) policy replaces the least-frequently-used pages
  - Least-Recently-Used (LRU) policy replaces the least-recently used ones
- **Example with the same access sequence and cache (LRU)**
  - Three compulsory misses and OS picks least-recently-used page for eviction

$$hit\ rate = \frac{hits}{hits + misses} = \frac{6}{6+5} = 54.5\%$$

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

  - Opposites of these algorithms:
    1) Most-Frequently-Used (MFU)
    2) Most-Recently-Used (MRU)

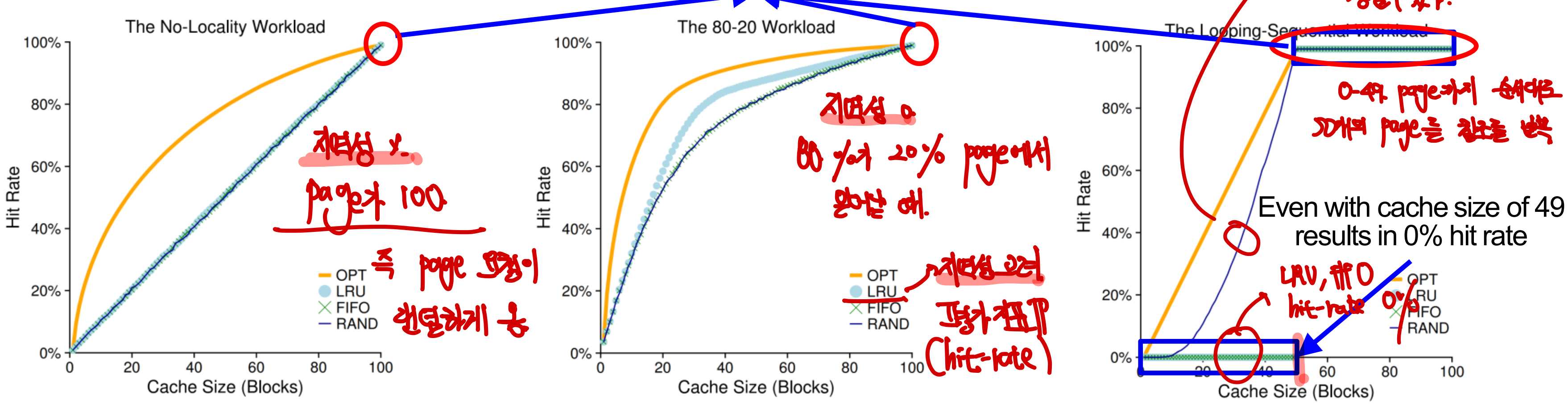
Workload Examples

10,000번의 접근

모든 page들을 다 담을 수 있을 만한 크기.

- **Examining more complex workloads to better understand**
  - 50 or 100 unique page accesses over time, choosing next page randomly → overall 10,000 accesses; cache size from 1 to 100 pages (holding 1 ~ all pages)
  1) No-locality workload: LRU, FIFO, and random all perform the same, with the hit rate exactly determined by the size of the cache, while optimal is the best
  2) 80-20 workload (80% references by 20% pages): while both random and FIFO do reasonably well, LRU does better (20% hot pages referenced again)
  3) Looping-sequence workload (0~49 page in a row and repeat): while both LRU and FIFO show a worst (kicking out older pages), random is better

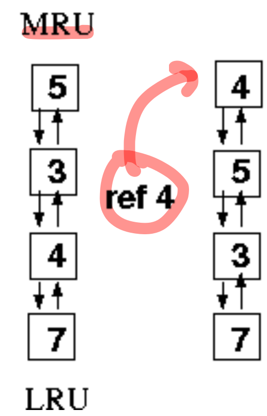
When the cache is large enough to fit the entire workload, it doesn't matter which policy you use; 100% hit rate



Implementing Historical Algorithms

→ policy 자체가 배열이 커지면 안되지 않을까.

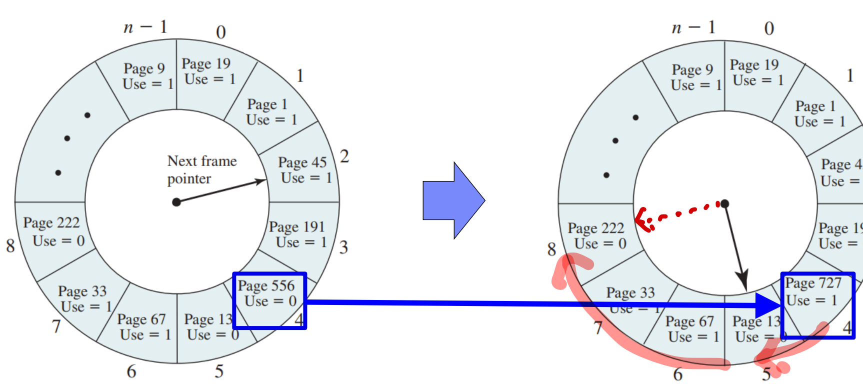
- **To implement LRU perfectly, a lot of work is required**
  - Upon each page access, some data structure needs to be updated to move this page to the front of this list (MRU side)
  - To keep track which pages were least- and most-recently used, accounting work on every memory reference is required
  - To speed-up, hardware's help is possible (e.g. time stamp)



- **Approximating LRU is more feasible for computation efficiency**

  → page의 수가 많아지면 배열 비용 크다 → 근사하자

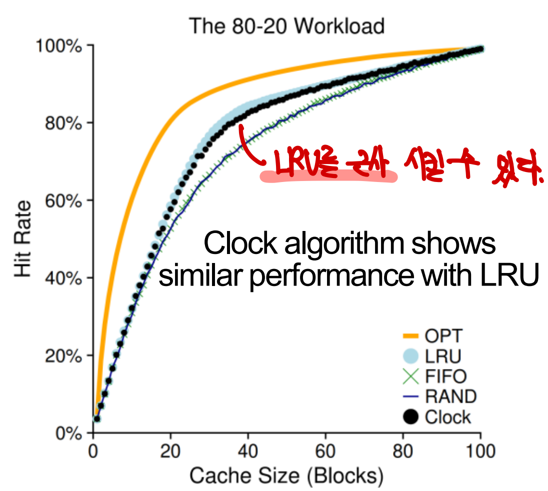
  - This requires some hardware support, a use (reference) bit; whenever a page is referenced, the bit is set by hardware to 1, then OS sets it to 0 by own algorithm
  - Clock algorithm: moving clock until finding a page with use bit = 0; then replacing it to new with use bit = 1



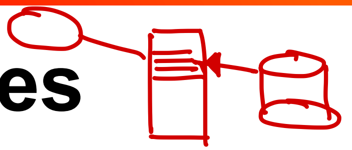
page 727 swap in

use=1

↳ 최근에 씀.



Considering Dirty Pages & Other Policies



- **If a page has been modified and is thus dirty, it must be written back to disk to evict it, which is expensive**

  cpu가 작성한 것들

  - If it has not been modified (and is thus clean), the eviction is free; the physical frame can be reused without additional I/O
  - Thus, some systems prefer to evict clean pages over dirty pages
  - To support this behavior, the hardware should include a modified (dirty) bit

  → 그래서 수정되지 않은 page 먼저 교체하도록 만듦.

- **Other policies for virtual memory system**

  (예측)
  먼저 가져올 수도 있음 ↔ on-demand.
  =prefetch

  - OS has to decide when to bring a page into memory; page selection policy:
    1) For most pages, OS brings the page into memory when it is accessed, "on demand"; demand paging
    2) OS could guess that a page is about to be used, and thus bring it in ahead of time; prefetching (e.g. code pages P & P+1 together)
  - Another policy determines how the OS writes pages out to disk:
    Many system collect a number of pending writes together in memory and write them to disk in one (more efficient) write; clustering or grouping of writes

  → delay write (지연 쓰기)를 통해 page에 대한 정보를 어느 정도 모아 한 번에 disk에 저장한다

  작은거 합쳐. 보냄. 큰거 하나.

Thrashing

= 실행중인 프로세스들이 요구하는 메모리의 크기가 실제 메모리의 크기보다 큰 경우.

- **What should OS do when physical memory is not enough to hold all the working sets of processes?**
  - The working set indicates a set of pages that a process is using actively
    자주 쓰는 것
  - The system will constantly be paging; most of the time is spent by OS paging data back and forth over the disk, called thrashing, degrading CPU utilization

mm ⟷ disk → phy mm이 부족하면 swap하는 데만 쓰임

- **Possible solutions are**
  - ① Admission control: reducing set of workings set to fit in memory by not running a subset of processes
    시스템의 사용 가능한 리소스를 바탕으로 새로운 프로세스를 수락할지를 결정
    → 늦게 실행
    → 일시적으로 대기시킬 수도 있음
  - ② Out-of-memory killer: killing some memory-intensive processes when memory is oversubscribed; causes problems
    프로세스 중에 메모리 많이 먹는거 죽임.
  - ③ Buy more memory!
    극단적인 수단. 매우 귀찮음