Operating System: Swapping Policies

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

Beyond Physical Memory: Policies

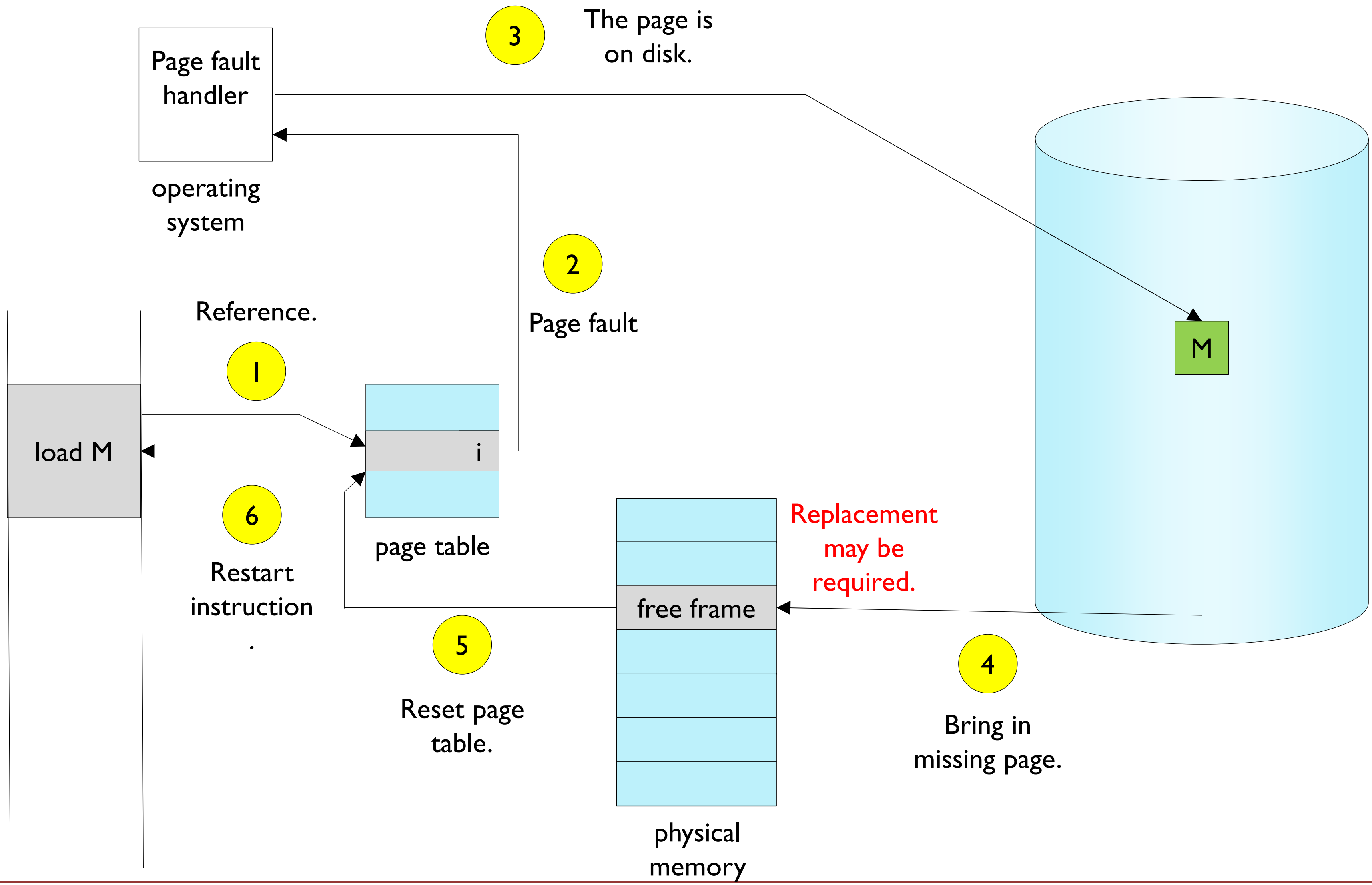
- Memory pressure forces the OS to start paging out pages to make room for actively-used pages

- Deciding which page to evict is encapsulated within the replacement policy of the OS

어떤 놈을
내쫓을 것이냐 ?

Demand Paging (review)

- Page fault handling



Demand Paging (Cont.)

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 - p) x memory access time + p x page fault service time
  - Example
    - ➢ Memory access time = 200 ns
    - ➢ Average page fault service time = 8 ms (=8,000,000ns)
    - ➢ EAT = (1 - p) x 200 + p x 8,000,000 = 200 + p x 7,999,800
    - ➢ If one access out of 1,000 causes a page fault, then
      - ✓ EAT = 8,200 ns
      - ✓ This is a slowdown by a factor of 40
- It is important to keep the page fault rate low
  - Good page replacement policy is required

여기서 하고 싶은 말은 page fault rate이 낮으면 낮을수록 EAT를 낮출수있으니 replacement policy가 추구해야하는 것은 이 fault rate를 얼마나 최소화 할수있는거임.

Page Replacement

- Page replacement
  - find some page not really in use ➔ swap it out
  - Good page replacement algorithm
    - ➢ results in minimum number of page faults

- Locality of reference Locality에 기반하여 안 쓰는 page를 예측
  - Same pages may be brought into memory several times
  - A phenomenon observed in most programs in practice
  - A program behavior that intensively references only a small number of pages at a certain time
    - ➢ ex. loop
  - Reason for the better performance of the paging system

Page Replacement

- Ideal algorithm
  - gains the lowest page-fault rate

- Algorithm can be evaluated by
  - running it on a particular string of memory references,
    - ➢ reference string ~ 이런예시를통해 평가
  - and computing the number of page faults on that string

- In some examples, the reference string is
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

The Optimal Replacement Policy

이상적인 교체정책

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed furthest in the future 가장 나중에 참조될 페이지를 내쫓는거임.
  - Resulting in the fewest-possible cache misses

결과적으로 가장 낮은 캐시 미스가 생기는데 어떻게 이론적으로 가장 낮은 페이지 fault를 보이는 case를 만들고 이와 실제 구현되는 알고리즘과 비교하기위함.

- Used for measuring how well your algorithm performs
  - Optimal algorithm presents lower bound of page
  - Serve only as a comparison point, to know how close we are to perfect

Tracing the Optimal Policy

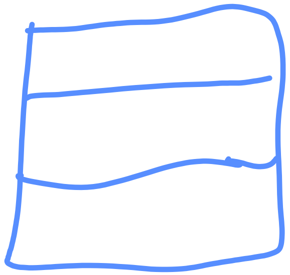
빈통에 넣기위해
오히려 생기는 미스~

Reference Row

0 1 2 0 1 3 0 3 1 2 1

cold-start 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



3칸짜리에
가능하다고 하면

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

Future is not known

A Simple Policy: FIFO

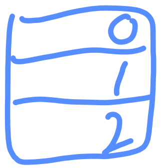
- Pages were placed in a queue when they enter the system

- When a replacement occurs, the page on the tail of the queue(the "<u>First-in</u>" pages) is evicted
  - It is simple to implement, but can't determine the importance of blocks

Tracing the FIFO Policy

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		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	2	3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

cold-start miss 제외시

$\frac{4}{8} = 50\%$

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

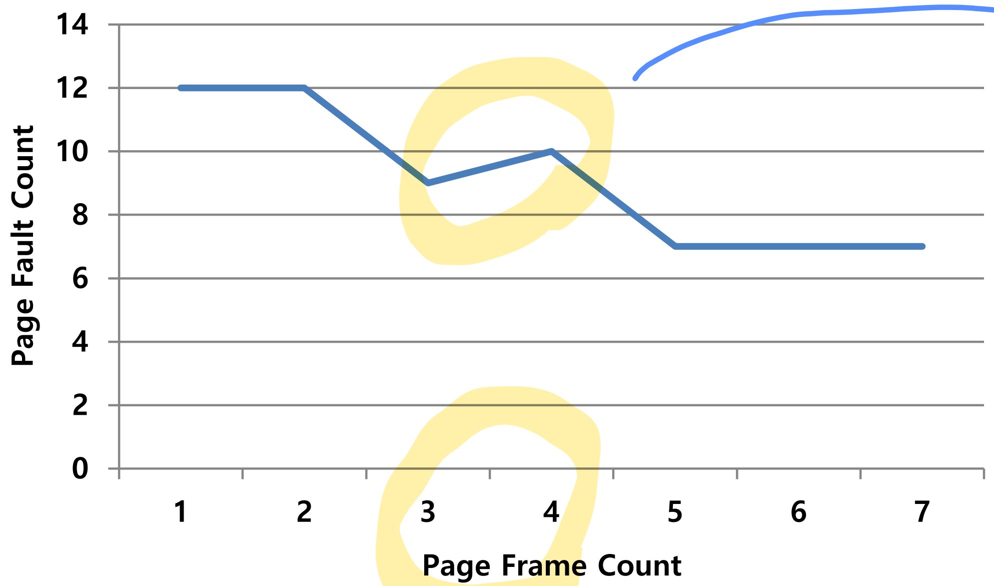
Even though page 0 had been accessed a number of times, FIFO still kicks it out

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



Frame의 크기가 늘었음에도 fault가 늘음 이를 BELADY'S ANOMALY 라고 함.

BELADY'S ANOMALY (Cont'd)

이런 일이 일어나는 이유는 Stack에서는 Frame 수가 증가하면 그 증가된 Frame이 이전 크기의 Frame들 다 가지고 있는 상태임. 근데 FIFO는 그렇지 않아서 생김.

PF rate = 9 / 12

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

PF rate = 10 / 12

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

Frame이 커졌음에도 이전 크기의 원소를 가지지 못해 생기는 현상.

Another Simple Policy: Random

그냥 랜덤픽.

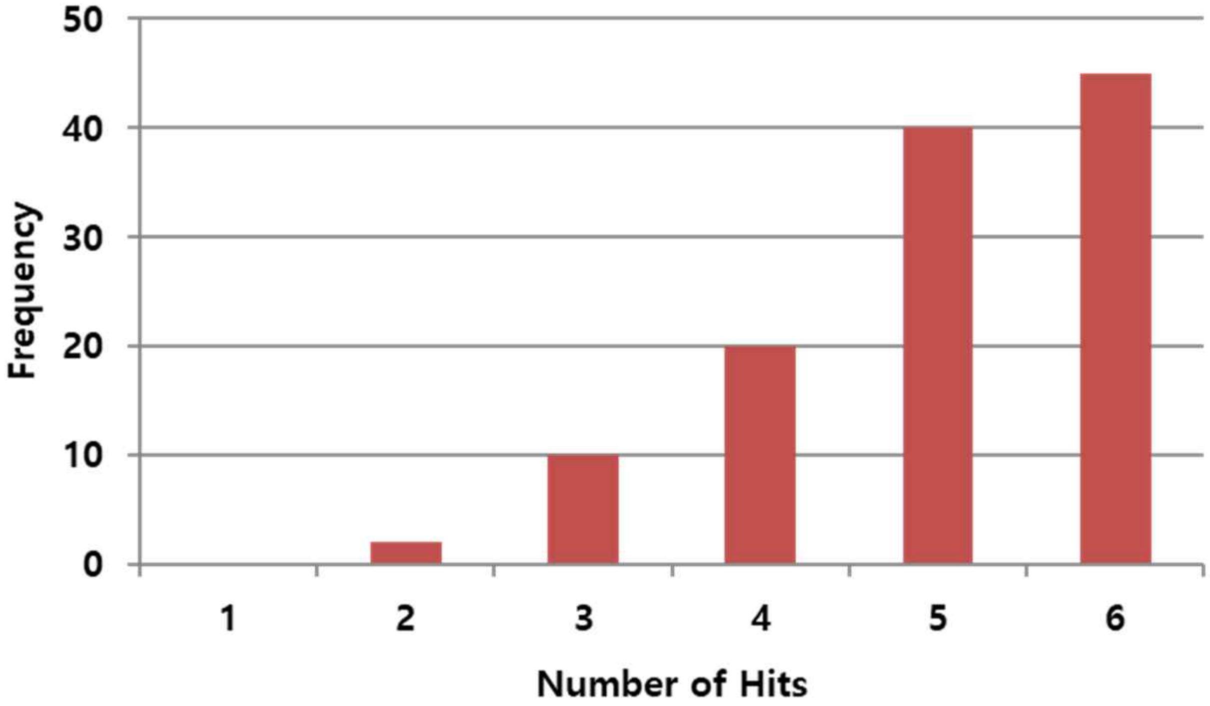
- Picks a random page to replace under memory pressure
  - It doesn't really try to be too intelligent in picking which blocks to evict
  - Random depends entirely upon how lucky Random gets in its choice

시스템관리 측면에서 안정성이 선호된다는 관점에서는 그리 좋은 알고리즘은 아님.

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

Random Performance

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



Random Performance over 10,000 Trials

Using History

이전을 보고

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

가장오래전참조가 교체대상

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

least recent used

least frequently used

가장적게사용된 것이

교체대상

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

Using History : LRU (Cont'd)

- Least Recently Used
  - Replace the page that has not been used for the longest time in the past
  - Use past to predict the future
    - ➢cf. OPT wants to look at the future
  - With locality, LRU approximates OPT
  - "Stack" algorithm: does not suffer from Belady's anomaly 이거 안일어남.
  - Harder to implement: must track which pages have been accessed 접근했던 시간을 저장해야하니 구현이 어려움.
  - Does not consider the frequency of page accesses
  - Does not handle all workloads well

Using History : LRU (Cont'd)

<Frame 개수>

n+1

n

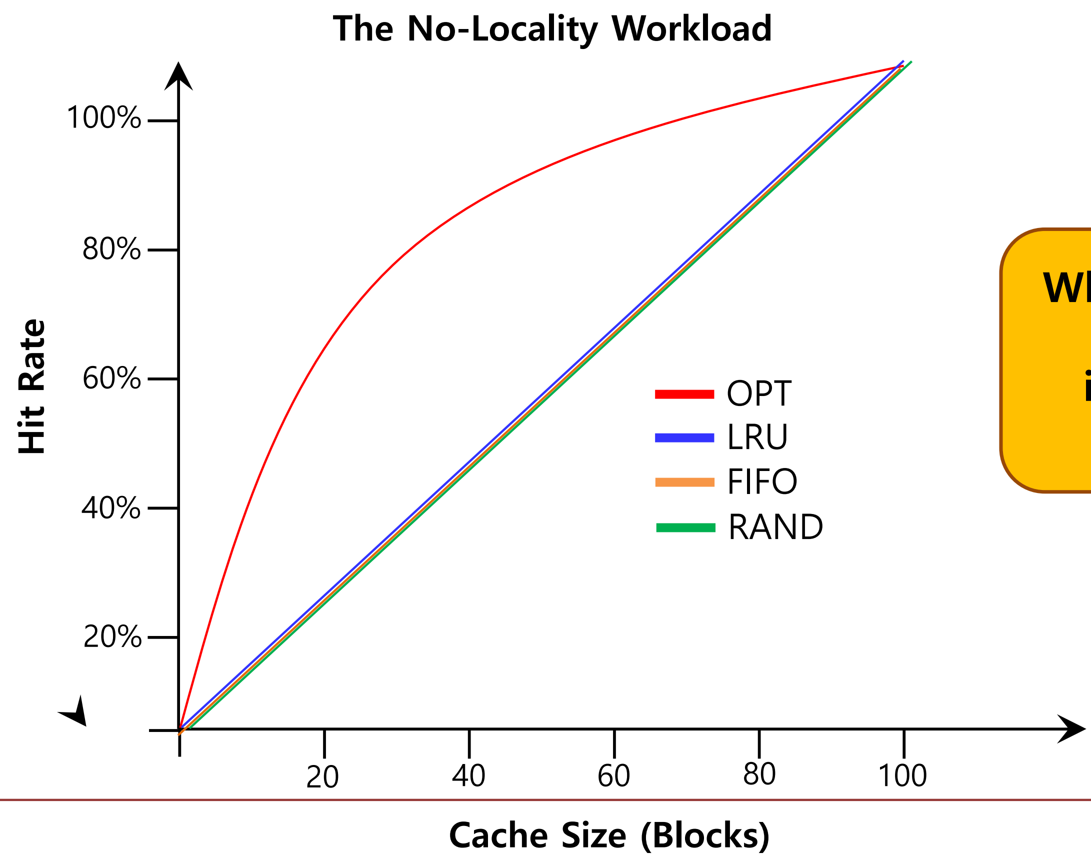
- Stack algorithms
  - Policies that guarantee increasing memory size does not increase the number of page faults (e.g. OPT, LRU, etc.)
  - Any page in memory with *m* frames is also in memory with *m+1* frames (called stack property)

PF rate = 10 / 12

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
Stack distance:	∞	∞	∞	∞	4	4	∞	3	3	5	5	5
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
							3	3	3	4	5	1
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss

Workload Example : 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



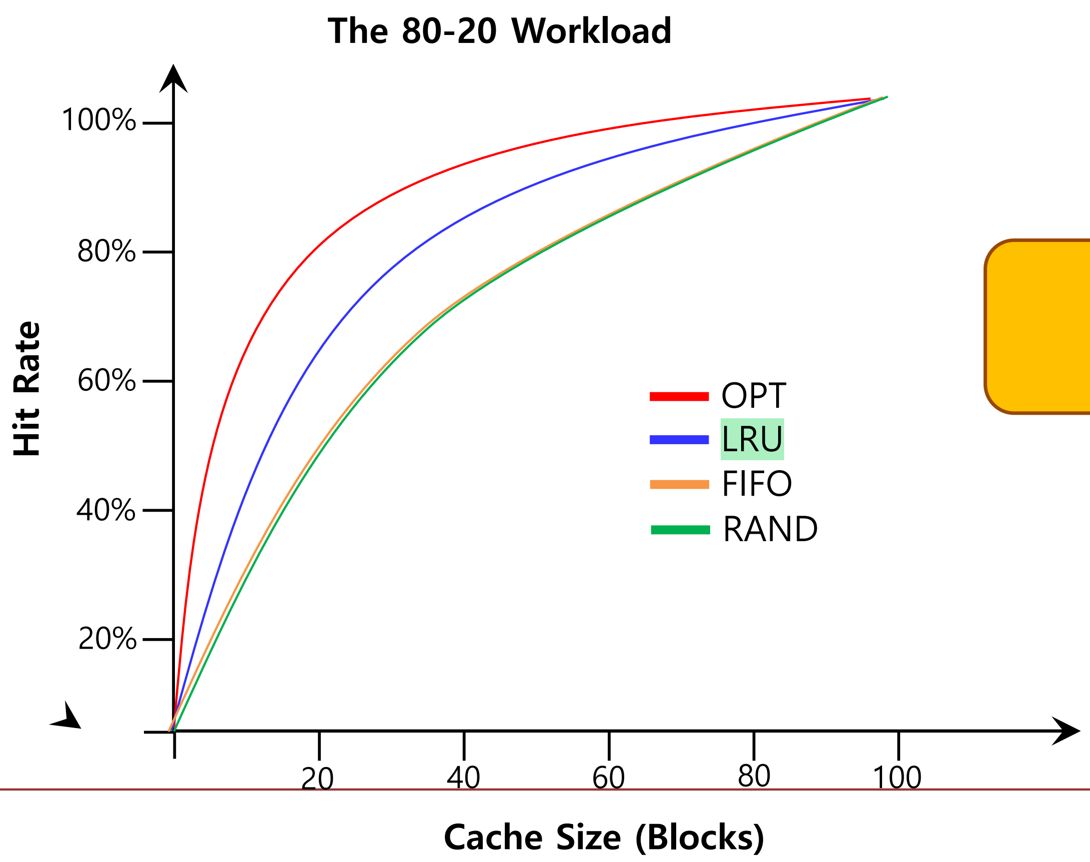
work load에 Locality가 없다면
알고리즘간 차이가 없다.

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

Workload Example : The 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

참조의 80%가 20% page에서 일어난다
= Locality 고려.



LRU is more likely to hold onto the hot pages

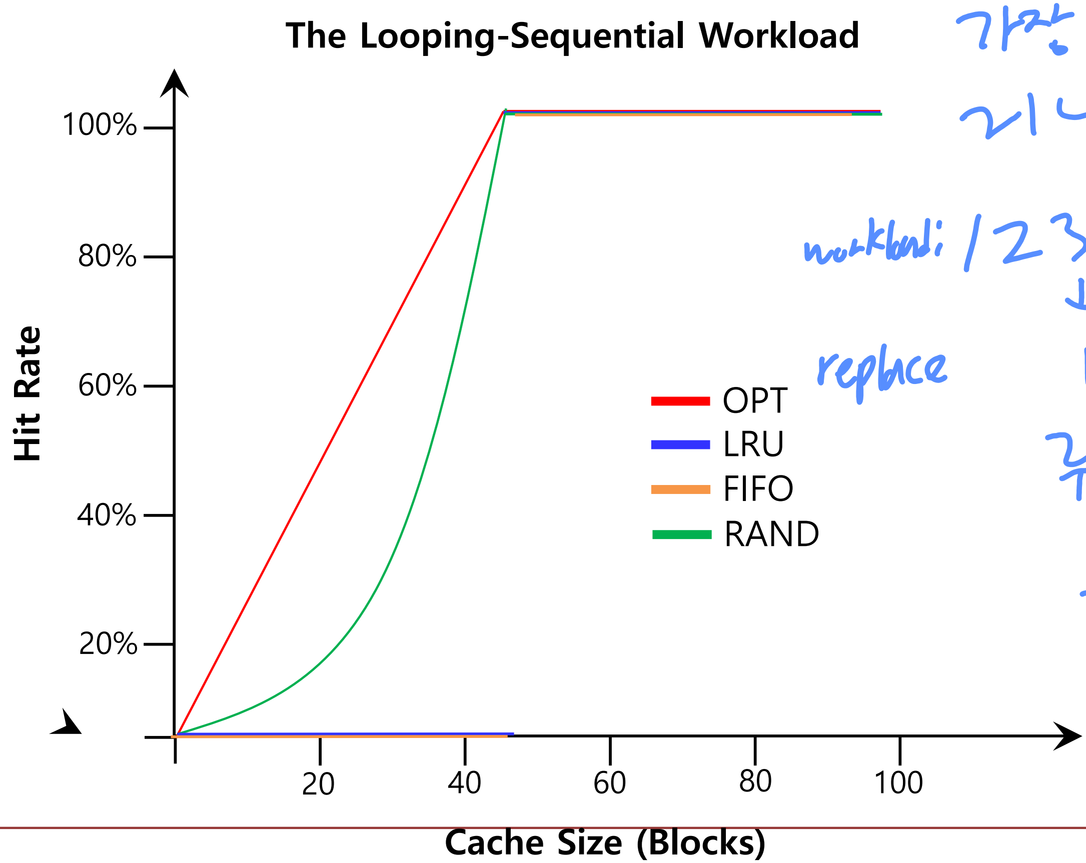
이때는 LRU가 좋음.

Workload Example : The Looping Sequential

workload에 따른 알고리즘의 선택이 필요.

- 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

루프를 돈다고 생각하면 가장 예전 값을 계속사용하는 거니 LRU 성능이 ↓



workload: 123 123 123 123

replace 1 2 3 1 2 3 1 2 3 1

루프를도는 그 크기를 넘어설때 까진 0% 임. 유발.

Implementing Historical Algorithms

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

메모리 참조 기록을 저장할 비트 필요.

Approximating LRU

- Require some hardware support, in the form of a use bit
  - 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

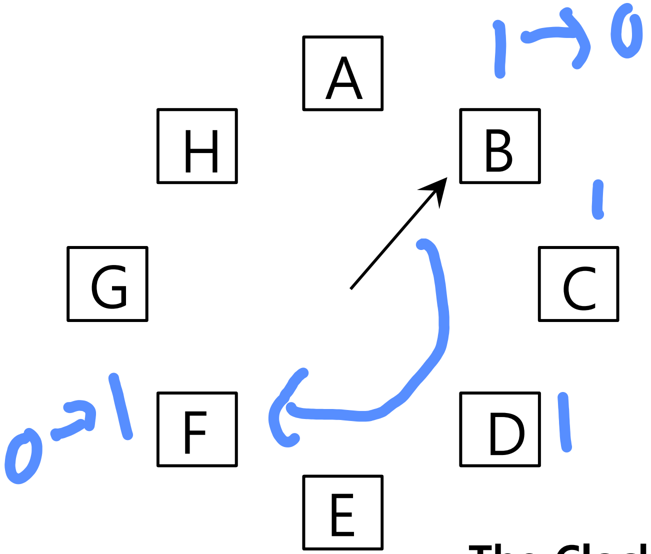
- Clock Algorithm
  - All pages of the system arranges in a circular list
  - A clock hand points to some particular page to begin with

Clock Algorithm

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

시간 기록 X LRU와

참조 여부만 동일 X



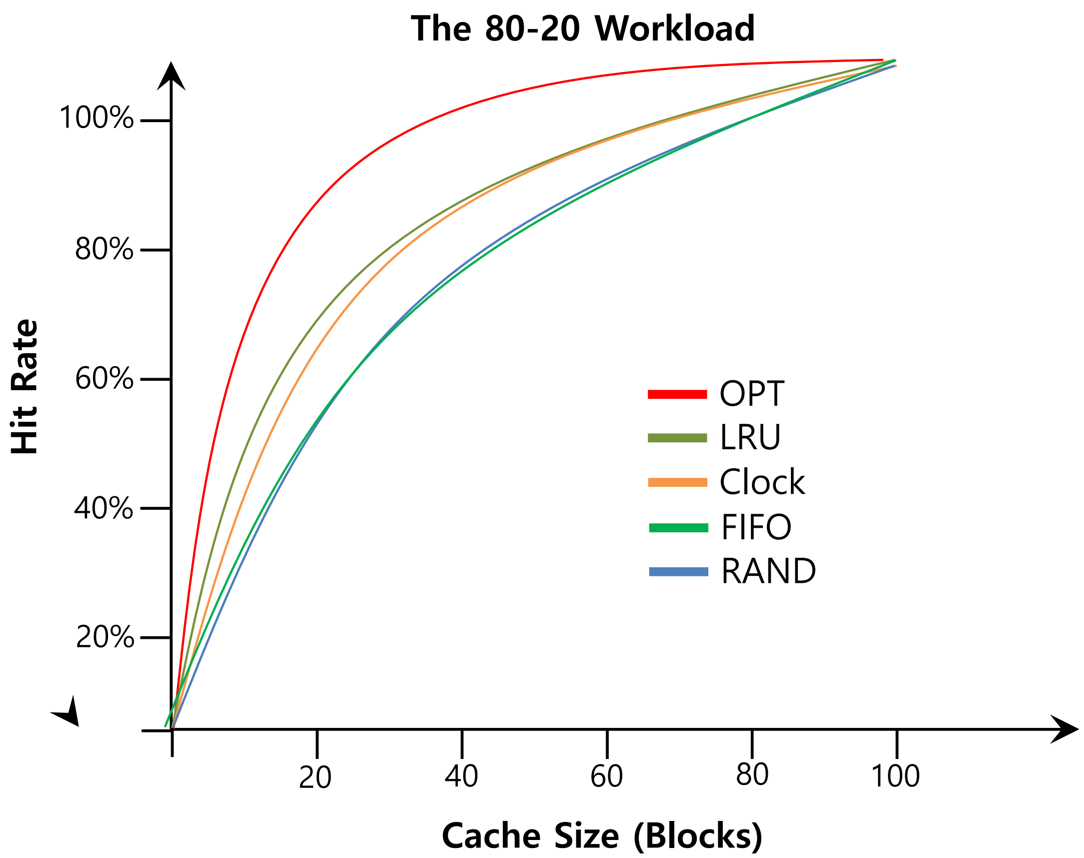
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

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

Workload with Clock Algorithm

- Clock algorithm doesn’t do as well as perfect LRU, it does better then approach that don’t consider history at all



Considering Dirty Pages

교체할때
이런비트
도 참고.

- The hardware include a <u>modified bit</u> (a.k.a <u>dirty bit</u>)
  - Page has been **<u>modified</u>** and is thus **<u>dirty</u>**, it must be written back to disk to evict it
  - Page has not been modified, the eviction is free

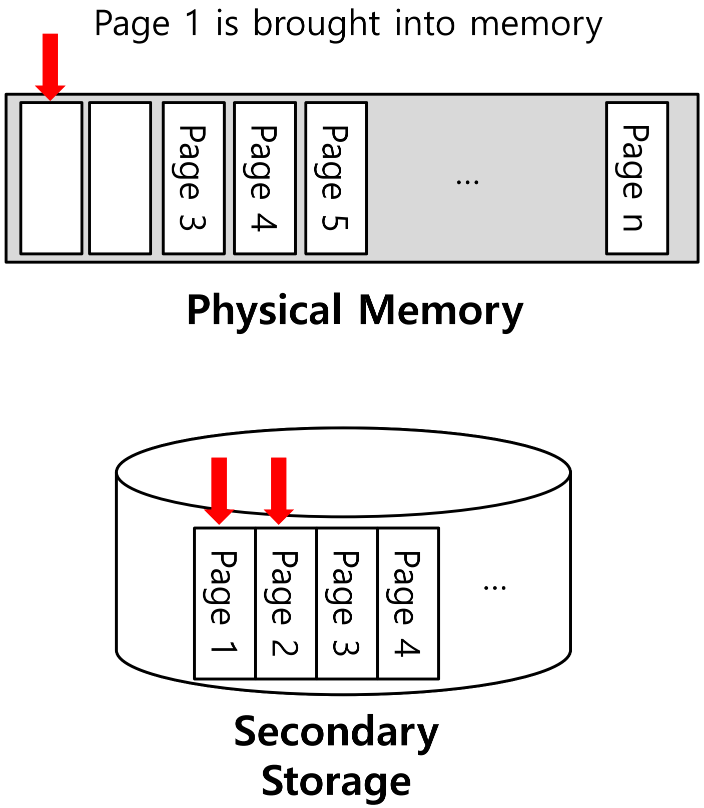
Page Selection Policy

- The OS has to decide when to bring a page into memory
- Presents the OS with some different options

Prefetching

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

Locality 를고려
근처 pages를
가져오는게
어때~?

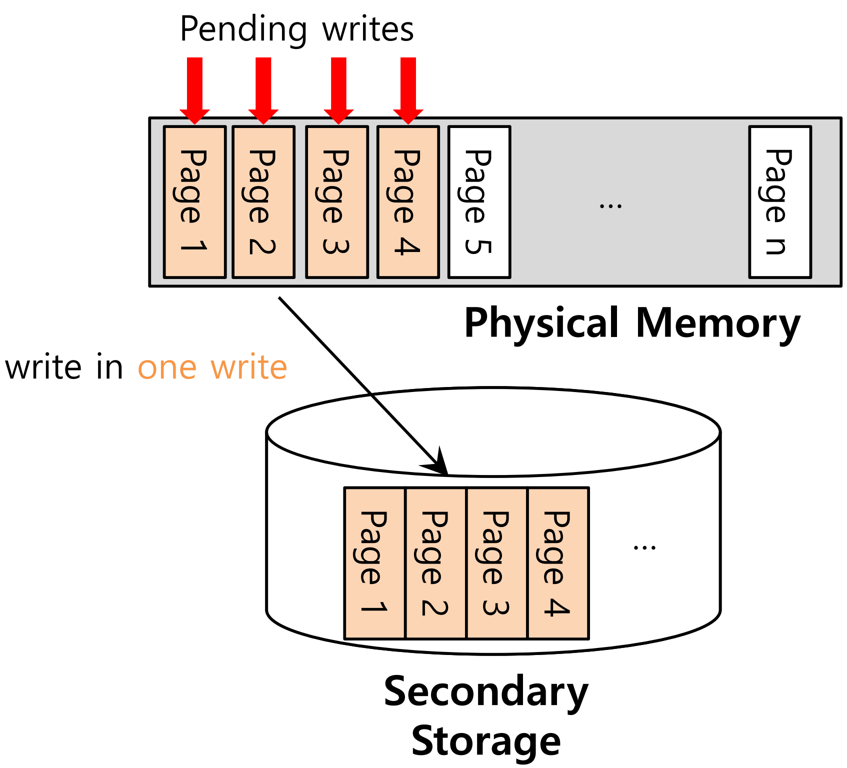


Page 2 likely soon be accessed and
thus should be brought into memory too

Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write
  - Perform a <u>single large write</u> more efficiently than <u>many small ones</u>

저장할 때도
그룹해서 저장.



Thrashing

- If a process does not have "enough" page frames, page fault rate is very high
- Thrashing
  – A process is busy in swapping pages in and out
- Why does thrashing occur?
  – Σ size of locality > total memory size →

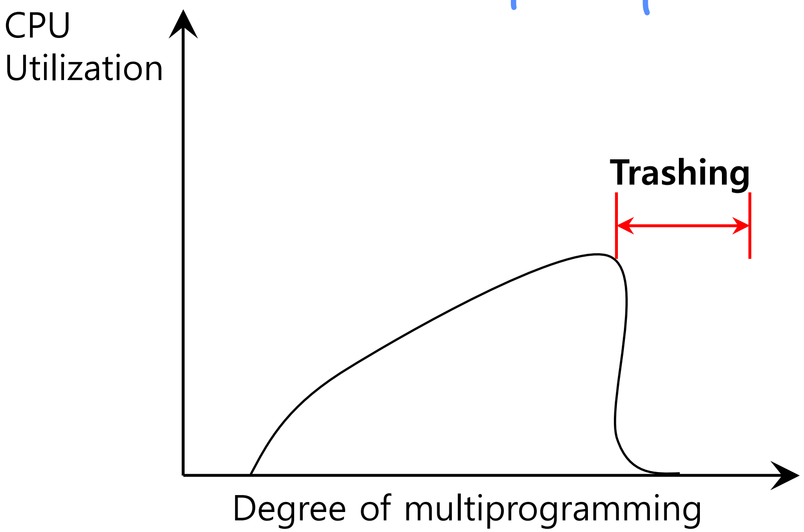
ex) A process
4 1234 1234 . . . . .
→ 3

4+2=6

3+1=4

B process
2 5656 . . . . . .
Page
1

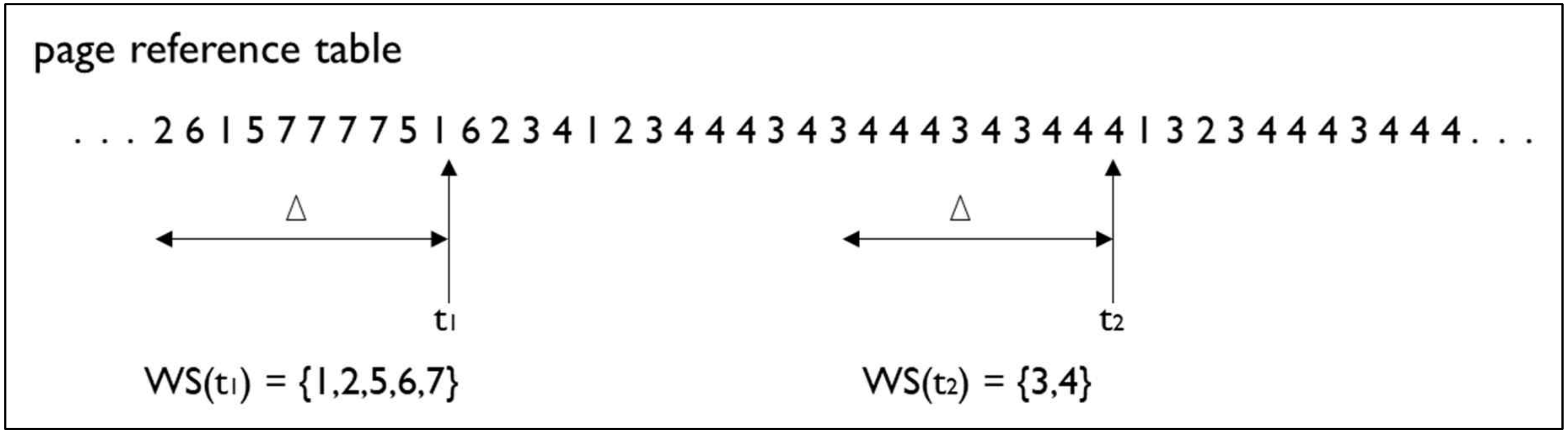
한마디로 swapping
시간쓰느라
CPU Utilization ↓



Working-set Model

특정시점에서 그 이전 구간의 |
Locality를 window로 구하는거

- Working set model
  - is based on the assumption of locality
  - $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i (working set of Process P_i)
  - total number of pages referenced in the most recent Δ
    - ➢ if Δ is too small, it will not encompass entire locality
    - ➢ if Δ is too large, it will encompass several localities
      - ✓ if $\Delta = \infty$, it will encompass entire program



- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - If $D > m$, Thrashing