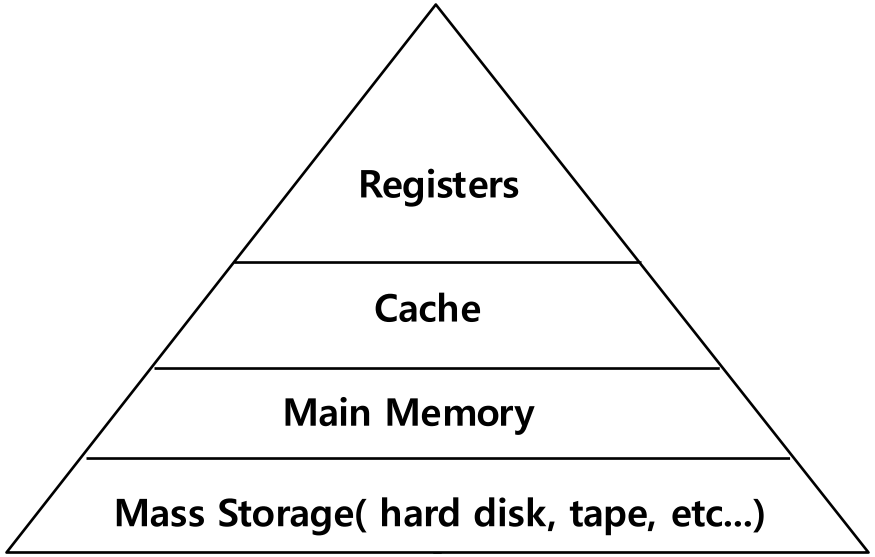
Memory Management Policy

Beyond Physical Memory: Mechanisms

- Require an additional level in the memory hierarchy
  - OS need a place to stash away portions of address space that currently aren't in great demand
  - In modern systems, this role is usually served by a hard disk drive



Memory Hierarchy in modern system

Single Large Address for a Process

- Always need to first arrange for the code or data to be in memory when before calling a function or accessing data

- To Beyond just a single process
  - The addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running process

Swap Space

- Reserve some space on the disk for moving pages back and forth
- OS need to remember to the swap space, in page-sized unit

	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
Swap Space	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

Physical Memory and Swap Space

Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk
  - When the hardware looks in the PTE, it may find that the page is not present in physical memory

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

What If Memory Is Full ?

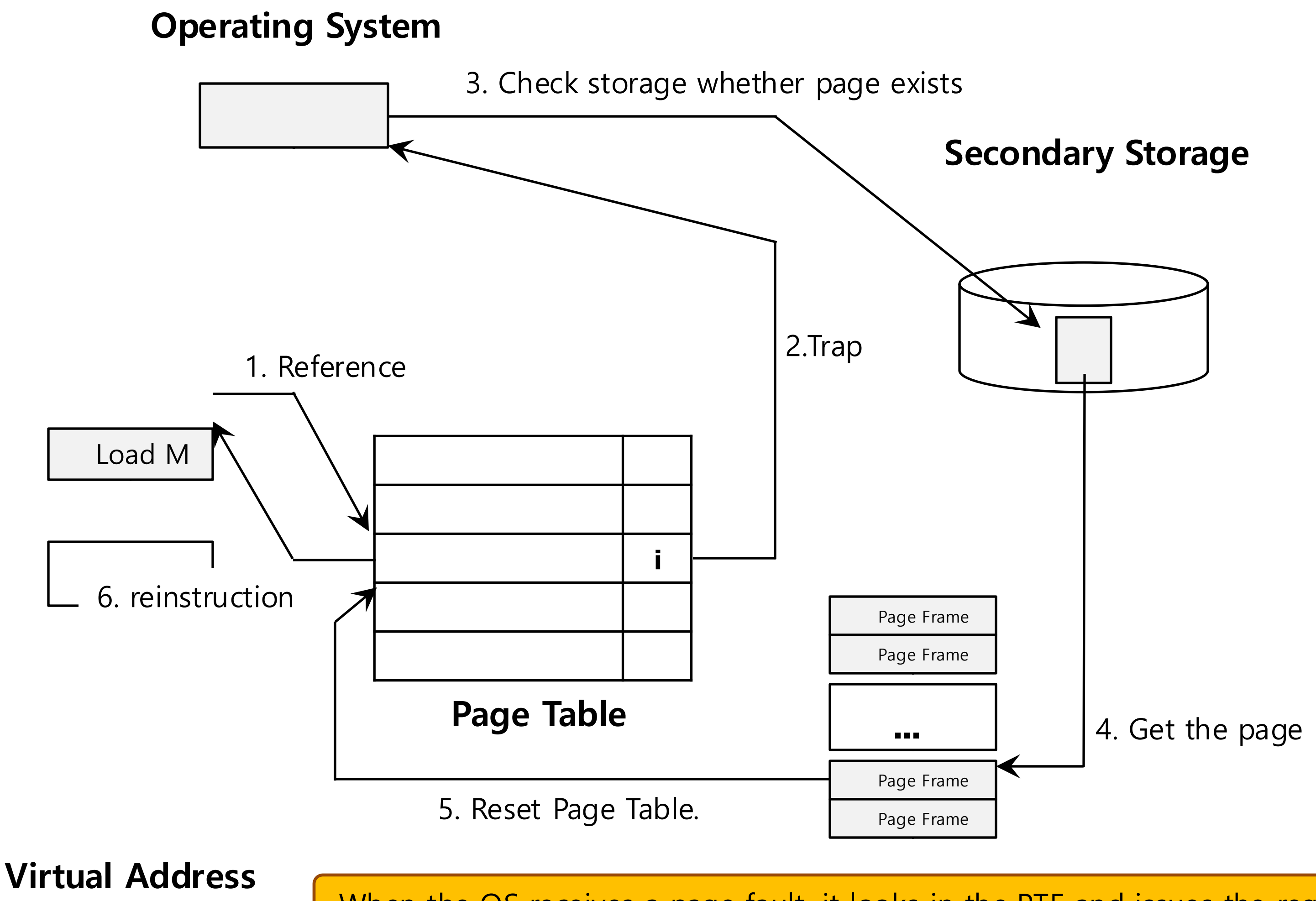
- The OS like to page out pages to make room for the new pages the OS is about to bring in
  - The process of picking a page to kick out, or replace is known as page-replacement policy

The Page Fault

- Accessing page that is not in physical memory
  - If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk addresses



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

Page Fault Control Flow – Hardware

```
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)
```

Page Fault Control Flow – Hardware

```
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)
```

Page Fault Control Flow – Software

```
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
```

- 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

When Replacements Really Occur

- 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
  - 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

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

Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses
- The number of cache hits and misses let us calculate the *average memory access time(AMAT)*

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Arguement	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

The Optimal Replacement Policy

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed <u>furthest in the future</u>
  - Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect

Tracing the Optimal 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	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

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 "**First-in**" pages) is evicted
  - It is simple to implement, but can't determine the importance of blocks

Tracing the FIFIO 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		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

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

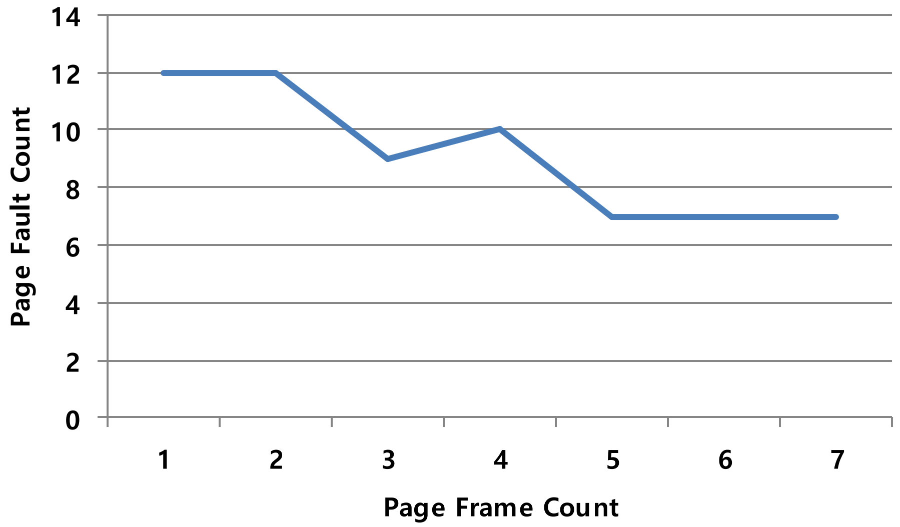
Even though page 0 had been accessed a number of times, FIFO still kicks it out

BELADY'S ANOMALY

- 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



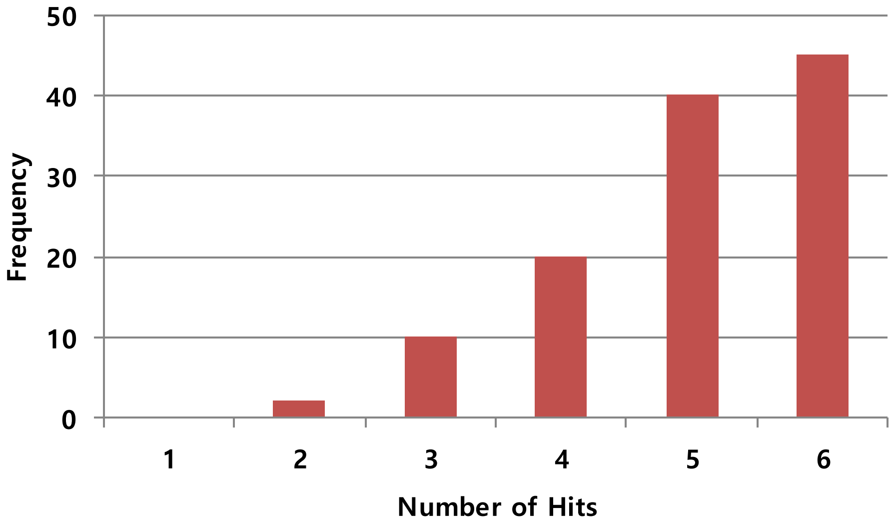
Another Simple Policy: Random

- 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

Random Performance

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



Random Performance over 10,000 Trials

Using History

- Lean on the past and use **history**
  - 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

Using History : LRU

- Replaces the least-recently-used page

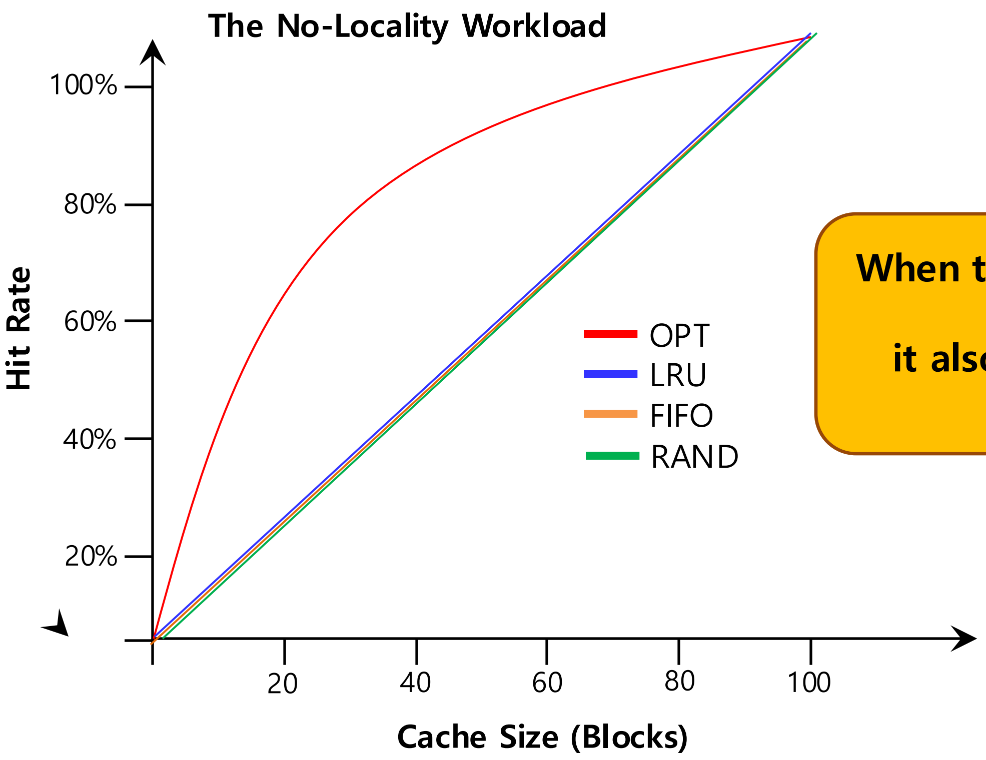
Reference Row

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

Workload Example : The No-Locality Workload

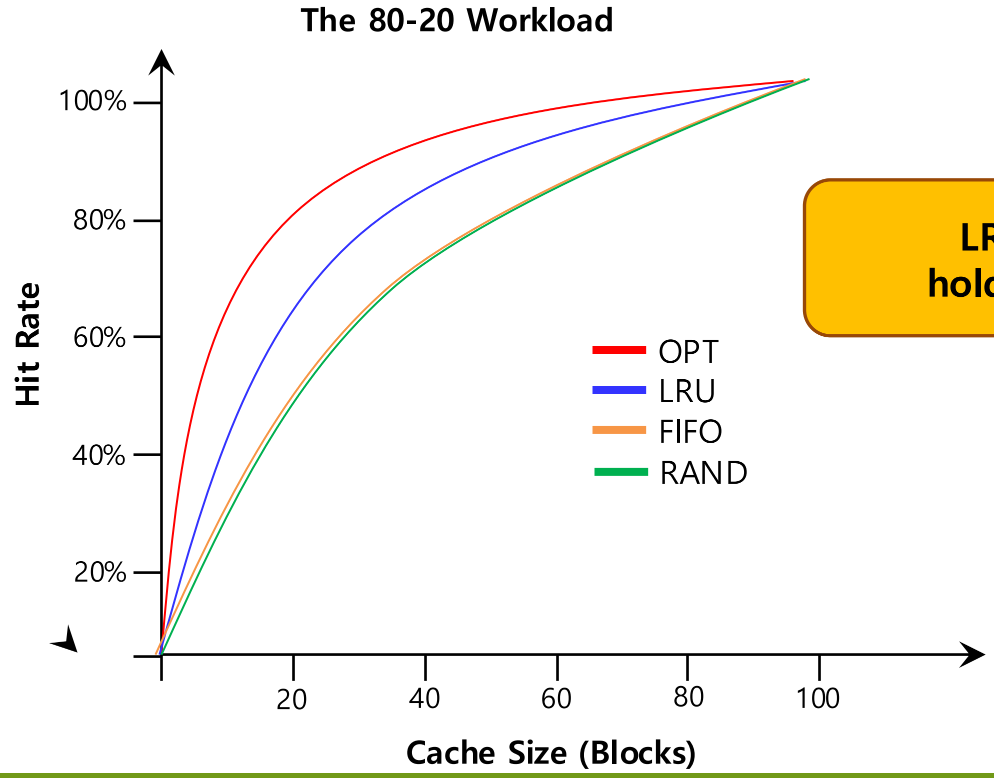
- Each reference is to a random page within the set of accessed pages
  - Workload accesses 100 unique pages over time
  - Choosing the next page to refer to at random



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

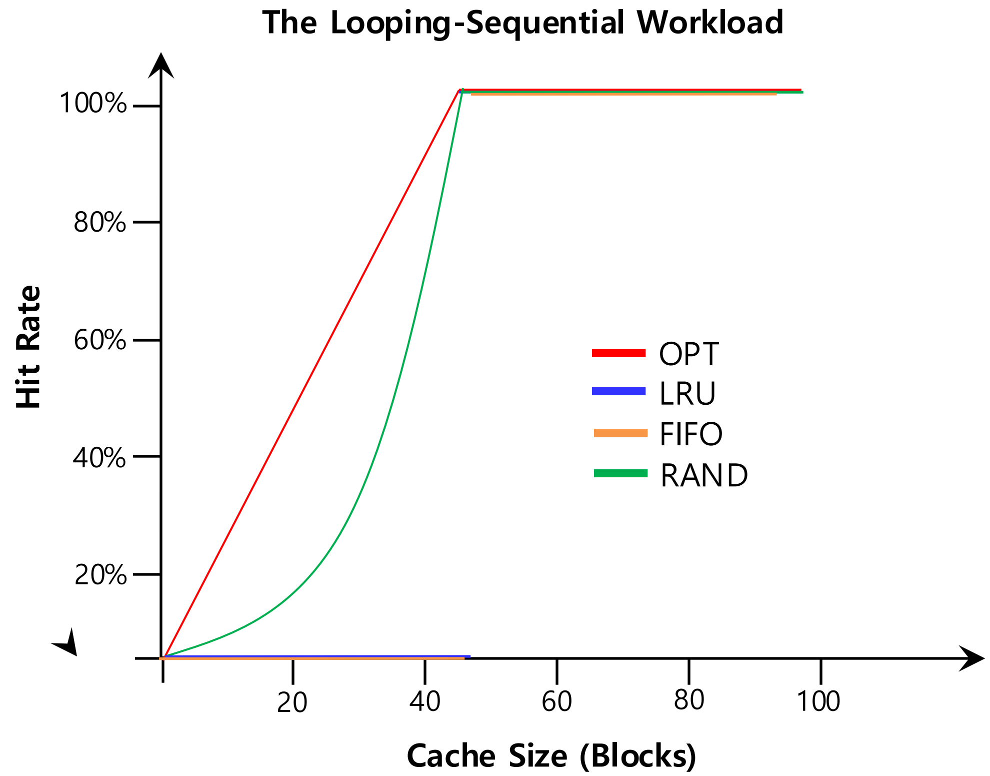
- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages



LRU is more likely to hold onto the **hot pages**

Workload Example : The 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



Implementing 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

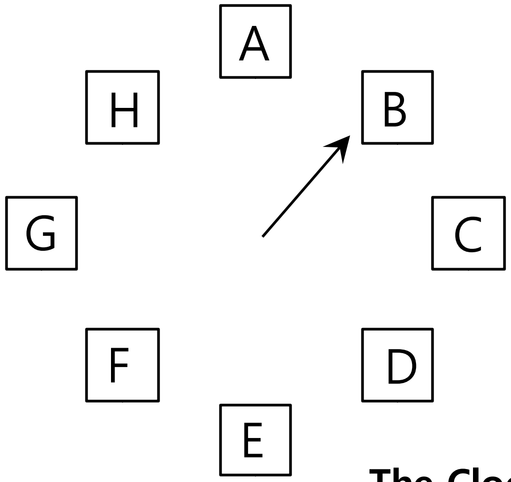
Approximating LRU

- Require some hardware support, in the form of a **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



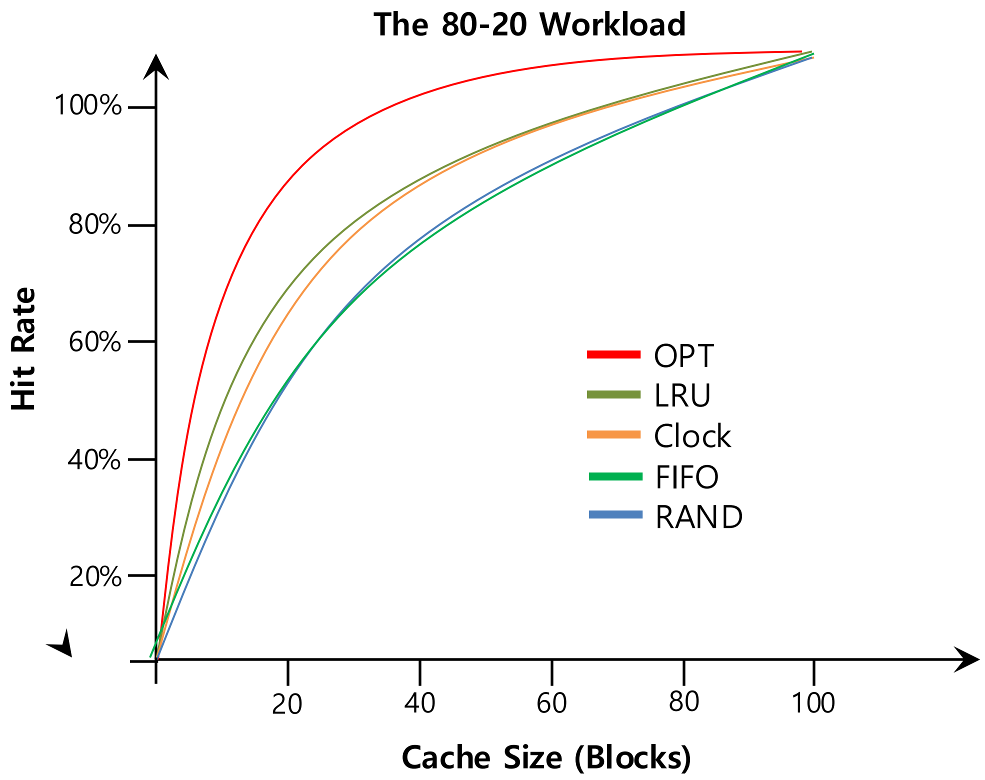
Use bit	Meaning
0	Evict the page
1	Clear **<u>Use bit</u>** and advance hand

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

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

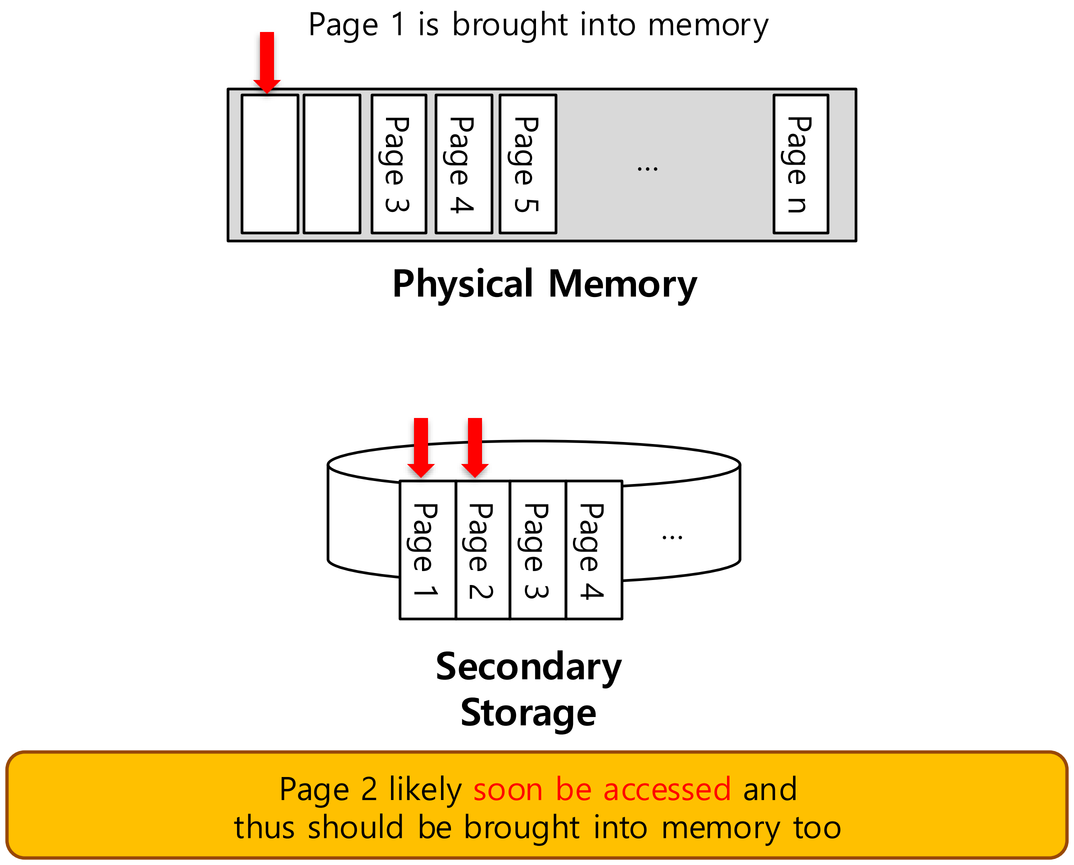
- The hardware include a **modified bit** (a.k.a **dirty bit**)
  - Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it
  - Page has not been modified, the eviction is free

Page Selection Policy

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

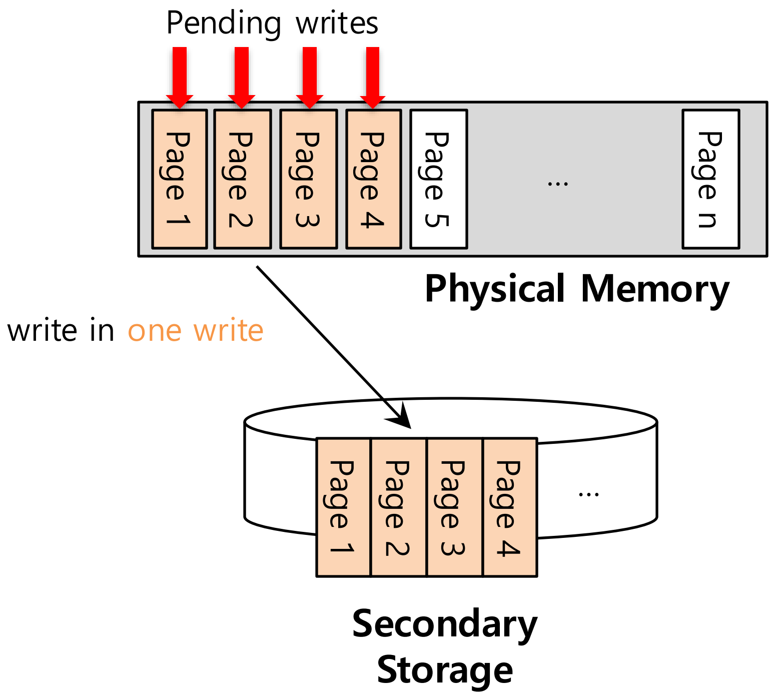
Prefetching

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



Clustering, Grouping

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



Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory
  - Decide not to run a subset of processes
  - Reduced set of processes working sets fit in memory