# CSCI3150 Introduction to Operating Systems

## Lecture 12: Memory Management Part III: Swapping

**Hong Xu**

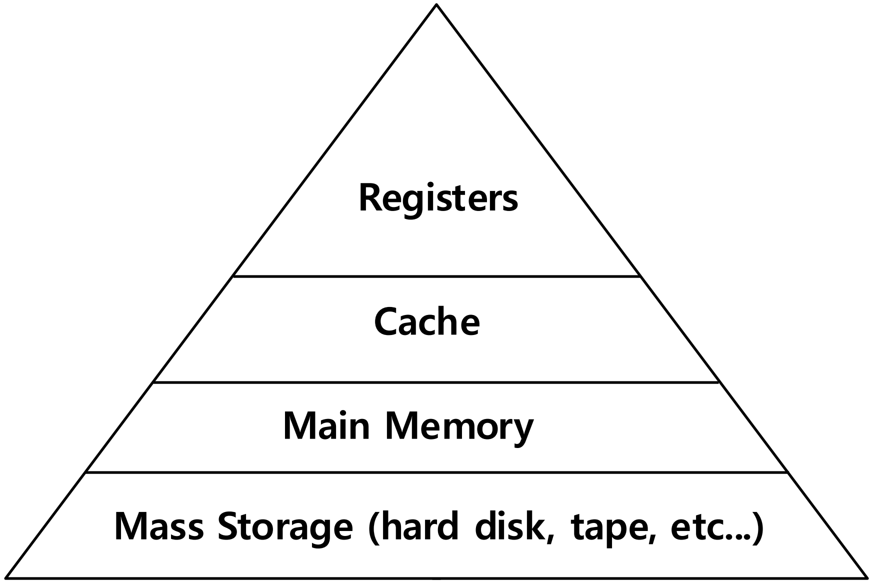
https://github.com/henryhxu/CSCI3150

# Overview

- Mechanisms
- Policies

# Beyond Physical Memory: Mechanisms

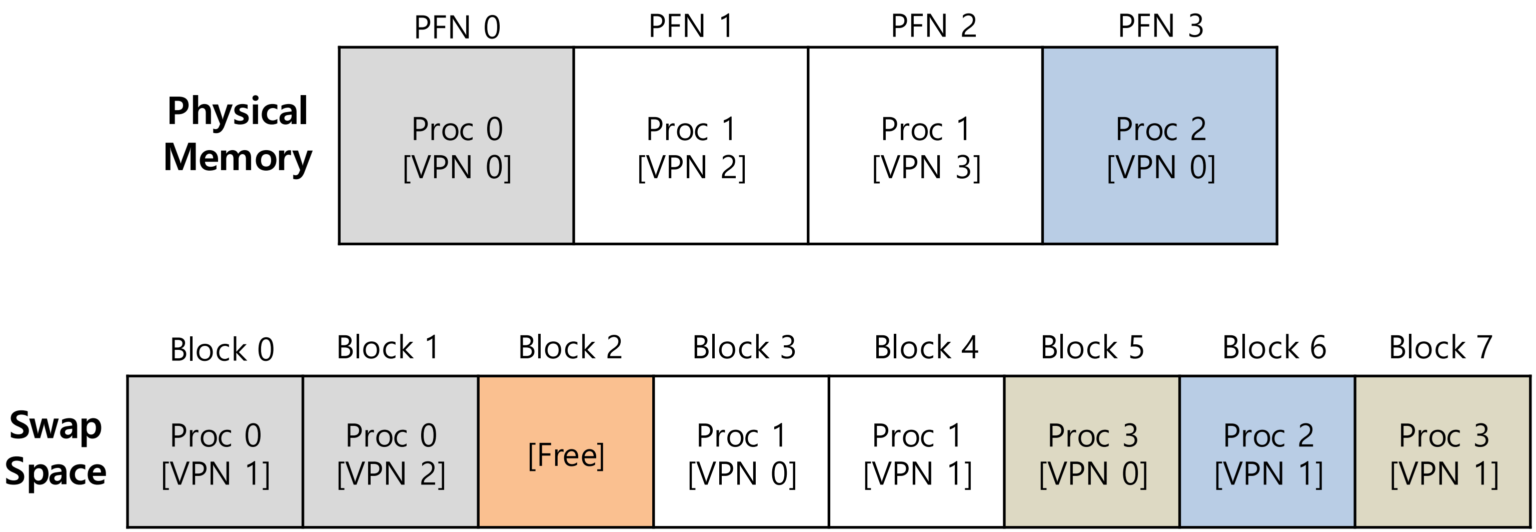
- Use part of disk as memory
  - OS needs a place to stash away portions of address space that currently aren't in great demand.
  - In modern systems, this role is usually served by a hard disk drive.




**Memory hierarchy in modern system**

# Swap Space

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




**Physical Memory and Swap Space**

# Present Bit

- Add some machinery higher up in the system to support swapping the 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. |

# Concepts

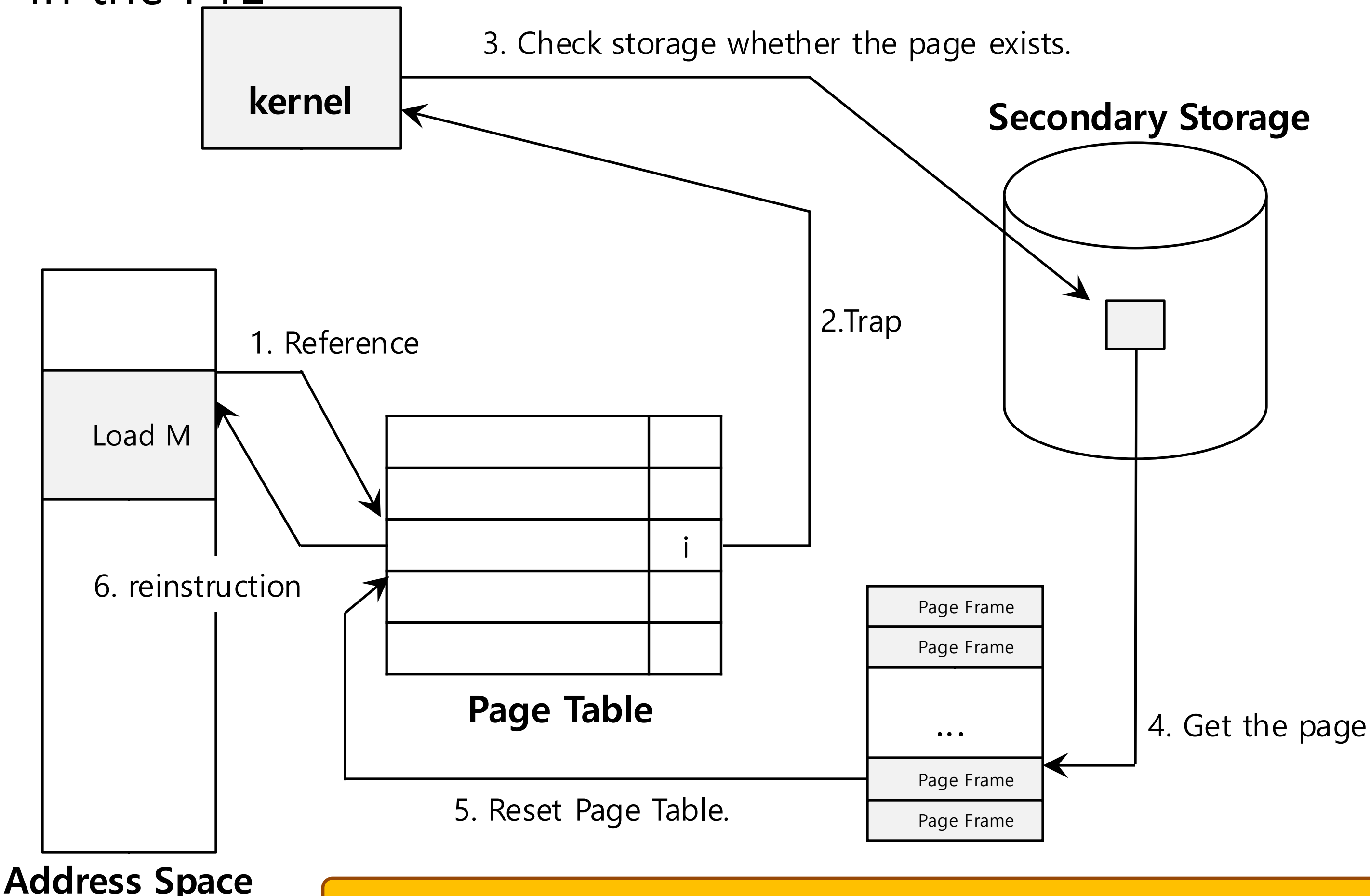
- Page fault
  - Accessing page that is not in physical memory.
  - If a page is not present and has been swapped to disk, the OS needs to swap the page back into memory in order to service the page fault.
- Page replacement
  - The OS likes to page out some pages to make room for the new ones it is about to bring in
  - The process of picking a page to evict (or replace) is known as page-replacement policy.

# When to Perform Page Replacement


- Lazy approach...
  - OS waits until memory is full to start replacing pages.
  - This is clearly unrealistic... Do not procrastinate!
- Swap Daemon, Page Daemon
  - When there are fewer than LW (low watermark) pages available, a background thread that is responsible for freeing memory runs.
  - The thread evicts pages until there are HW (high watermark) pages available.

# Page Fault Control Flow

- Bits used for data such as the page's PFN are used for a disk address in the PTE




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)
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

# Summary

- Swapping: making the part of disk as memory
- Present bit required

# Swapping Policies

# Goal of Cache Management

- To minimize cache misses
- To improve *average memory access time (AMAT)*

$T_M \ll T_D$

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

| Notation | Meaning |
|---|---|
| $T_M$ | The cost of accessing memory |
| $T_D$ | The cost of accessing disk |
| $P_{Hit}$ | The probability of hitting the data item in the cache |
| $P_{Miss}$ | The probability of not finding the data in the cache |

# The Optimal Replacement Policy

- Lead to the fewest number of misses overall
  - Replace the page that will be accessed furthest in the future.
  - Result in the fewest possible cache misses.
- Serve only as a comparison point, to know how close we are to perfection.

# 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 unknown!**

# A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the head of the queue (the "**first-in**" page) is evicted.
  - Simple to implement
  - Agnostic to the importance of pages

# 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 | | 3,0,1 |
| 2 | Miss | 3 | 0,1,2 |
| 1 | Hit | | 0,1,2 |

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

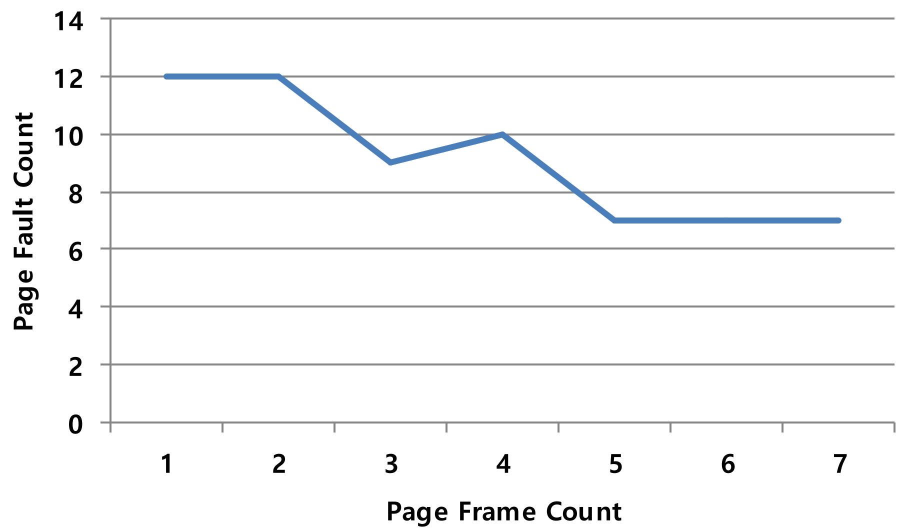
**Even though page 0 had been accessed a number of times, FIFO still kicks it out.**

# 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 page 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 |

# Using History

- Learn on the past and use **history**.
  - Two types 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 replaced as it clearly has some value | LFU |

# Using History: LRU

- Replace the least-recently-used page.

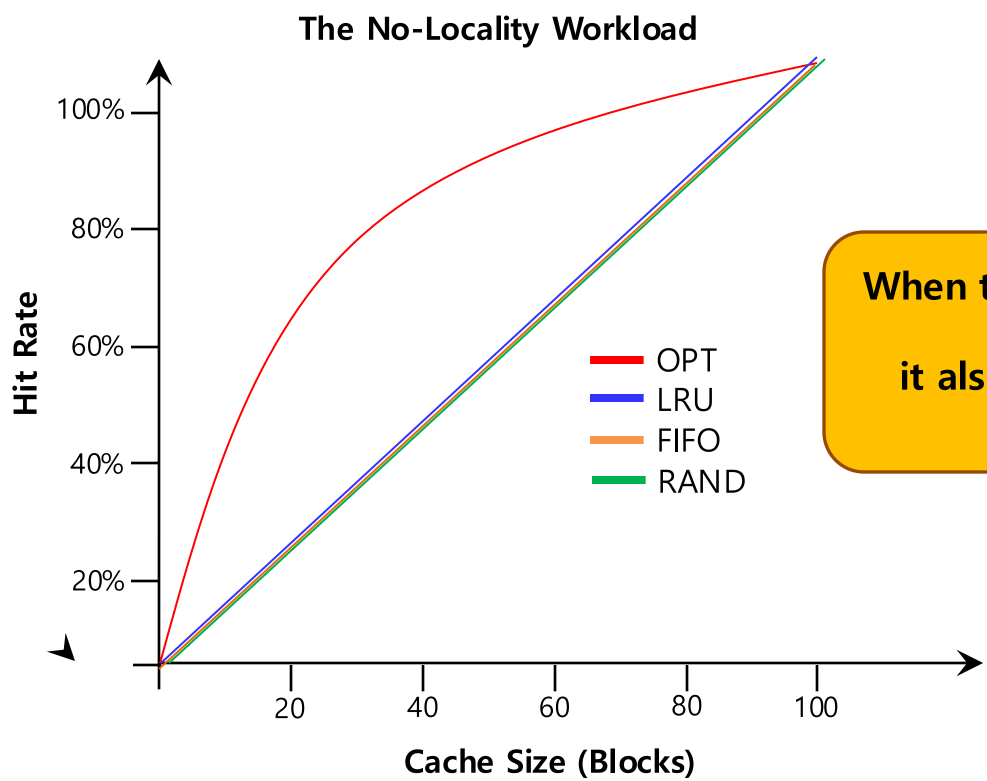
**Reference Row**

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

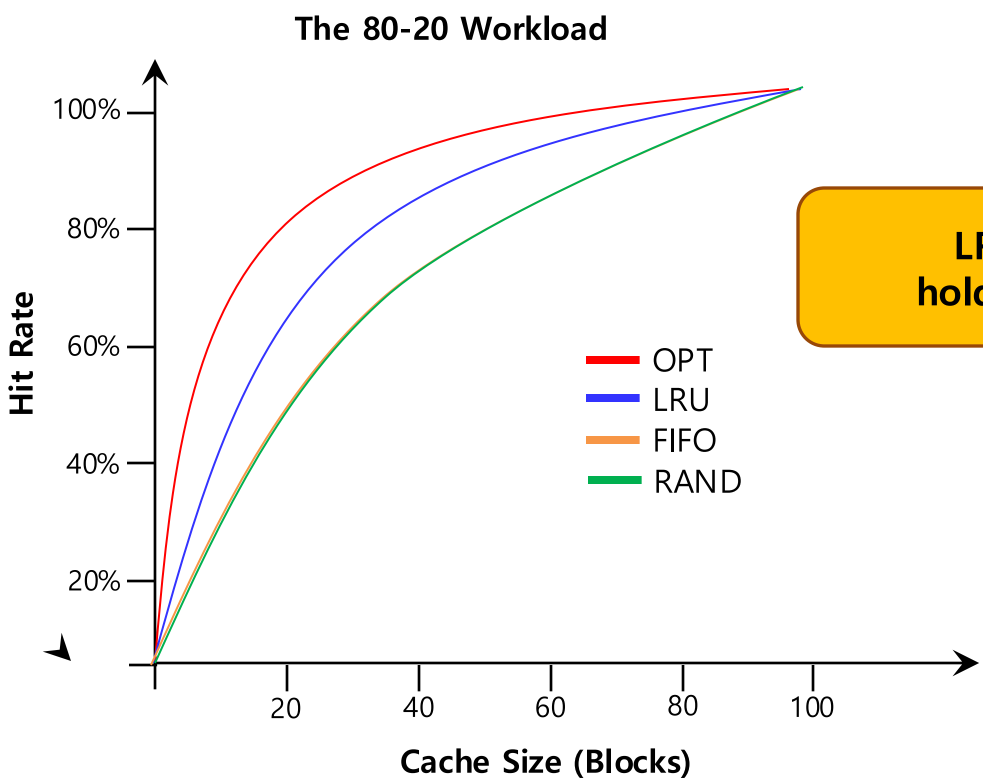
- Each reference is to a random page in a set of 100 pages
  - Workload has 100 accesses over time.
  - Choosing the next page to refer to at random




**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

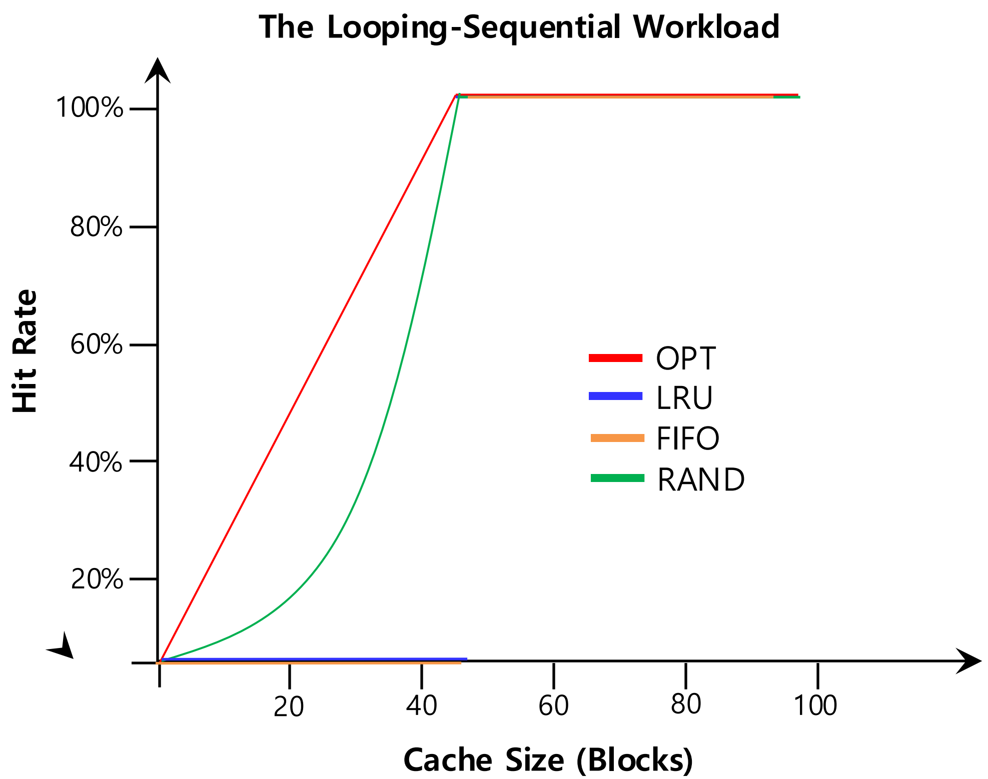
- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.




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

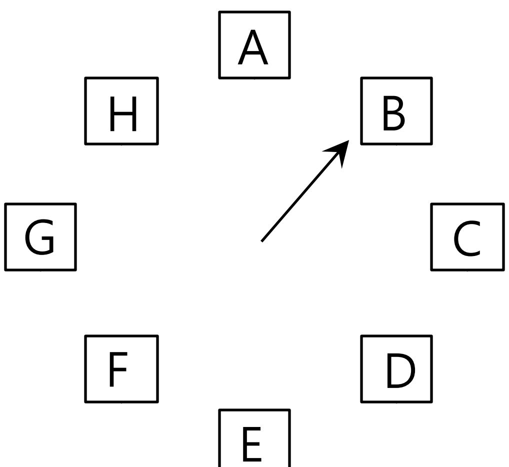
# Workload Example : The Looping Sequential

- 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.



# Approximating LRU: Clock Algorithm

- Require hardware support: **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 a page to begin with.
  - The algorithm continues until it finds a use bit that is set to 0.

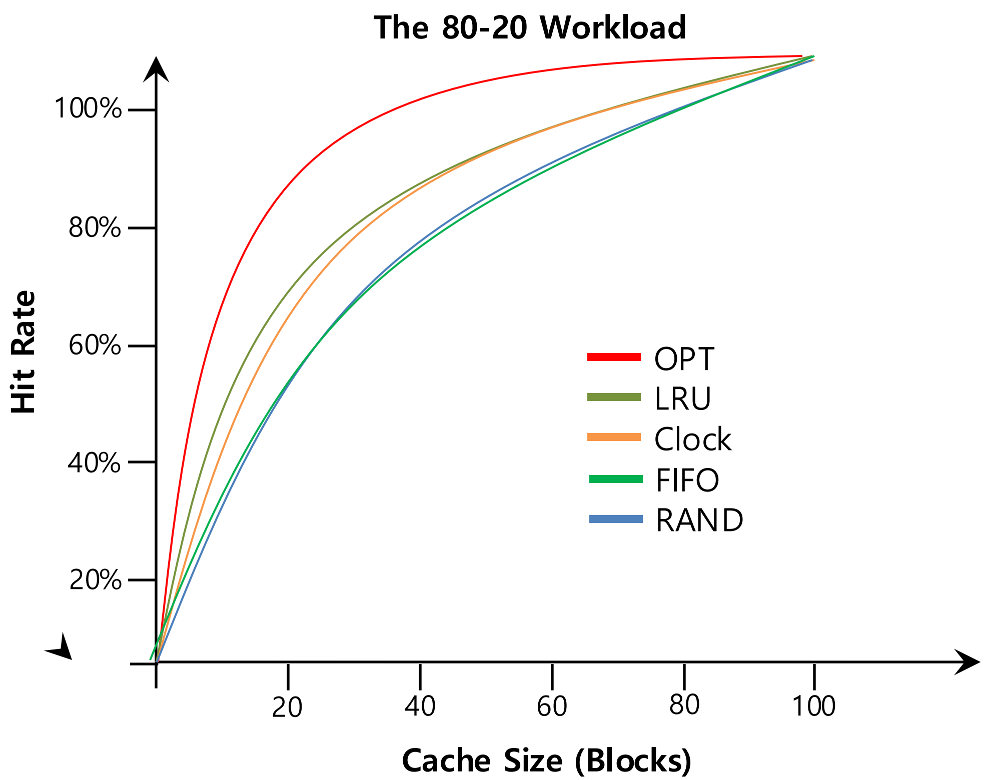



| Use bit | Meaning |
|---|---|
| 0 | Evict the page |
| 1 | Clear **use bit** and advance hand |

**The Clock page replacement algorithm**

# Workload with Clock Algorithm

- Clock algorithm doesn't do as well as LRU; but better than approaches that don't consider history at all

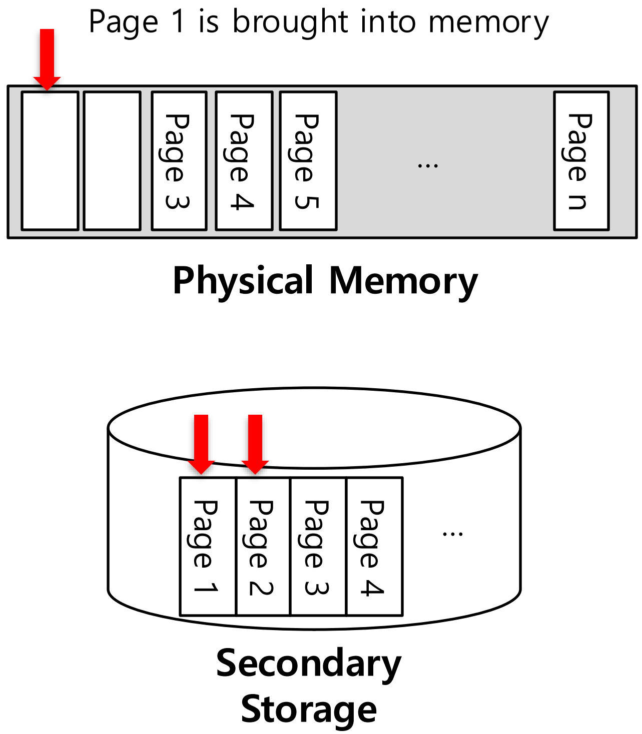


# Considering Dirty Pages

- The hardware includes 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.

# Prefetching

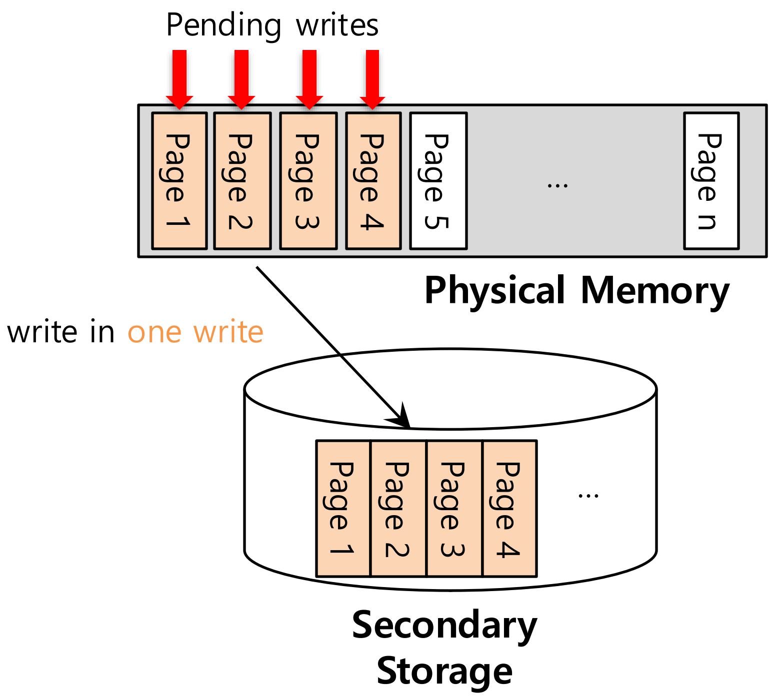
- The OS guesses that a page is about to be used, and thus bring it in ahead of time.




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

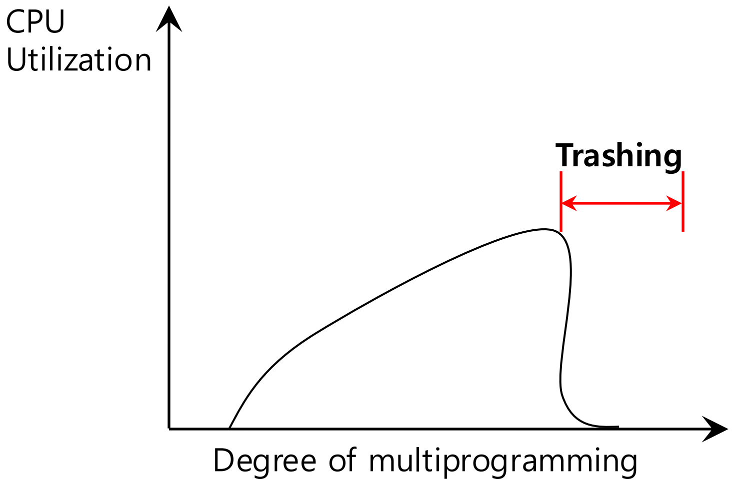
# Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - Perform a **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.



# Current Trends in Memory Management

- Less critical now
  - Memory is cheap → larger physical memory (well except for Apple's devices…)
- Larger page sizes
  - Better TLB coverage; smaller page tables
- Larger virtual address space
  - 64-bit address
- File I/O using the virtual memory system
  - Memory mapped I/O: mmap()

# Summary

- Swapping: use part of disk as memory
- Page replacement (page/swap out, page/swap in)
  - LRU, LFU, Random, FIFO
- Approximation to LRU: Clock
- Batching the disk IO
  - Clustering
  - Grouping
  - Prefetching