ITP 30002 Operating System

Swapping

OSTEP Chapters 21 and 22

Shin Hong

Beyond Physical Memory

- What if the amount of the allocated pages in running processes exceed the physical memory capacity?

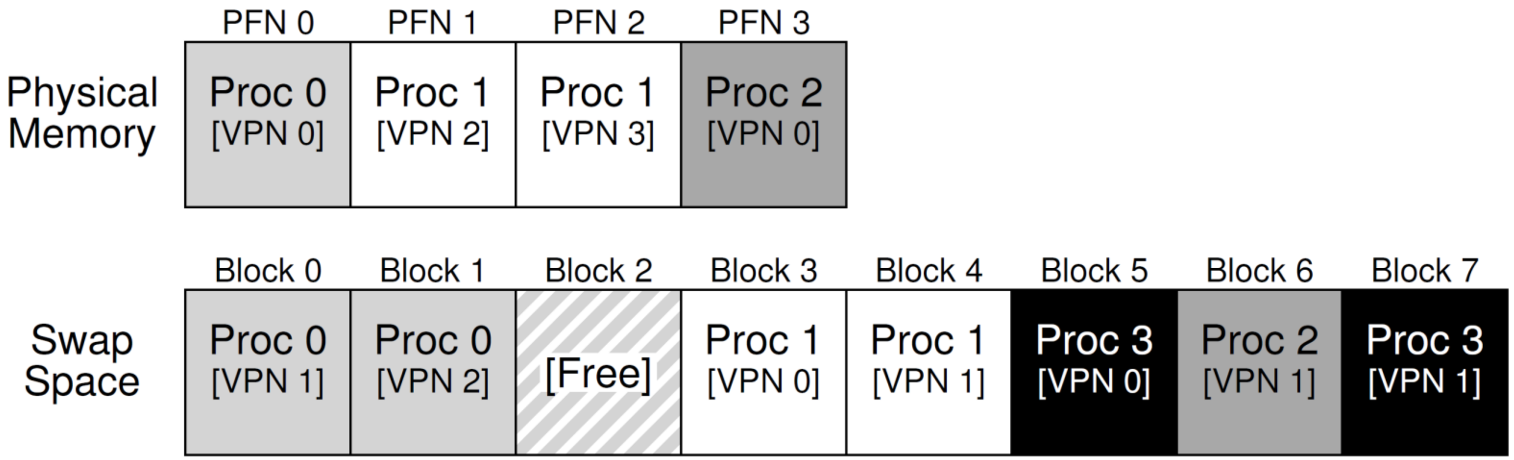
- OS stashes away some pages that are not in great demand at the moment
    - usually to a swap space in HDD or SSD

- OS brings a stored pages back to the main memory before a process accesses data on the page
    - by stashing other pages

Swap Space

- reserve **swap space** on the disk for moving pages back and forth
  - swap space is a collection of page-sized blocks
  - OS needs to store the location on a swap space where a page is stored

- Ex.



Page Fault

- A page table entry has the present bit to indicate whether the page is present in physical memory or stored in swap space

- A page fault is raised by an architecture if the present bit is off, so that the page-fault handler is invoked to serve the page fault
  - the location of the page in swap space is stored at a page table
  - the process will be blocked during the I/O for stashing and reloading

- Page fault and consequent page-in and page-out operations are all taken placed transparently to the process

Page Fault Control Flow Algorithm


```
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 Replacement

- Before page in a page, OS must page out some pages to make a ro om if the memory is almost fully used
  - most OS's have high and low watermarks for a page daemon to proactively st art page out in background

- OS must have a proper page replacement policy because unnecess ary page-in or page-out incurs significant runtime cost
  - A disk access takes 10000 to 100000 times longer than a memory access
  - Average Memory Access Time ($AMAT$) = T_{Mem} + $p_{Miss} \times T_{Disk}$
    - E.g., T_{Mem}=100 ns, T_{Disk}=10 ms, p_{Miss}=10% : $AMAT$ = 1.01 ms

Page Replacement Policies

- Policies
  - FIFO
  - Random
  - Optimal (ideal)
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)

- Simple cost model
  - There's only one process in the system, which uses all frames
  - A workload is represented as a sequence of VPN accesses
  - A page-out occurs only when the memory is full
  - The time cost of workload is measured as the total number of page-in and page-out operations with the workload

FIFO

- Page out the one that was paged in first (i.e., stayed longest time)
- Example: 3 frames, 4 pages (0 to 3)

0 1 2 0 1 3 0 3 1 1 2 1. 3

0	0	0	0	0	**3**	3	3	3	3	**2**	2	2
	1	1	1	1	1	**0**	0	0	0	0	0	**3**
		2	2	2	2	2	2	**1**	1	1	1	1

Random

- Pick a random page at a page replacement decision
- Example: 3 frames, 4 pages (0 to 3)

0	1	2	0	1	3	0	3	1	1	2	1.	3
0	0	0	0	0	0	0	0	0	0	**2**	2	2
	1	1	1	1	**3**	3	3	3	3	3	3	3
		2	2	2	2	2	2	**1**	1	1	1	1

Optimal

- Select a page whose next use is furthest in future
- Example: 3 frames, 4 pages (0 to 3)

0	1	2	0	1	3	0	3	1	1	2	1	3
0	0	0	0	0	0	0	0	0	0	2	2	2
	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	3	3	3	3	3	3	3	3

Least Frequently Used (LFU)

- Select a page which has been used least
- Example: 3 frames, 4 pages (0 to 3)

0 1 2 0 1 3 0 3 1 1 2 1 3

0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	**3**	3	3	3	3	**2**	2	**3**

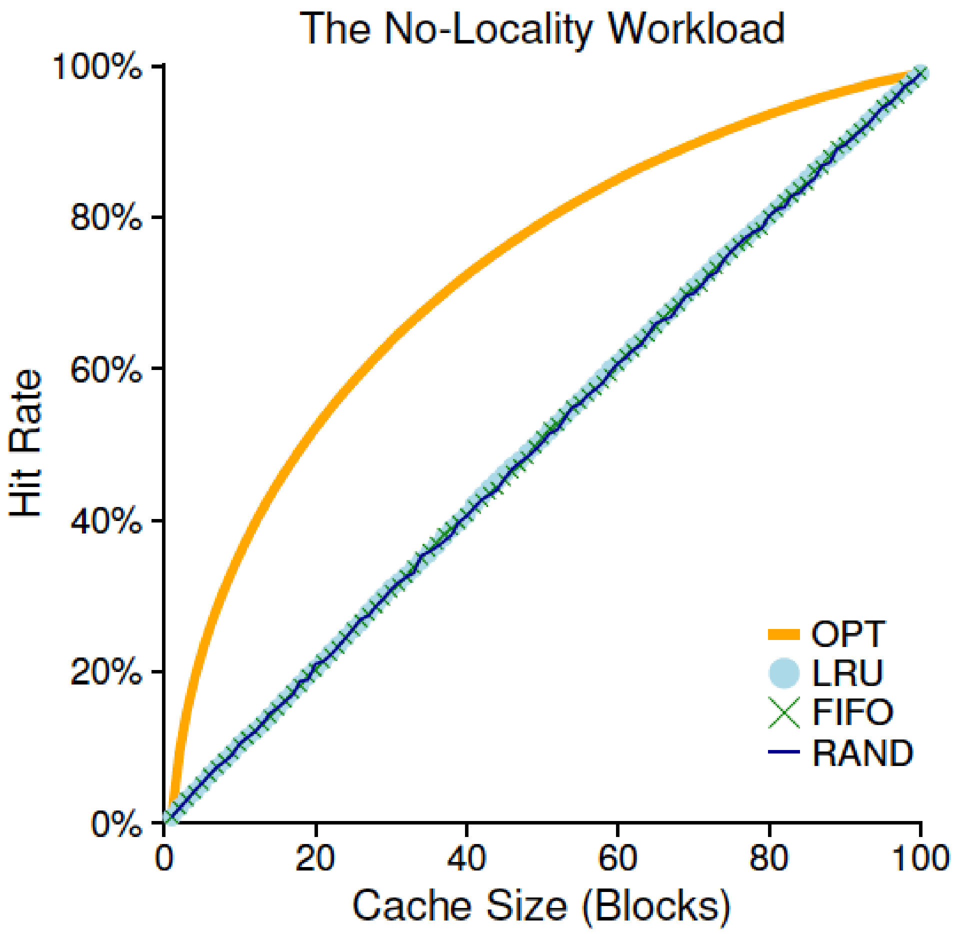
Least Recently Used (LRU)

- Select a page whose last use is furthest in past
- Example: 3 frames, 4 pages (0 to 3)

0	1	2	0	1	3	0	3	1	1	2	1	3
0	0	0	0	0	0	0	0	0	0	**2**	2	2
	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	**3**	3	3	3	3	3	3	3

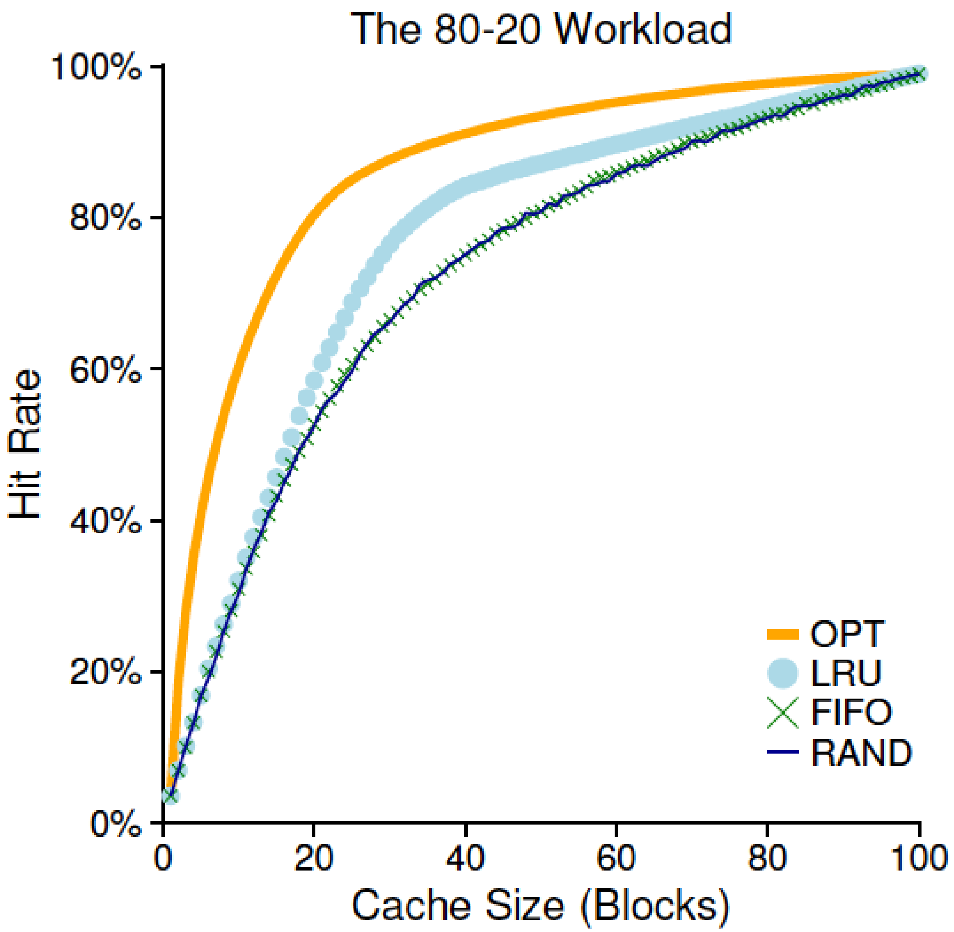
Workload Example – No Locality

- A random workload with 10000 accesses on 100 pages



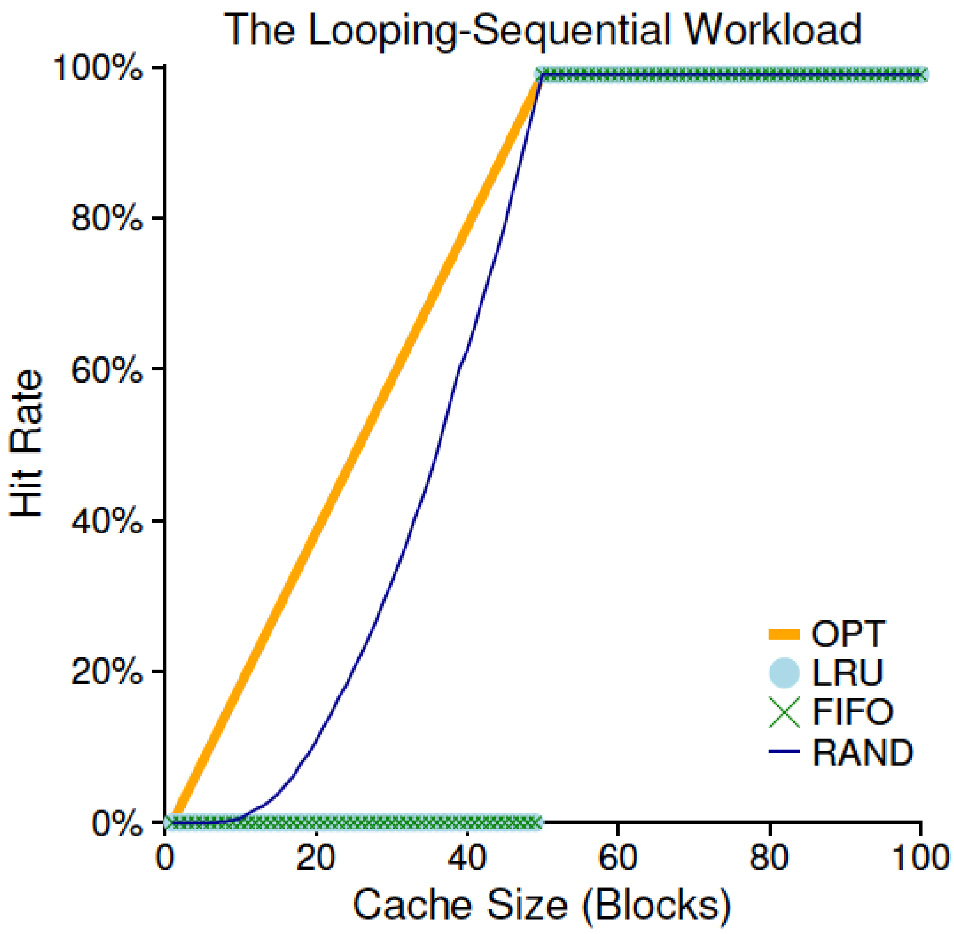
Workload Example – 80:20 Workload

- 10000 accesses on 100 pages where 80% of accesses are on a 20% of the pages



Workload Example – Looping Sequential

- 10000 accesses on 50 pages
- repeating sequential accesses on 0, 1, 2, …, 49 for 200 times

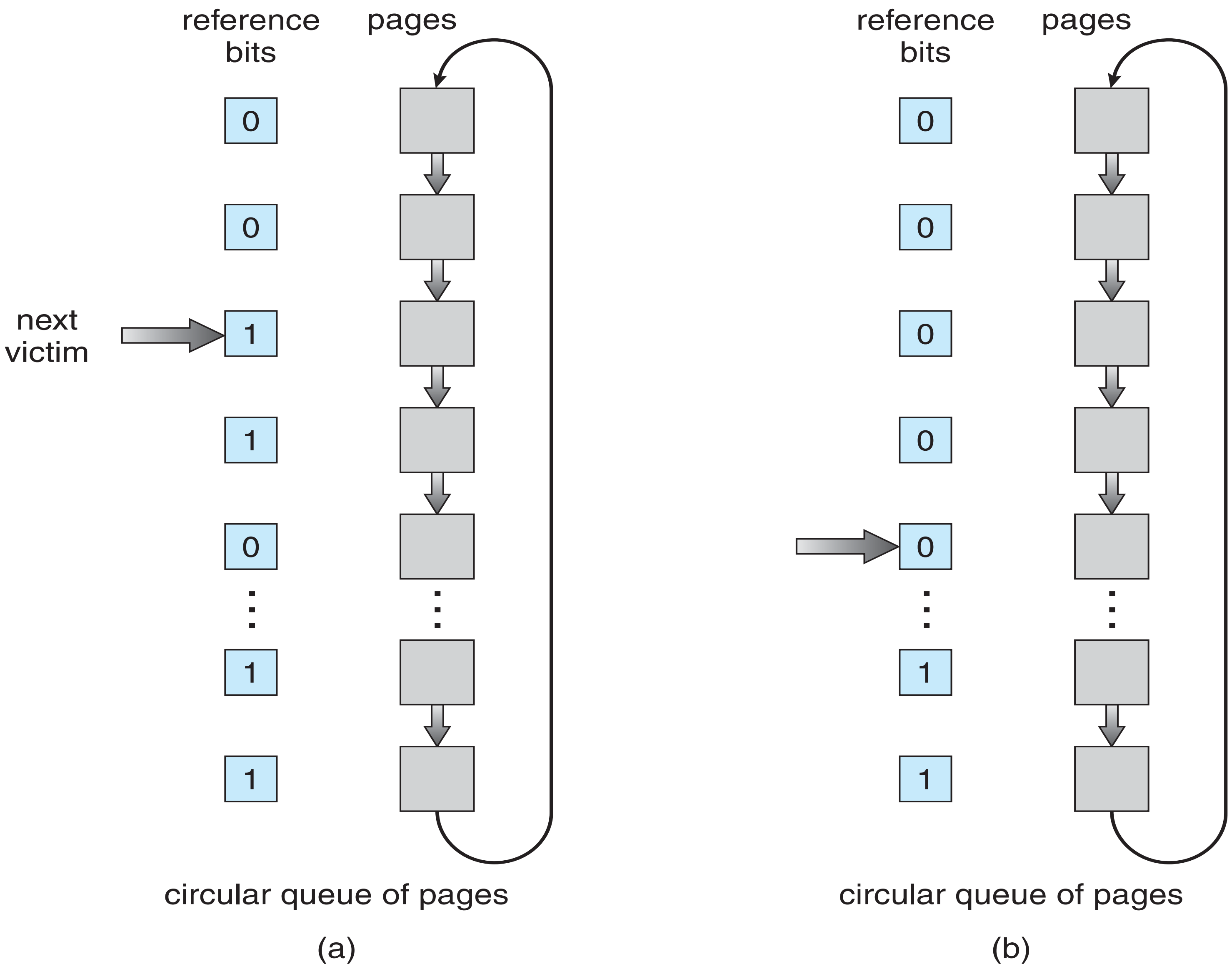


Approximating LRU

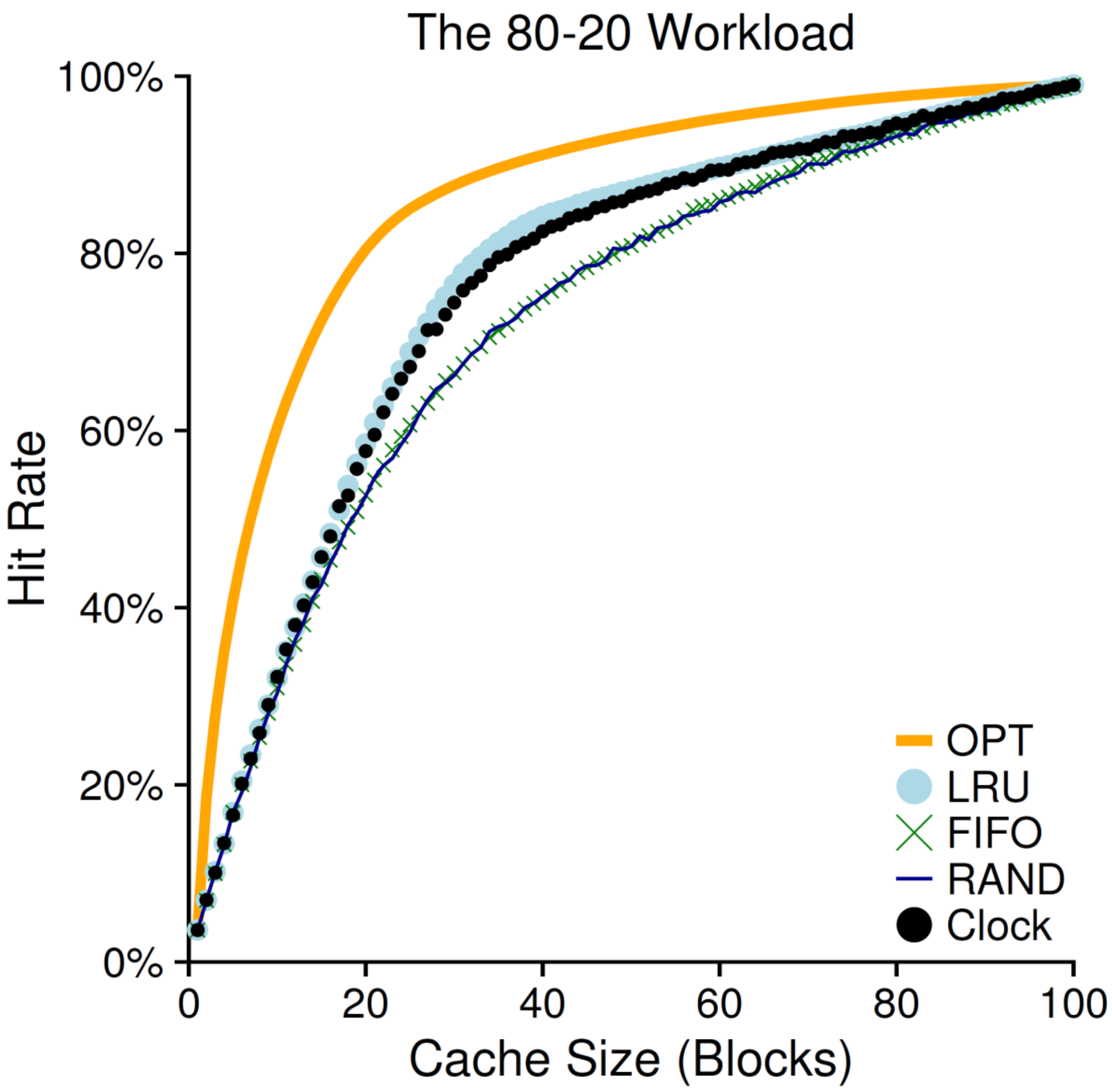
- An exact implementation of LRU is expensive
  - update last access time of a page at every memory access

- Clock (second-chance) algorithm using reference bit
  - a reference bit (use bit) is given to each page table entry, such that the reference bit is set to 1 by the hardware at a memory access
  - pages are maintained in a circular list with one clock hand
  - the clock hand iterates over the list to choose the first node with the reference bit off
    - the clock hand turns off the reference bit when it passes over the node with the reference bit on.

Clock Algorithm Example



Clock Algorithm Performance



Considering Modified Bit

- A page table entry has a modified bit (dirty bit) which is turned on by a hardware if the page is ever written

- It is more efficient to evict a clean page than a modified page, because there is no need to write back (page out) the content to the storage
  - (reference bit, modified bit)
    - (0, 0) : best page to replace
    - (0, 1) : not quite as good, must write out before replacement
    - (1, 0) : probably will be used again soon
    - (1, 1) : probably will be used again soon and need to write out

Other Virtual Memory Policies for Efficiency

- **Demand paging**: brings the page into memory when it is actually accessed

- **Prefetching**: brings the page ahead of its use time if it is likely to be used soon

- **Clustering**: collect a number of dirty pages and write them once to the storage

Thrashing

- A system is in **thrashing** when it is trapped by serial pagings for the working sets of the processes exceed an available number of frames
  - page out an existing page, but quickly the page gets paged in again
  - frequent swapping results low CPU utilization, which leads the system to increase the degree of multiprogramming
- A system resolves thrashing by running only a subset of the processes or killing some of the running processes