

OSTEP

Virtual Memory

Questions answered in this lecture:

- How to run process when not enough physical memory?
- When should a page be moved from disk to memory?
- What page in memory should be replaced?
- How can the LRU page be approximated efficiently?

Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

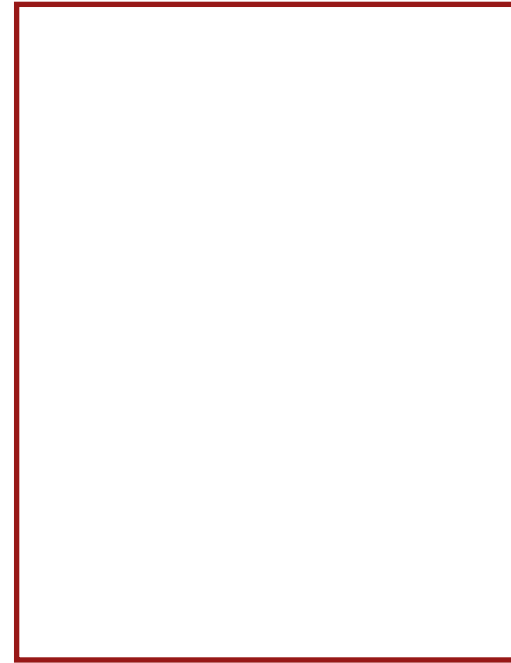
User code should be independent of amount of physical memory

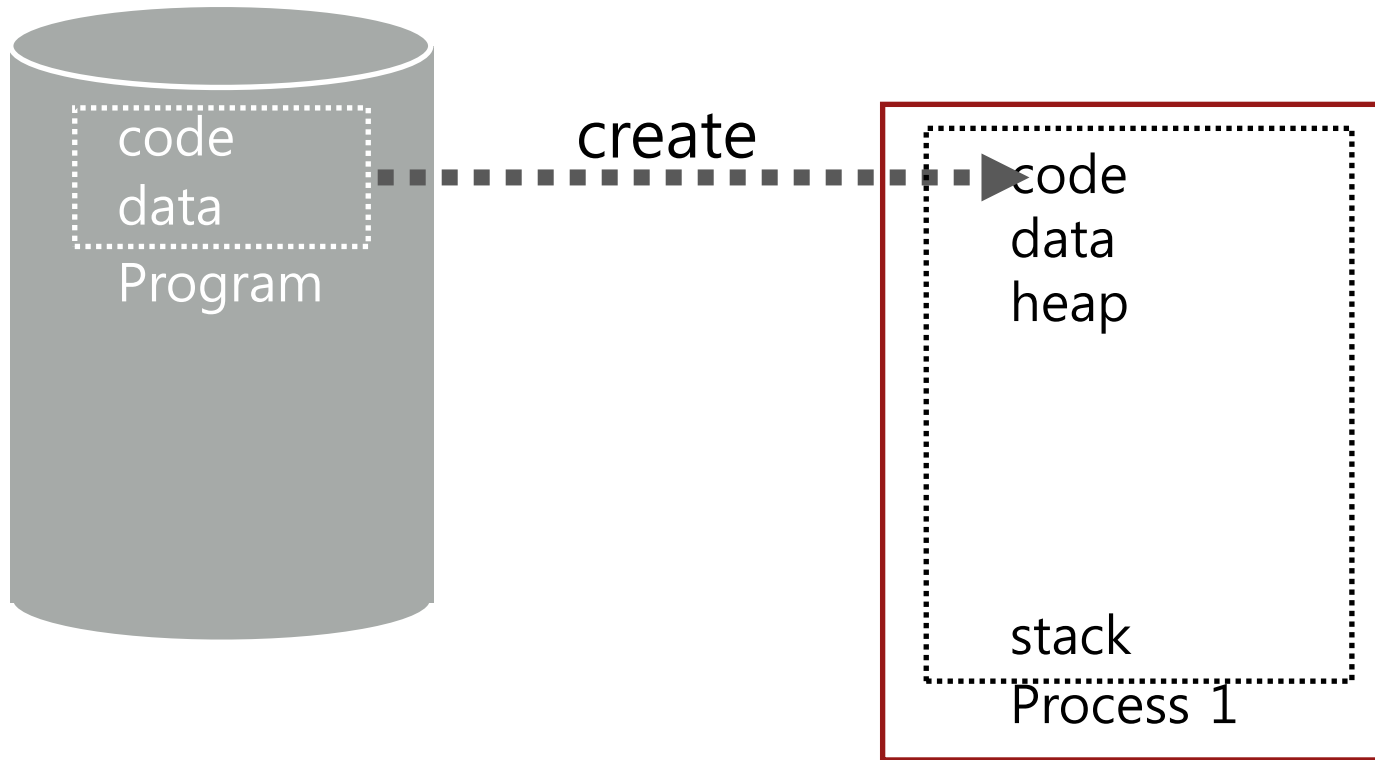
- Correctness, if not performance

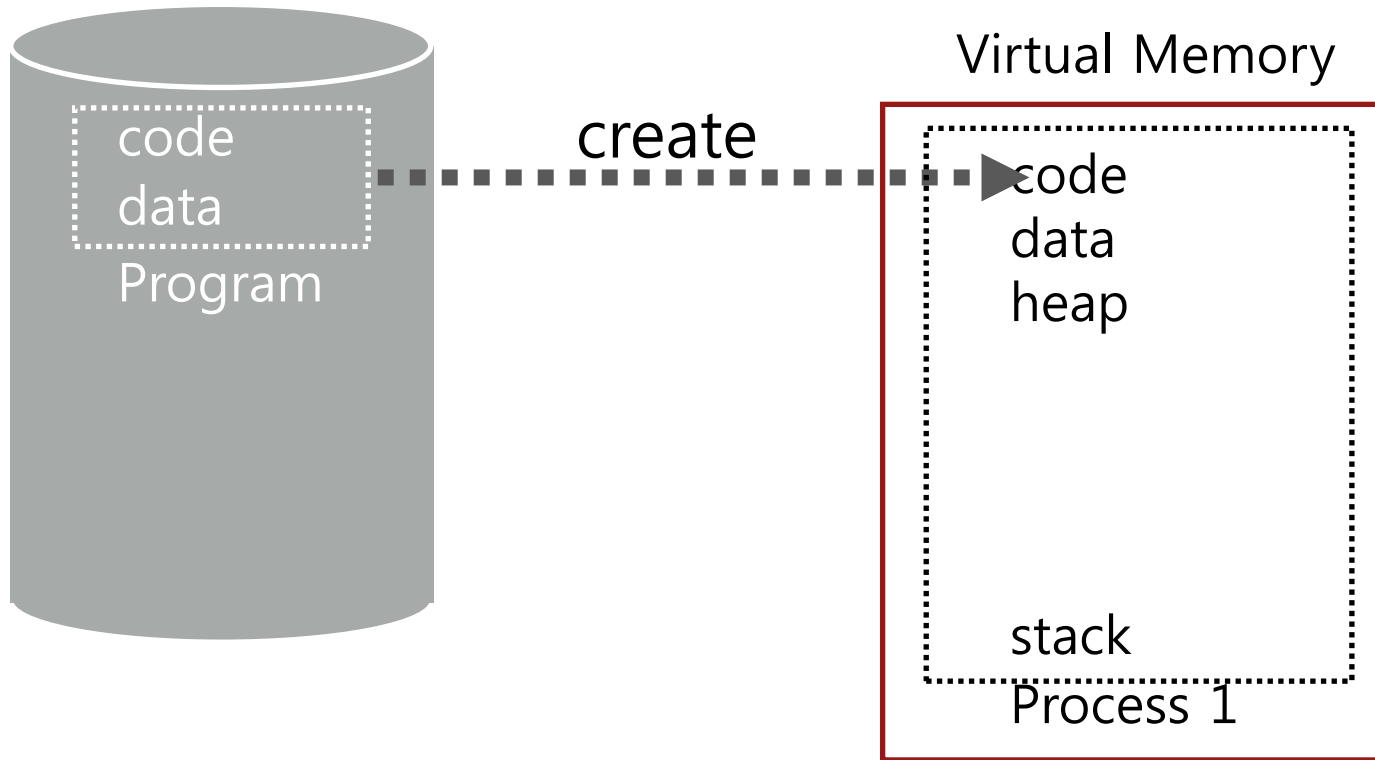
Virtual memory: OS provides illusion of more physical memory

Why does this work?

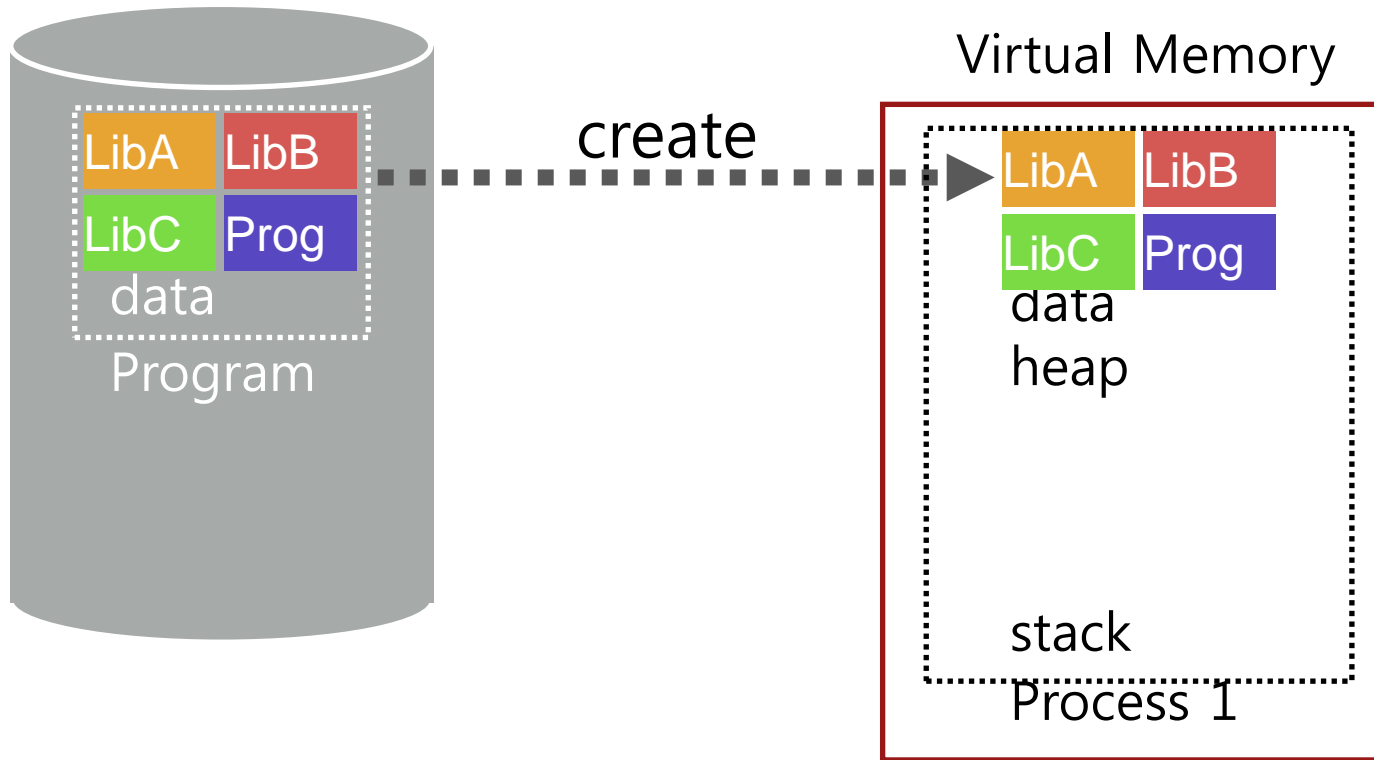
- Relies on key properties of user processes (workload) and machine architecture (hardware)





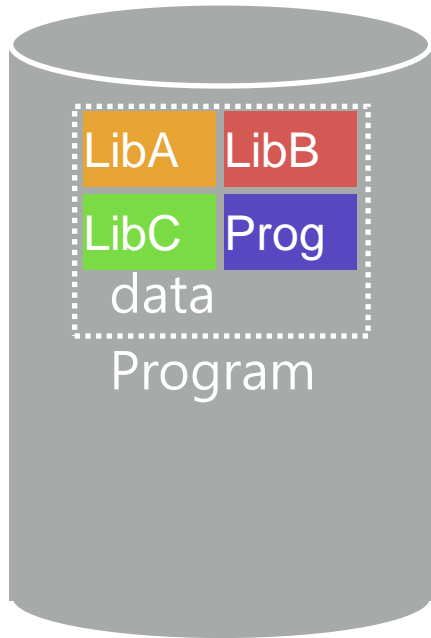


what's in code?



many large libraries, some
of which are rarely/never used

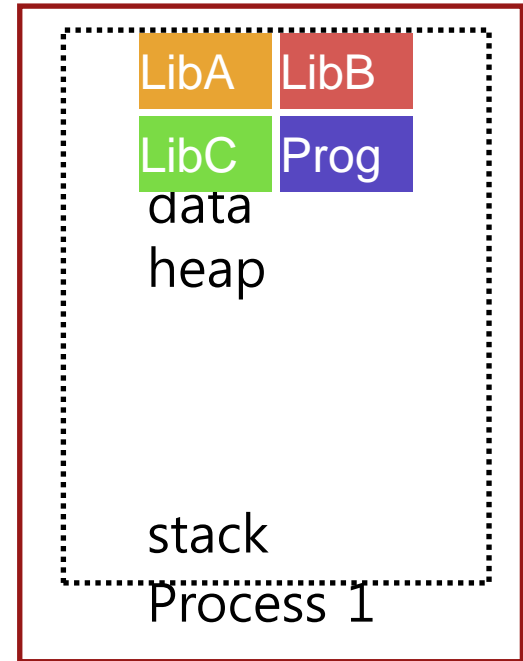
How to avoid wasting **physical pages** to back
rarely used **virtual pages**?

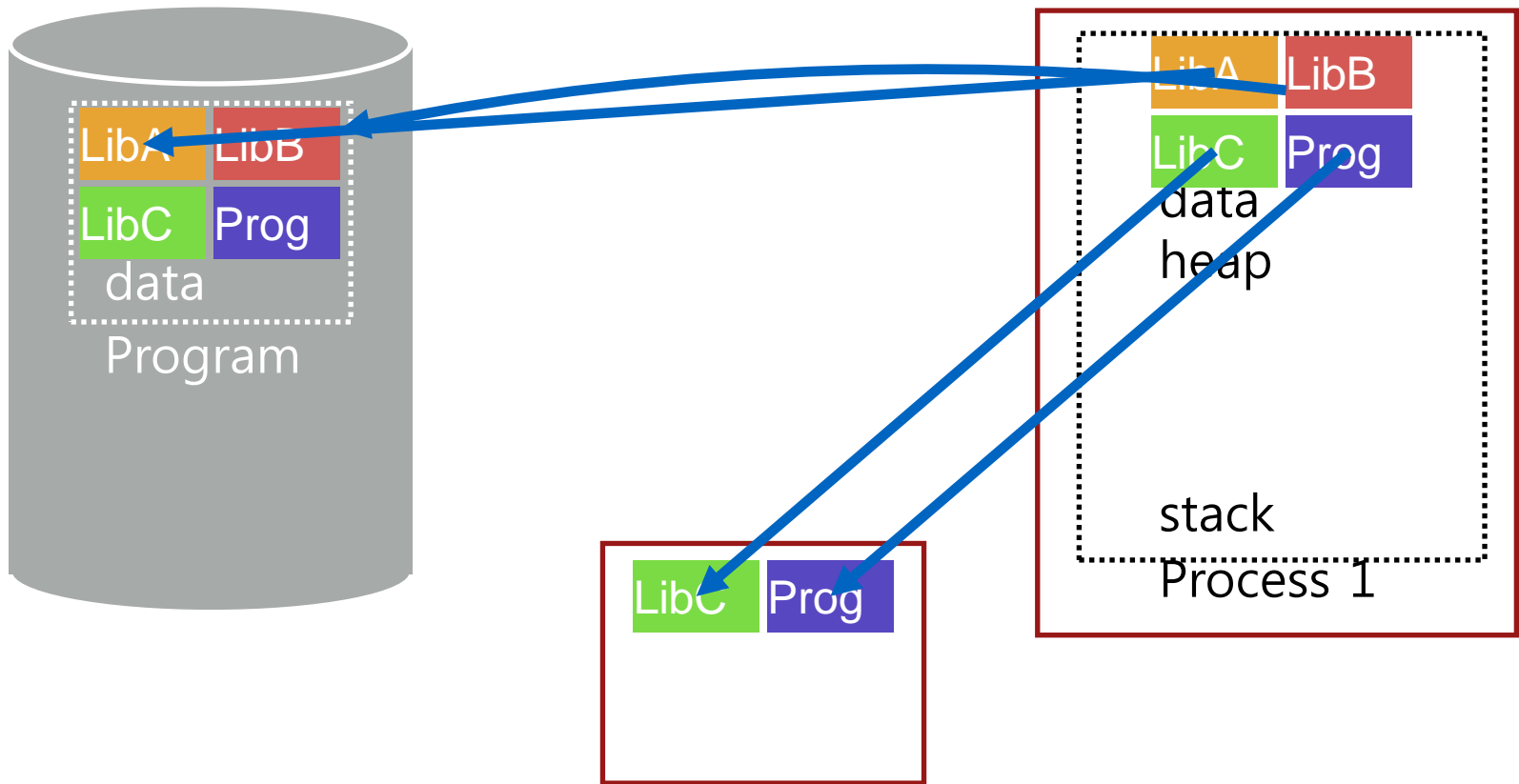


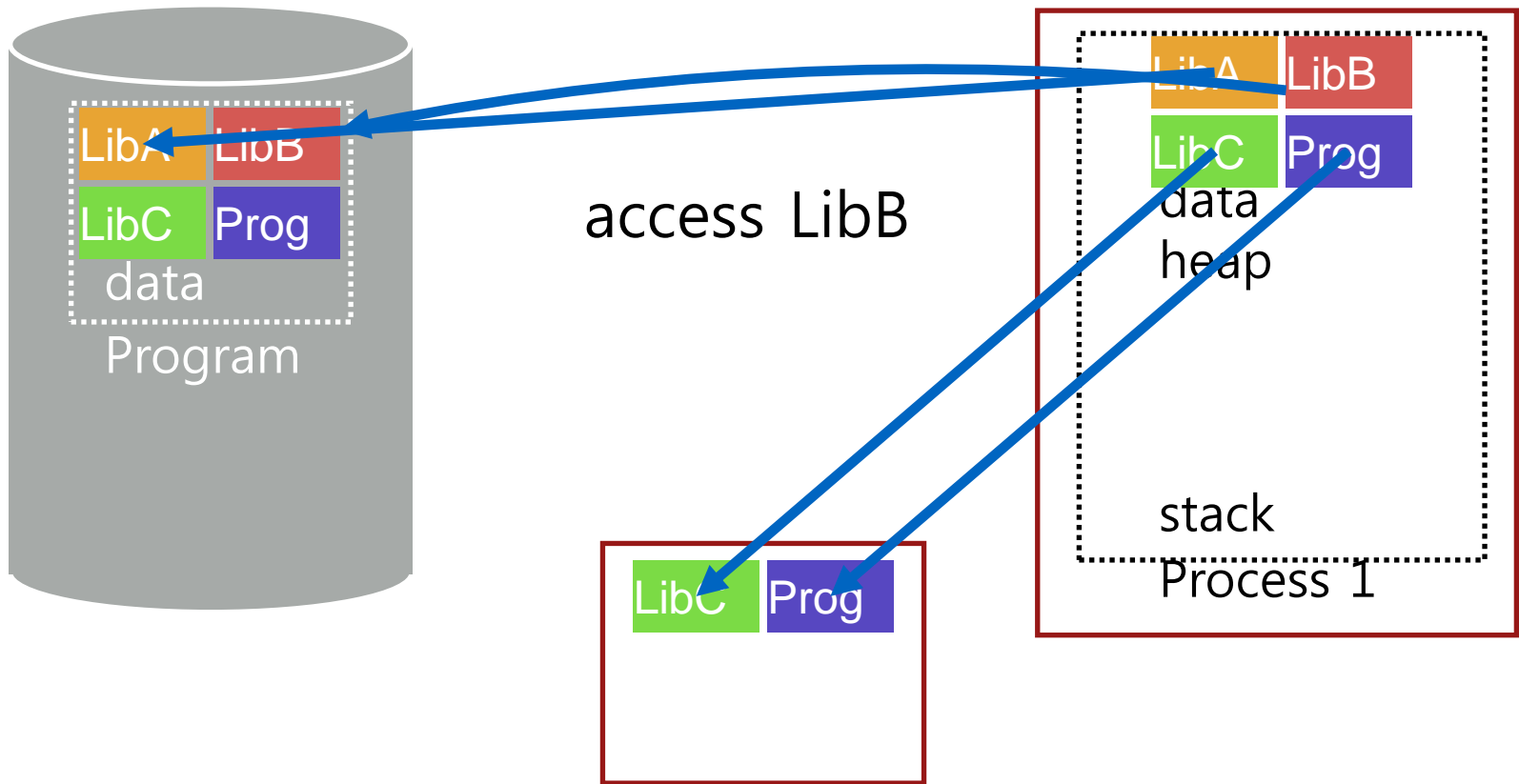
Phys Memory

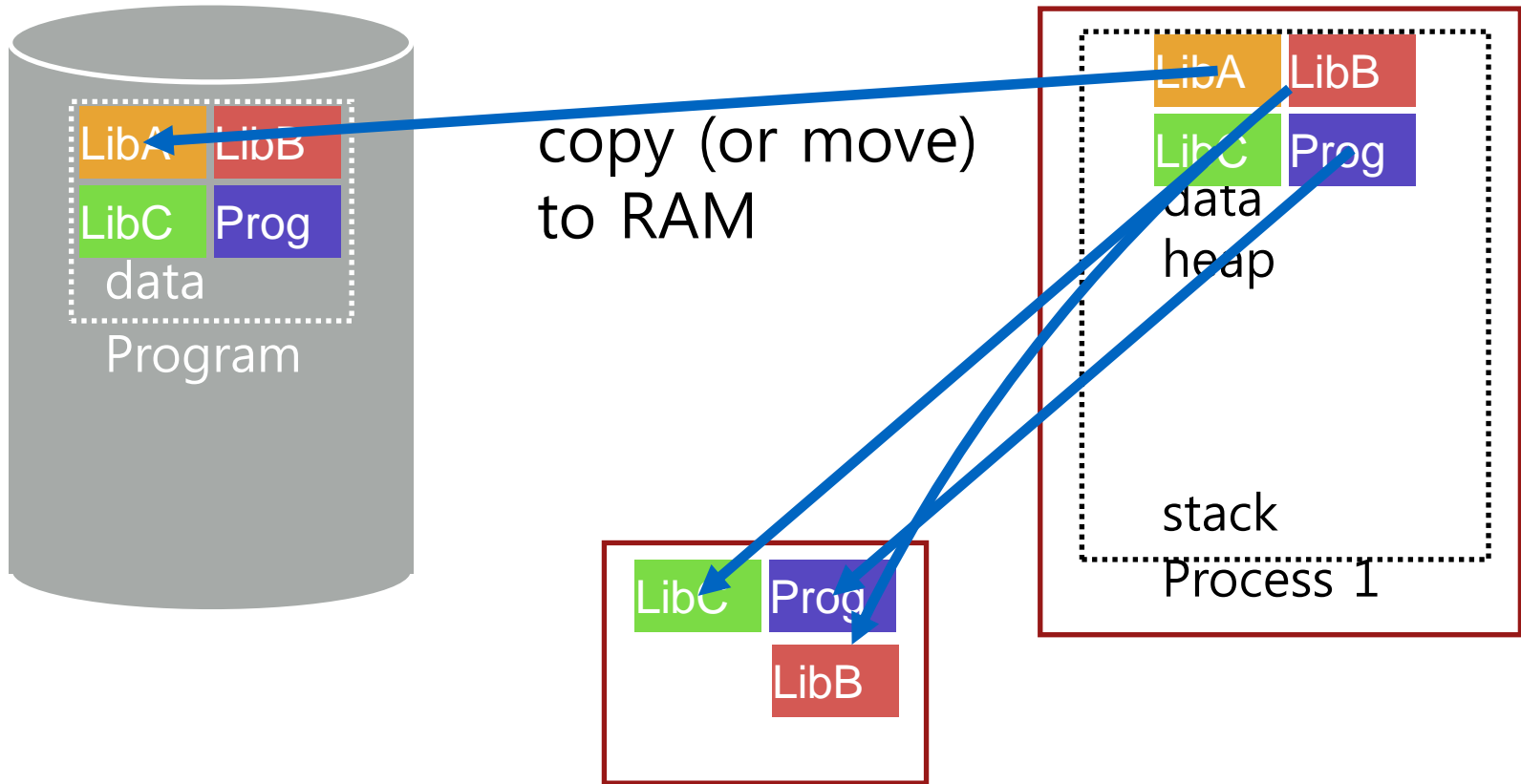


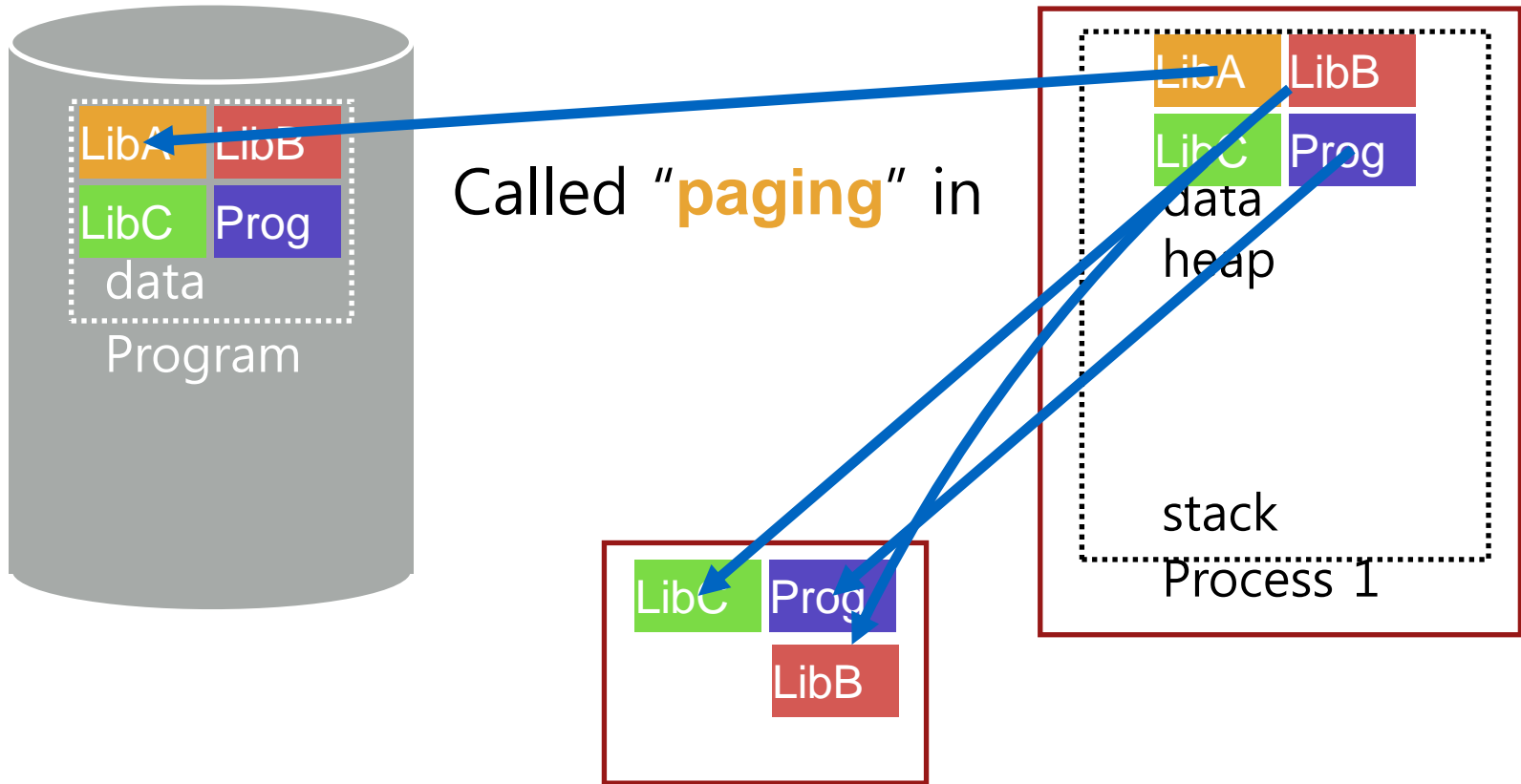
Virtual Memory











Locality of Reference

Leverage **locality of reference** within processes

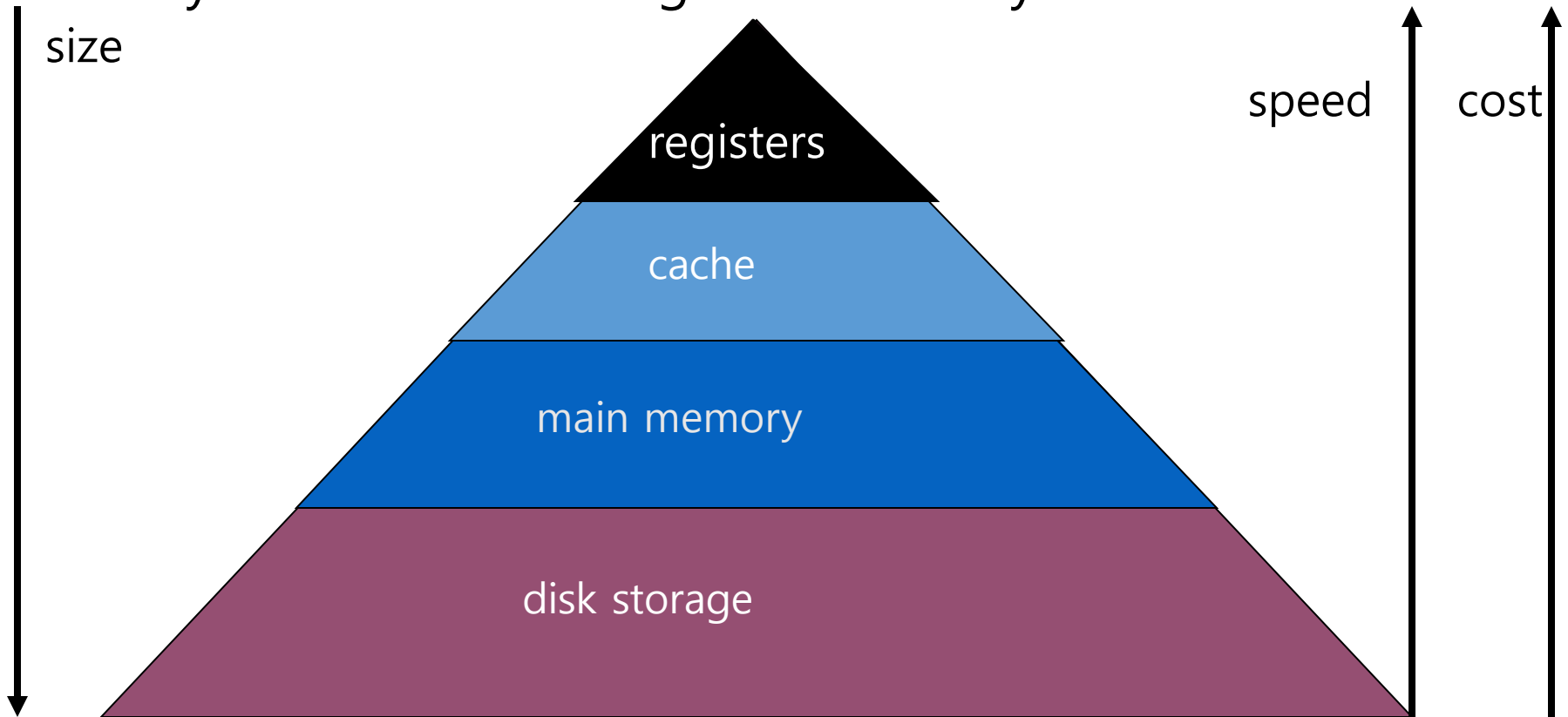
- **Spatial**: reference memory addresses **near** previously referenced addresses
- **Temporal**: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
 - Estimate: 90% of time in 10% of code

Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage **memory hierarchy** of machine architecture
Each layer acts as “backing store” for layer above



- 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**

Virtual Memory Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

- Same behavior as if all of address space in main memory
- Hopefully have similar performance

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

Virtual Address Space Mechanisms

Each page in virtual address space maps to one of three locations:

- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present

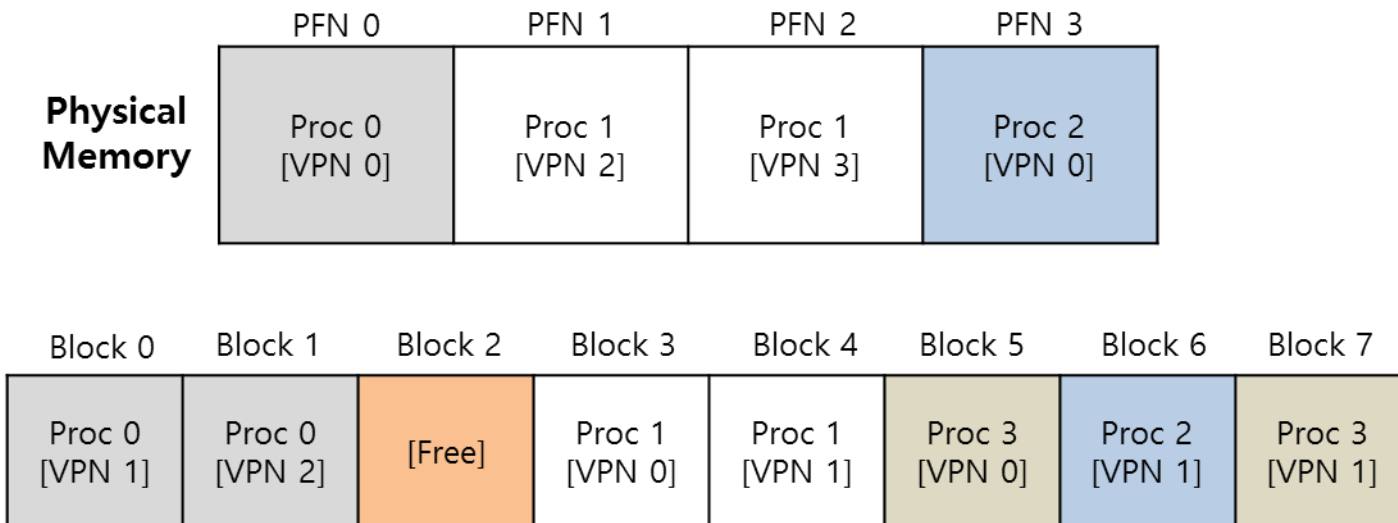
- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
 - PTE points to block on disk
 - Causes trap into OS when page is referenced
 - **Trap: page fault**

Single large address for a process

- Always need to first arrange for the code or data to be in memory 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**



Physical Memory and Swap Space

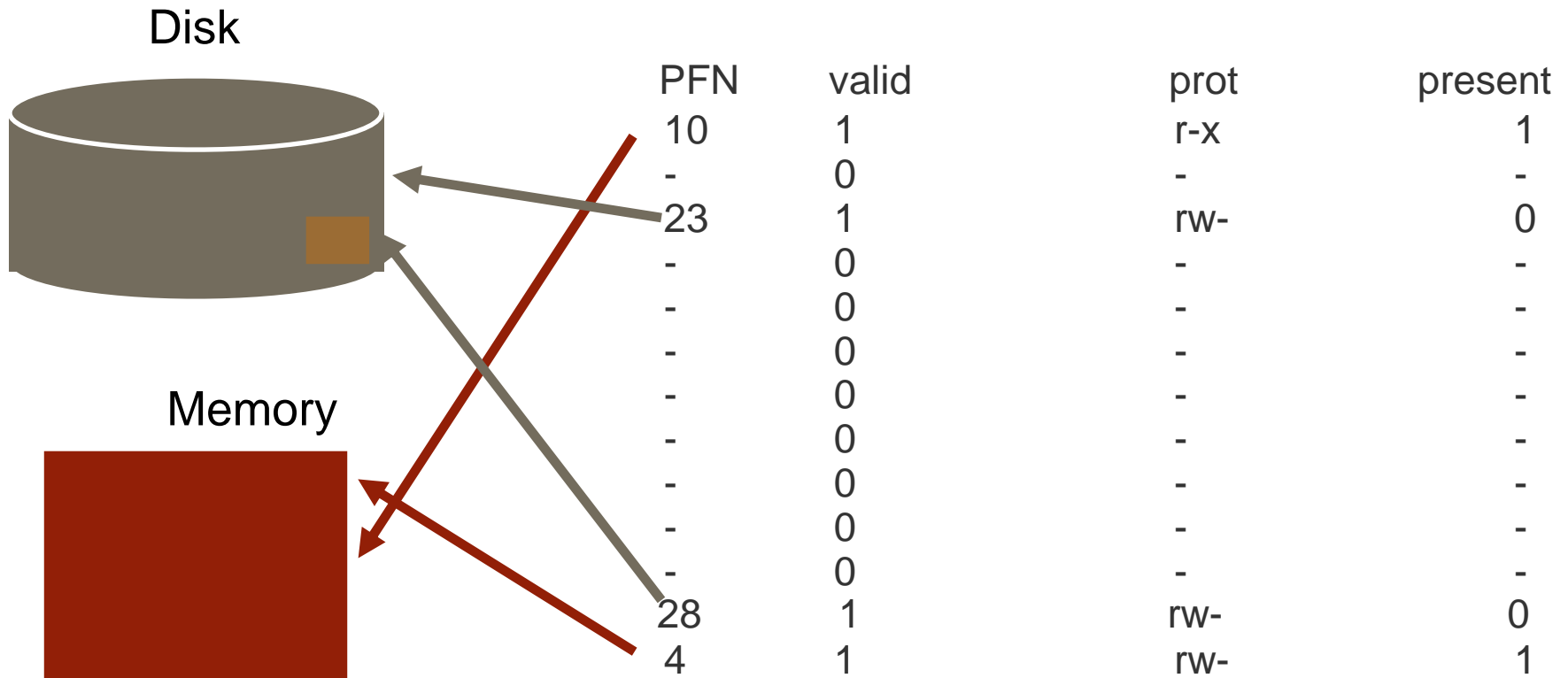
Present Bit

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.

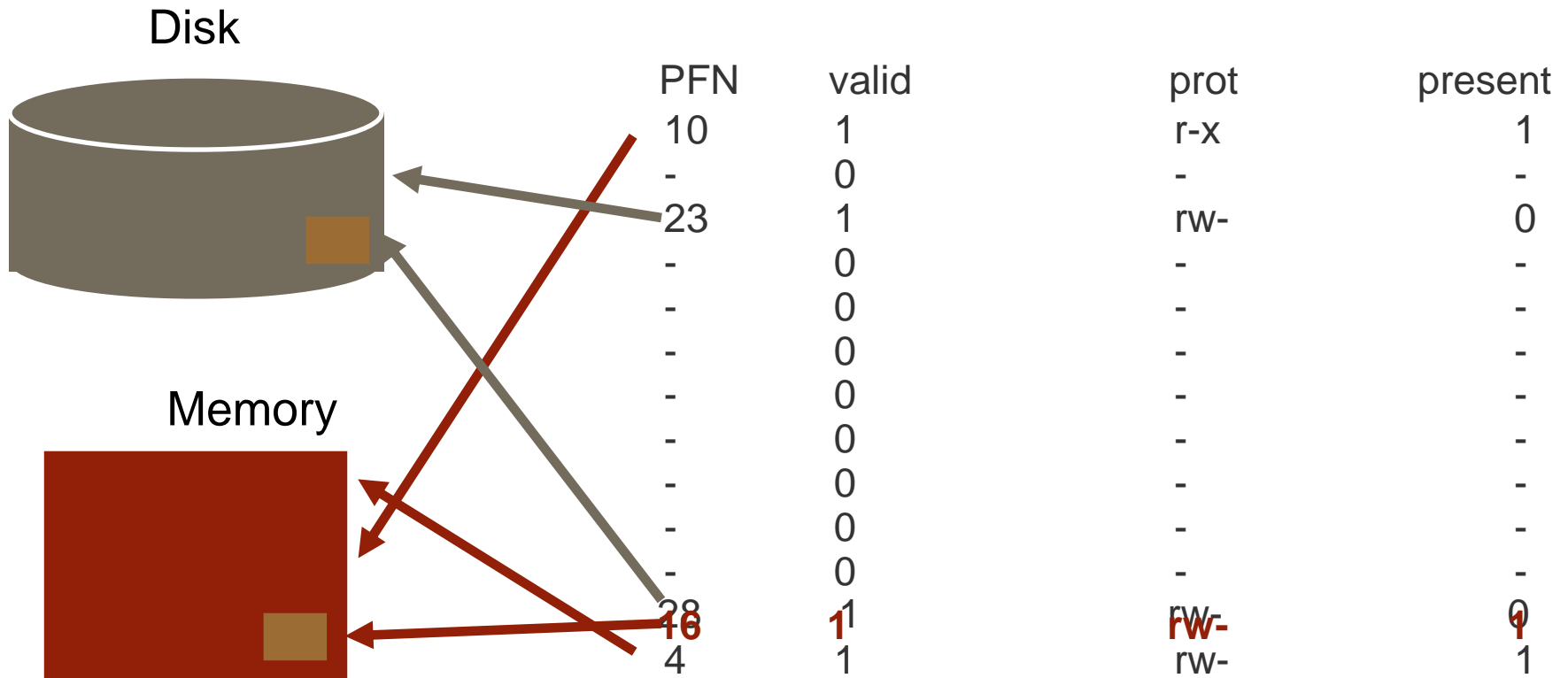
Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

Present Bit(Cont.)



What if access vpn 0xb?

Present Bit(Cont.)



What if access vpn 0xb?

What if no RAM is left?

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

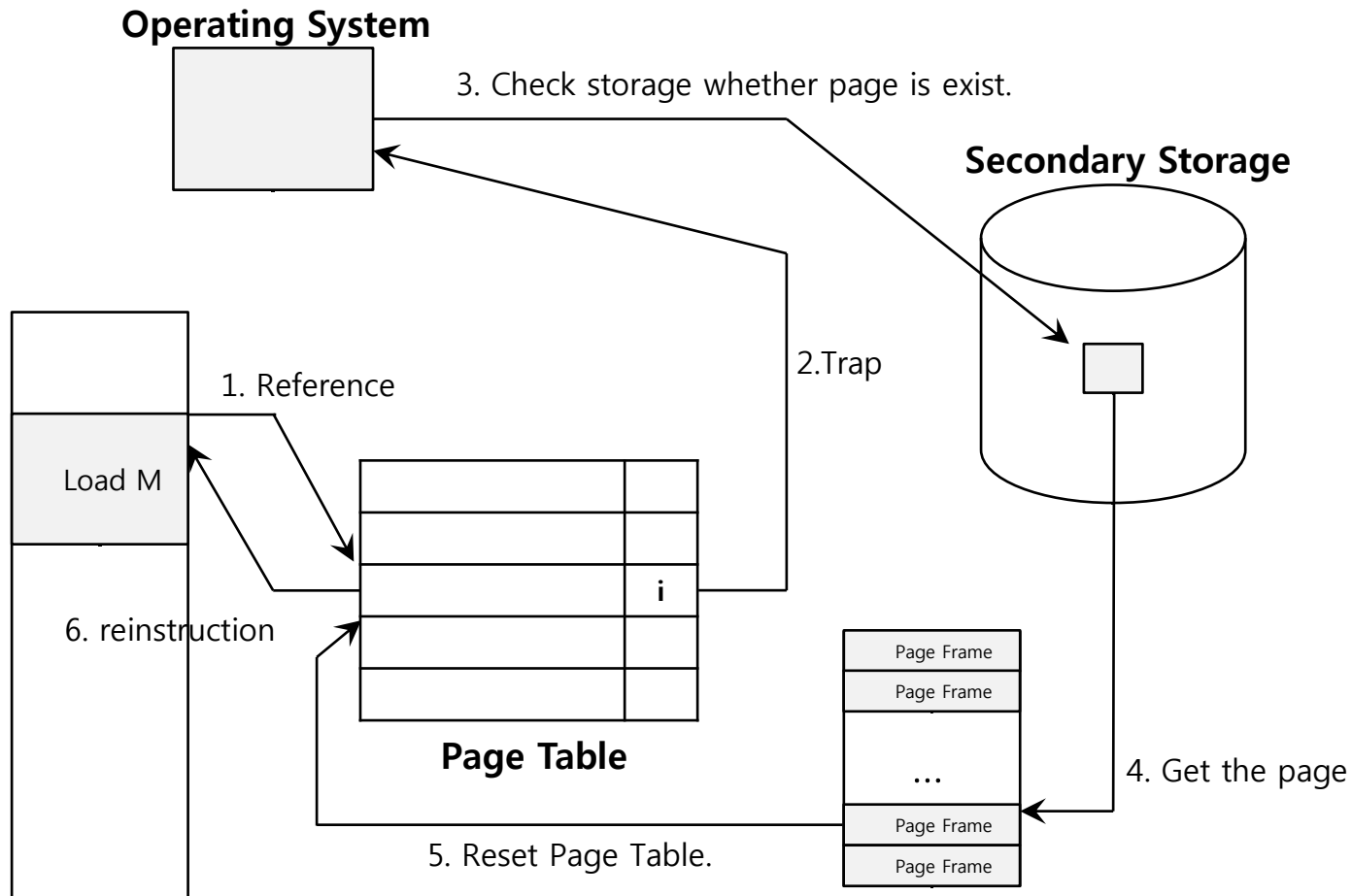
The Page Fault

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 address.



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

Page Fault Control Flow – Hardware

```
1:     VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:     (Success, TlbEntry) = TLB_Lookup(VPN)
3:     if (Success == True) // TLB Hit
4:         if (CanAccess(TlbEntry.ProtectBits) == True)
5:             Offset = VirtualAddress & OFFSET_MASK
6:             PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:             Register = AccessMemory(PhysAddr)
8:         else RaiseException(PROTECTION_FAULT)
```


Page Fault Control Flow – Hardware

```
9:         else // TLB Miss
10:         PTEAddr = PTBR + (VPN * sizeof(PTE))
11:         PTE = AccessMemory(PTEAddr)
12:         if (PTE.Valid == False)
13:             RaiseException(SEGMENTATION_FAULT)
14:         else
15:             if (CanAccess(PTE.ProtectBits) == False)
16:                 RaiseException(PROTECTION_FAULT)
17:             else if (PTE.Present == True)
18:                 // assuming hardware-managed TLB
19:                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
20:                 RetryInstruction()
21:             else if (PTE.Present == False)
22:                 RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

```
1:      PFN = FindFreePhysicalPage()
2:      if (PFN == -1) // no free page found
3:          PFN = EvictPage() // run replacement algorithm
4:          DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5:          PTE.present = True // update page table with present
6:          PTE.PFN = PFN // bit and translation (PFN)
7:          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.

Swapping: Policies

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)*.

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

Argument	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 furthest in the future
- 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 $\frac{\text{Hits}}{\text{Hits}+\text{Misses}} = 54.6\%$

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 "**First-in**" 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		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

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

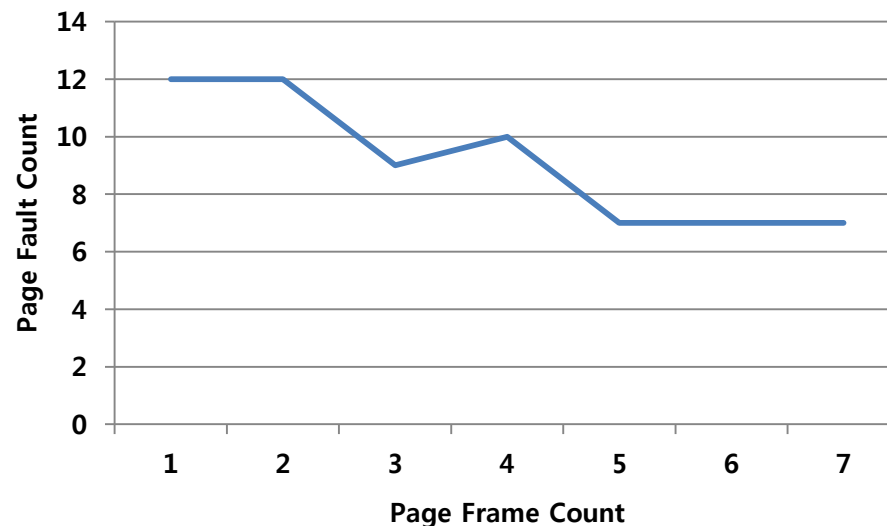
Even though page 0 had been accessed a number of times, **FIFO still kicks it out.**

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
---	---	---	---	---	---	---	---	---	---	---	---



Another Simple Policy: Random

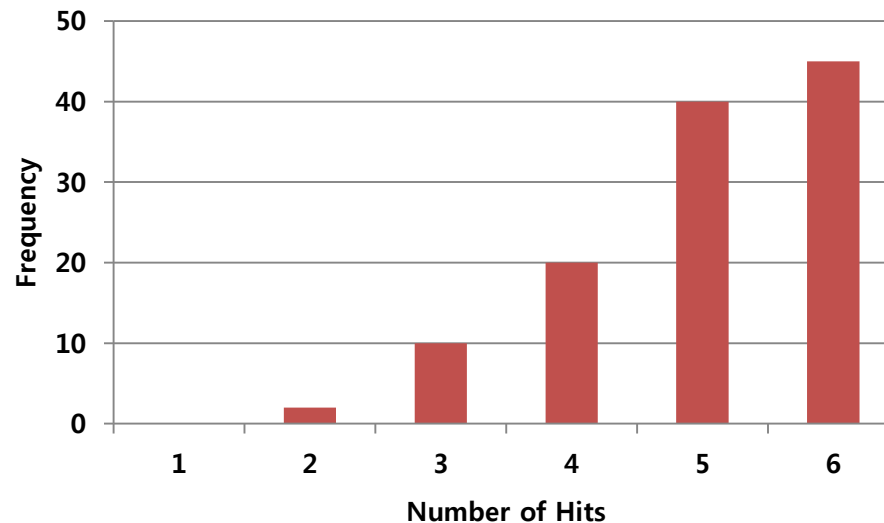
Picks a random page to replace under memory pressure.

- 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

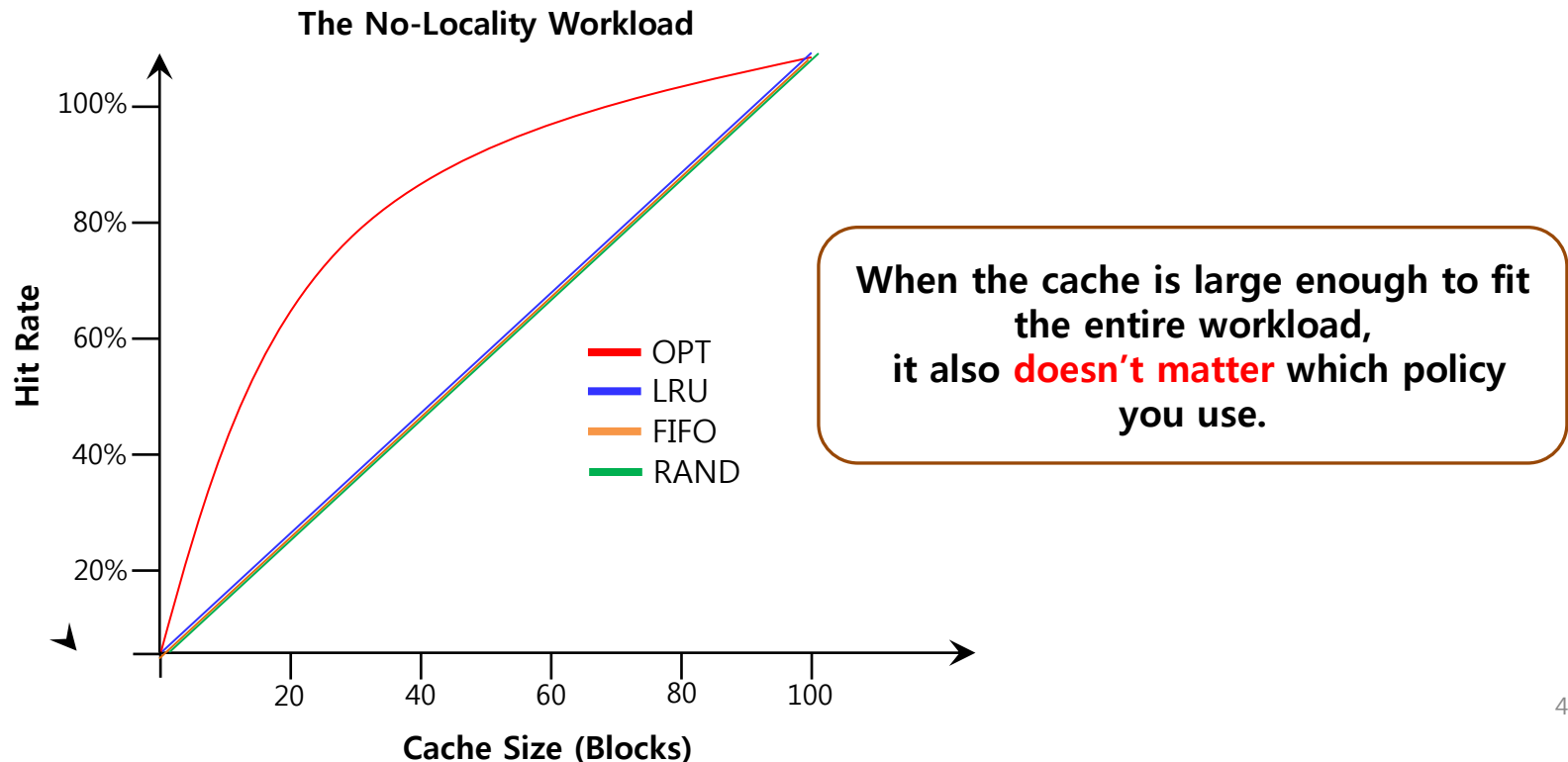
0 1 2 0 1 3 0 3 1 2 1

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.

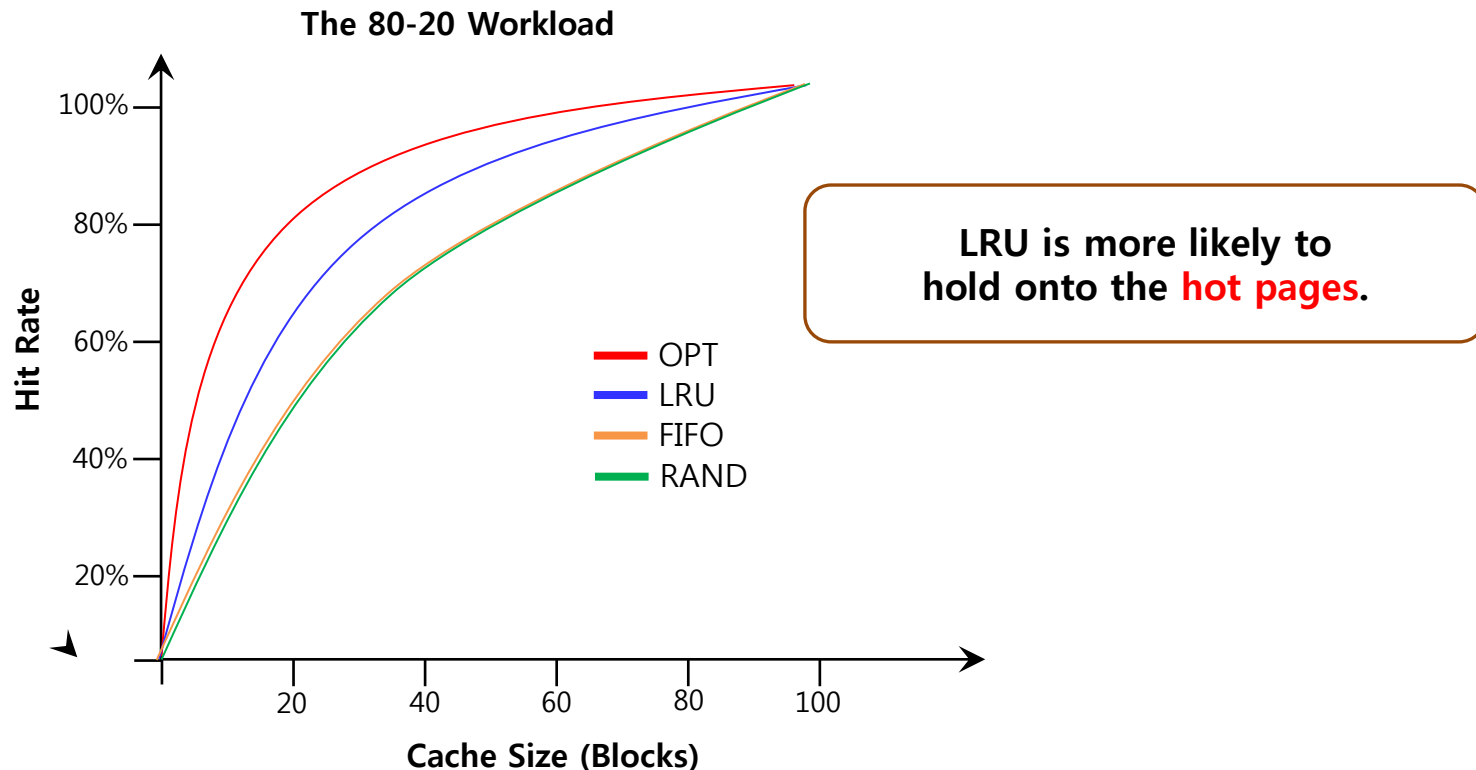
- Workload accesses 100 unique pages over time.
- Choosing the next page to refer to at random



Workload Example : The 80-20 Workload

Exhibits locality: 80% of the **reference** are made to 20% of the page

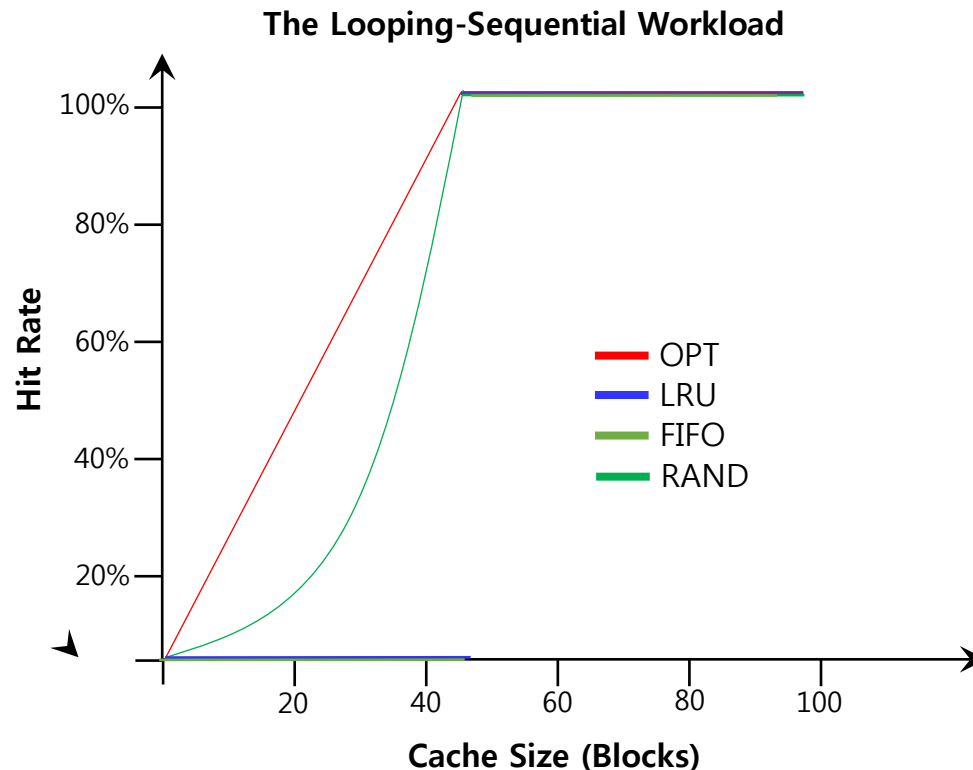
The remaining 20% of the **reference** are made to the remaining 80% of the pages.



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**

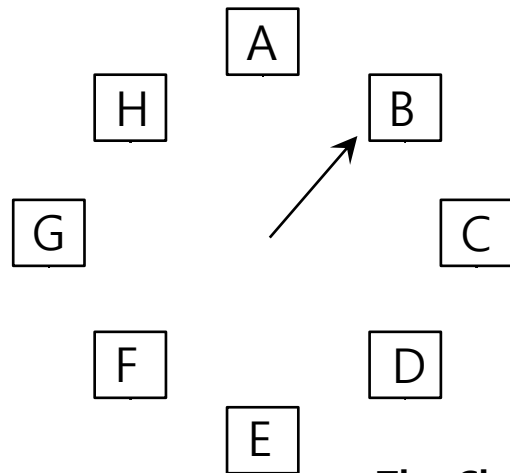
- Whenever a **page is referenced**, the use bit is set by hardware to 1.
- Hardware **never** clears the bit, though; that is the responsibility of the OS

Clock Algorithm

- All pages of the system arranged 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 Use bit and advance hand

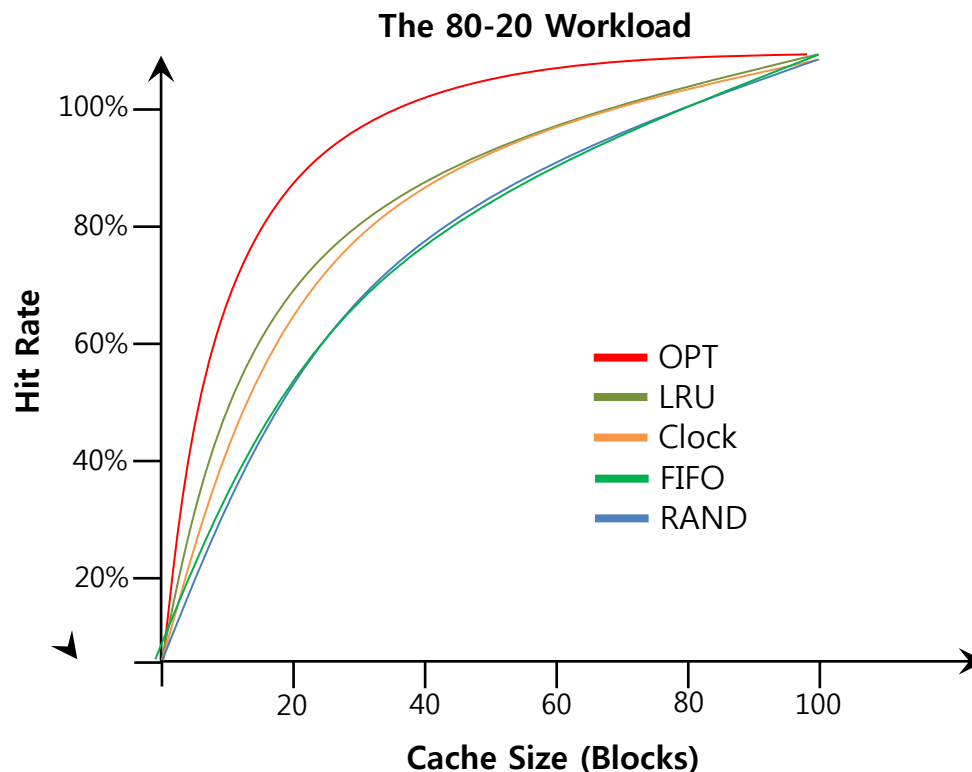
The Clock page replacement algorithm

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 than approach that don't consider history at all.



Considering Dirty Pages

The hardware include a **modified bit** (a.k.a **dirty bit**)

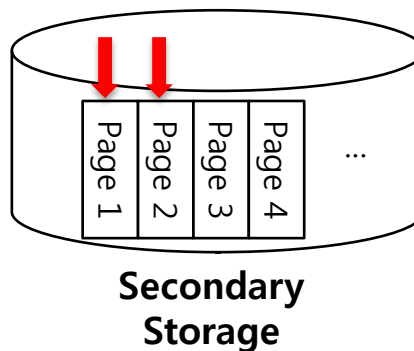
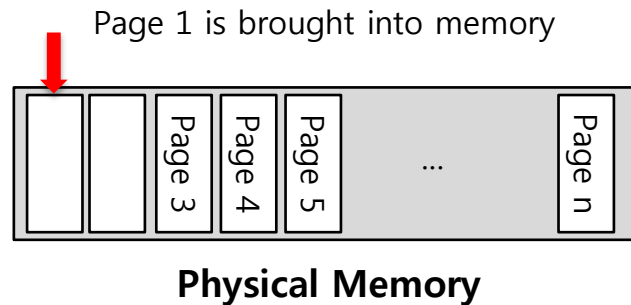
- Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it.
- Page has not been modified, the eviction is free.

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.

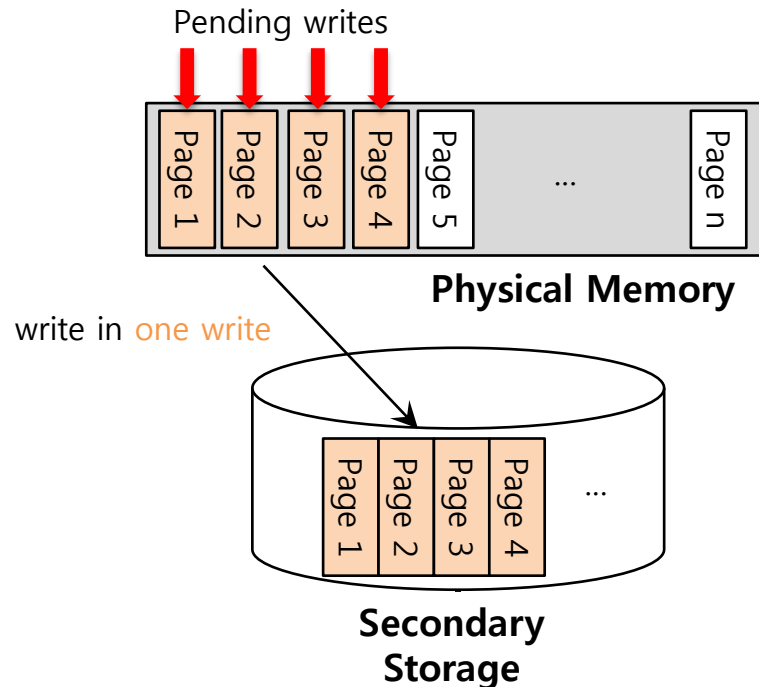


Page 2 likely soon be accessed and thus should be brought into memory too

Clustering, Grouping

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

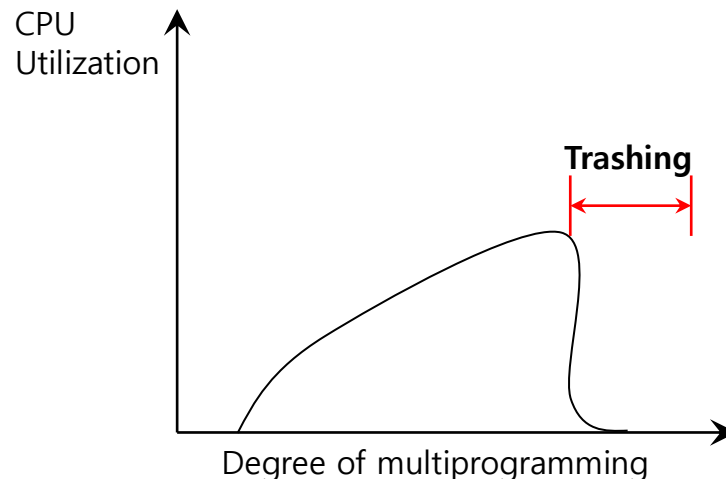
Perform a **single large write** more efficiently than **many small ones**.



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.



Virtual Memory Mechanisms

Hardware and OS cooperate to translate addresses

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

If **TLB miss**...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory

If **page fault** (i.e., present bit is cleared)

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
 - Write victim page out to disk if modified (add dirty bit to PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

What should scheduler do?

Mechanism for Continuing a Process

Continuing a process after a page fault is tricky

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
 - When instruction is being fetched
 - When data is being loaded or stored
- Requires hardware support
 - **precise interrupts**: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

Complexity depends upon instruction set

- Can faulting instruction be restarted from beginning?
 - Example: `move +(SP), R2`
 - Must track side effects so hardware can undo

Virtual Memory Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions

- Page selection
 - **When** should a page (or pages) on disk be **brought into** memory?
- Page replacement
 - **Which** resident page (or pages) in memory should be **thrown out** to disk?

Page Selection

When should a page be brought from disk into memory?

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (**oracle**) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- **Problems?**

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: `madvise()` in Unix

Page Replacement

Which page in main memory should be selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

FIFO: Replace page that has been in memory the longest




























- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
 - Harder to implement, must track which pages have been accessed
 - Does not handle all workloads well

Page Replacement Example

Page reference string: ABCABDADBCB

		OPT	FIFO	LRU
Metric: Miss count	ABC			
	A			
5, 7, 5 misses	B			
	D			
	A			
	D			
	B			
	C			
	B			

Three pages
of physical memory

Page Replacement Comparison

Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
 - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
 - Stack property: smaller cache always subset of bigger
- FIFO: Add more memory, usually have fewer page faults
 - Belady's anomaly: May actually have **more** page faults!

FIFO Performance may Decrease!

Consider access stream: ABCDABEABCDE

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

3 pages: 9 misses

4 pages: 10 misses

Problems with LRU-based Replacement

LRU does not consider frequency of accesses

- Is a page accessed **once** in the past equal to one accessed **N** times?
- Common workload problem:
 - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Examples of other more sophisticated algorithms:

- LRU-K and 2Q: Combines recency and frequency attributes
- Expensive to implement, LRU-2 used in databases

Implementing LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

Clock Algorithm

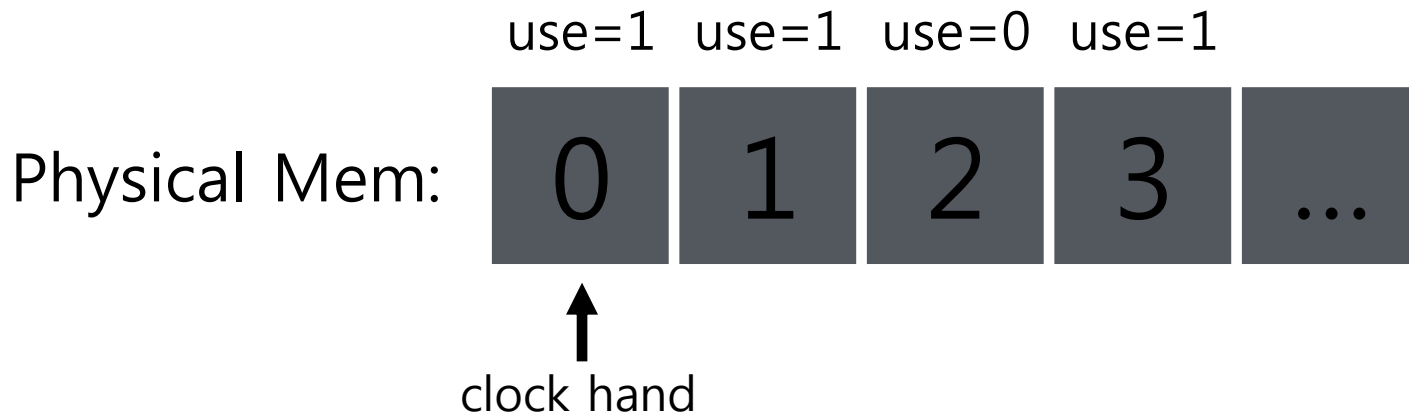
Hardware

- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

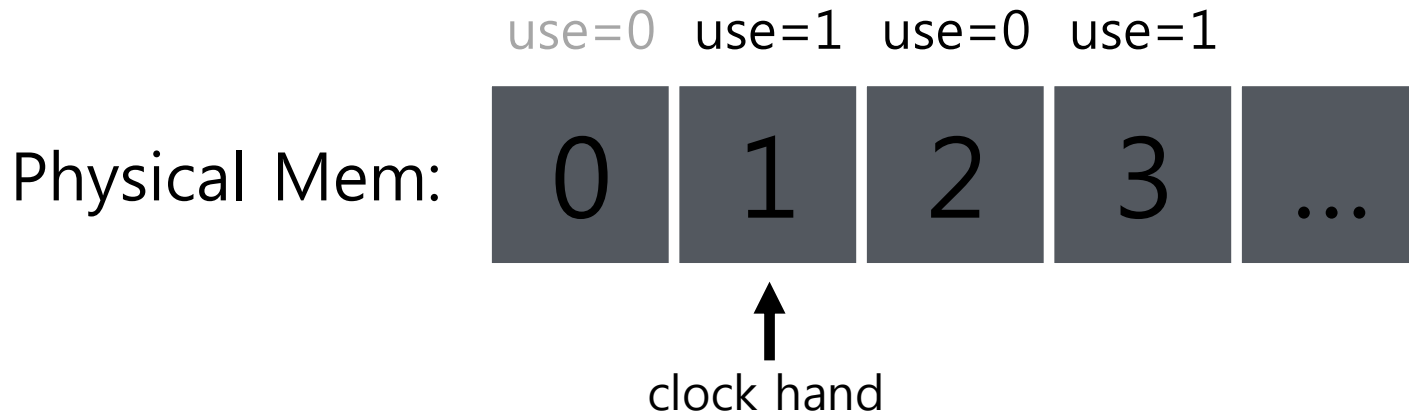
Operating System

- Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
- Implementation:
 - Keep pointer to last examined page frame
 - Traverse pages in circular buffer
 - Clear use bits as search
 - Stop when find page with already cleared use bit, replace this page

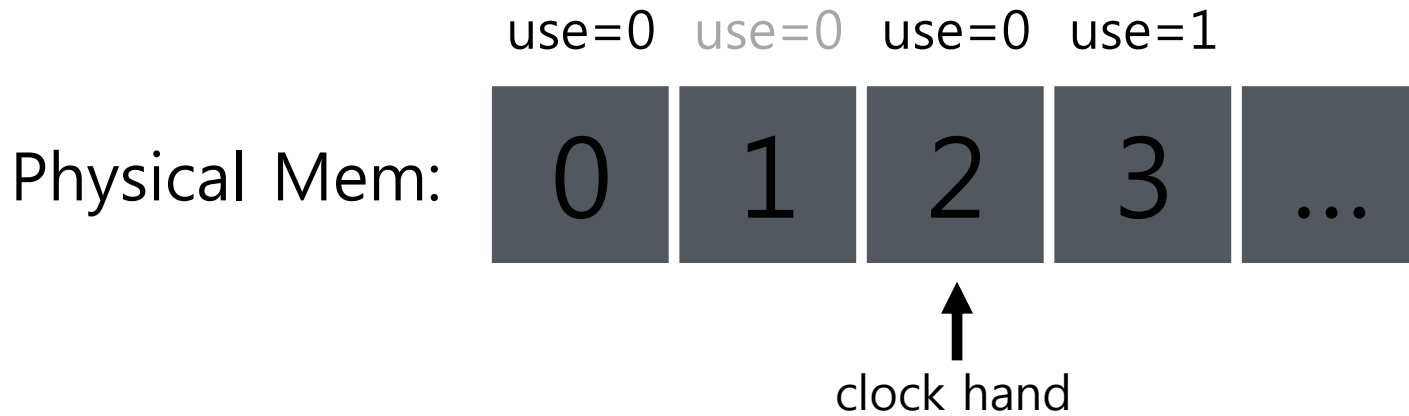
Clock: Look For a Page



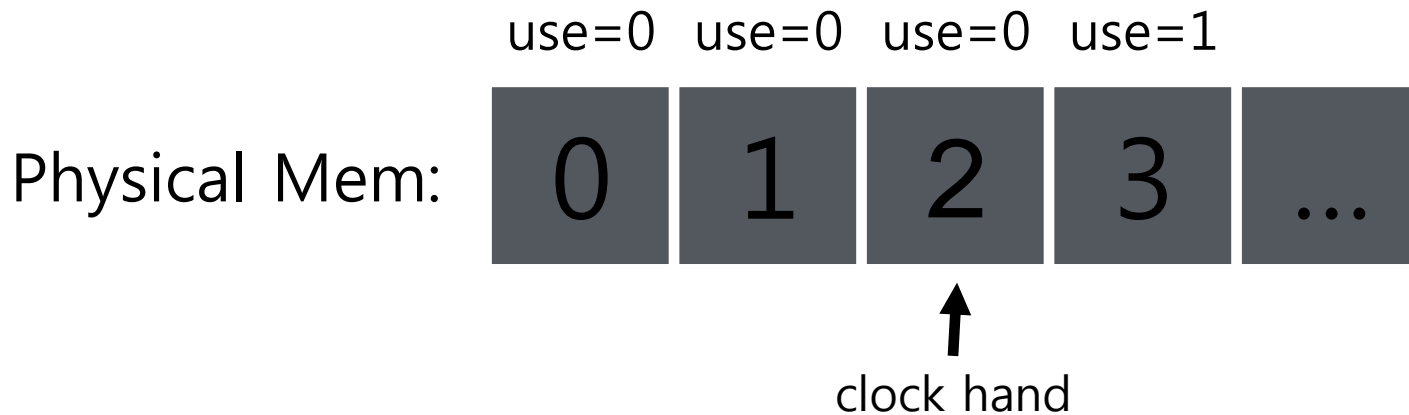
Clock: Look For a Page



Clock: Look For a Page

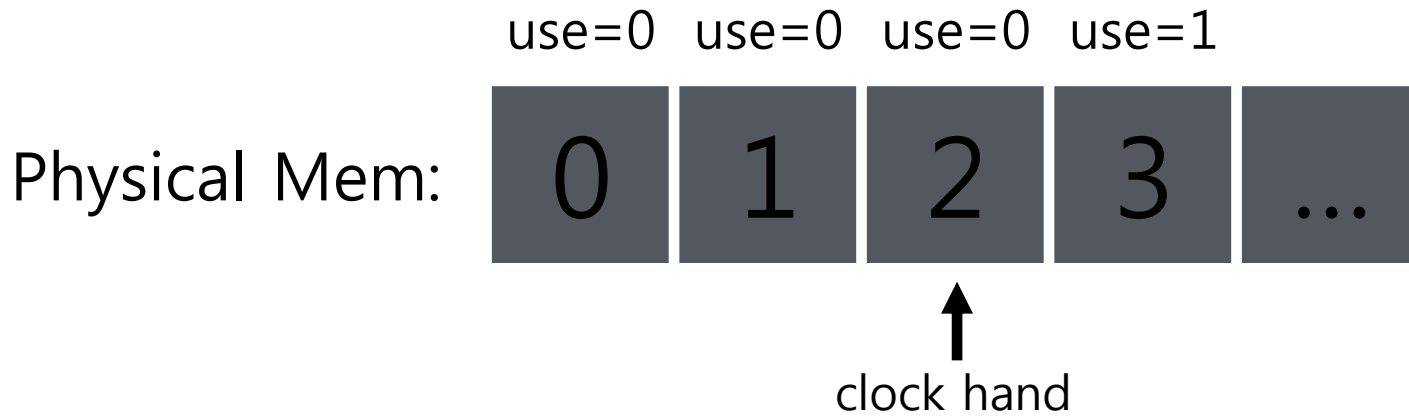


Clock: Look For a Page



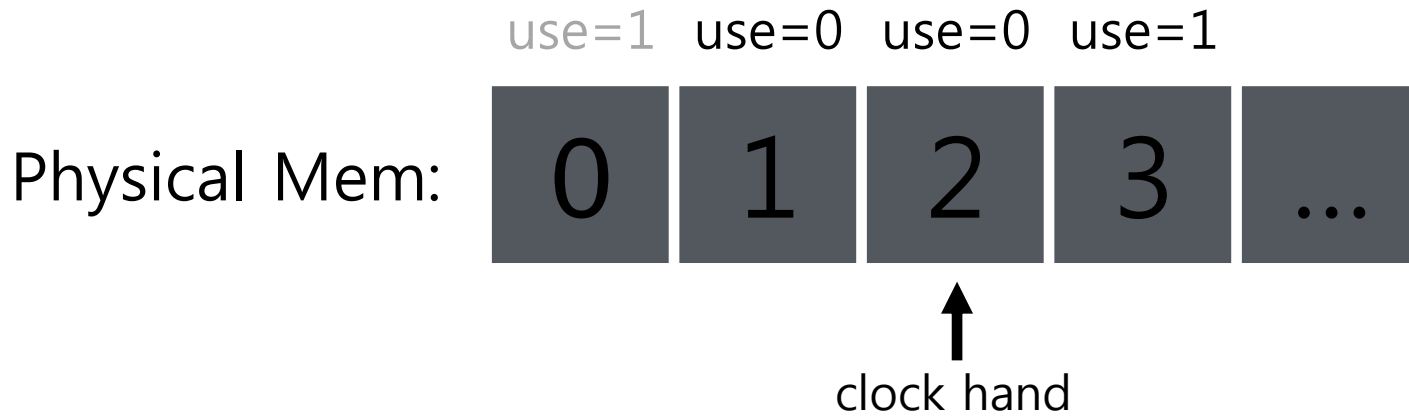
evict **page 2** because it has not been recently used

Clock: Look For a Page

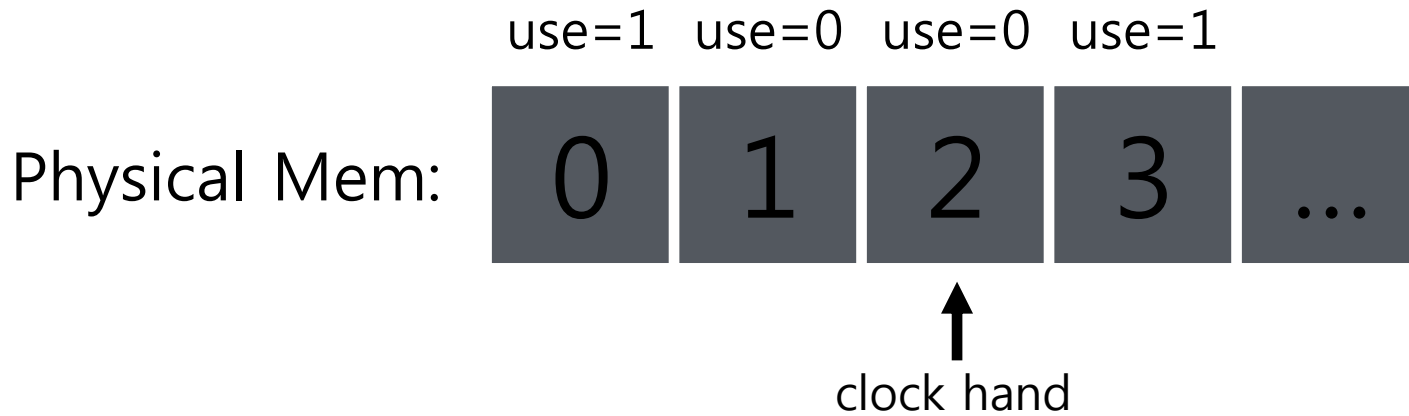


page 0 is accessed...

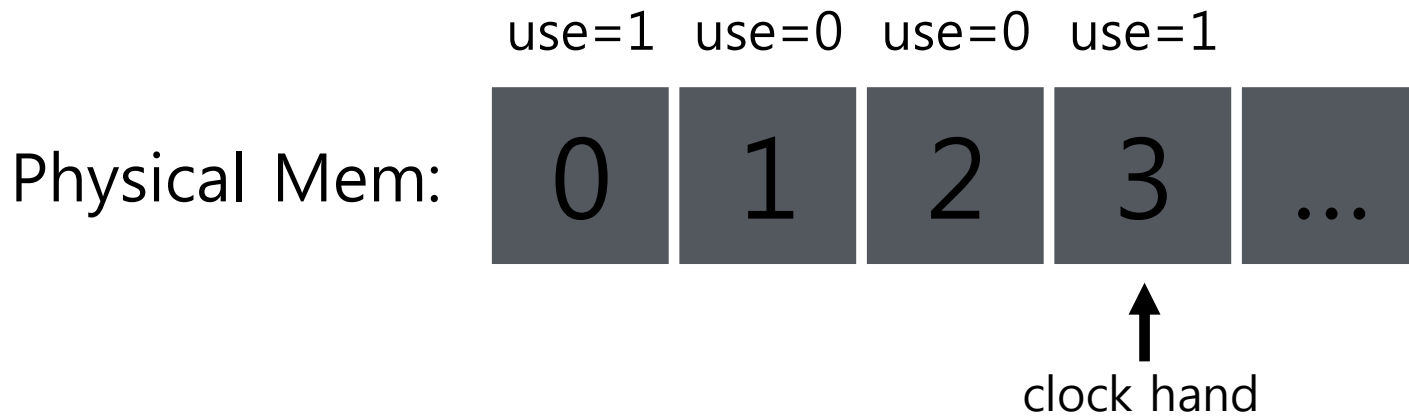
Clock: Look For a Page



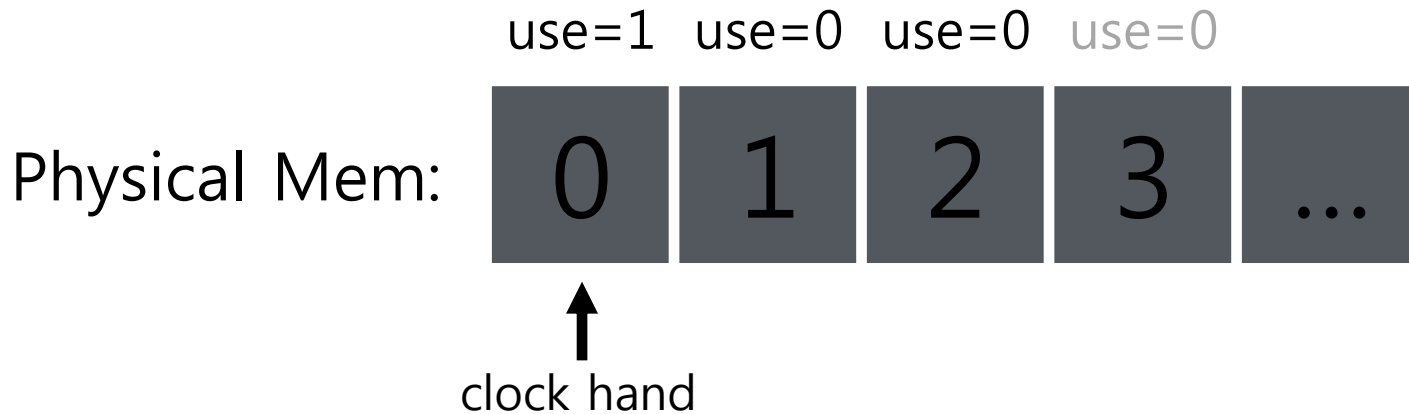
Clock: Look For a Page



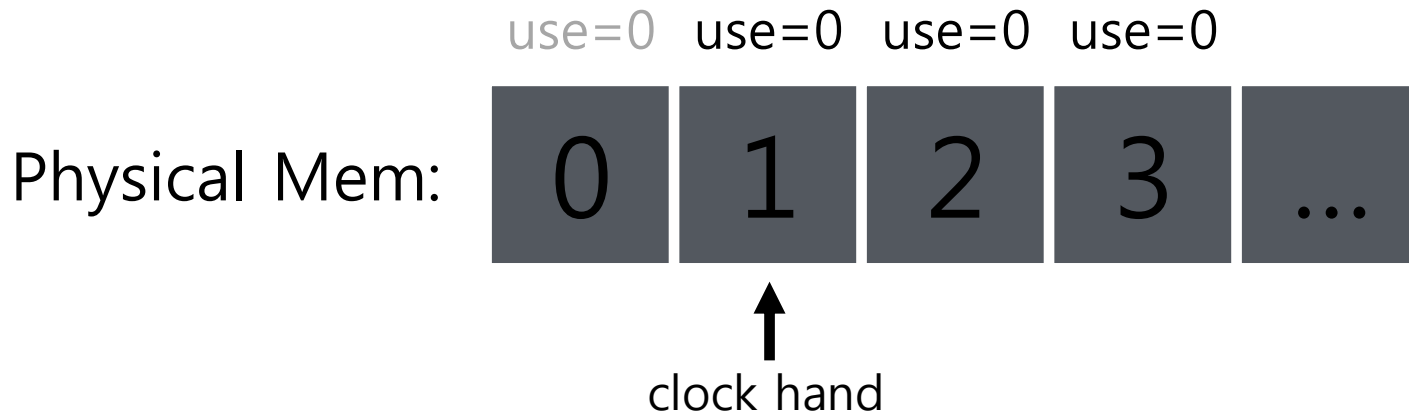
Clock: Look For a Page



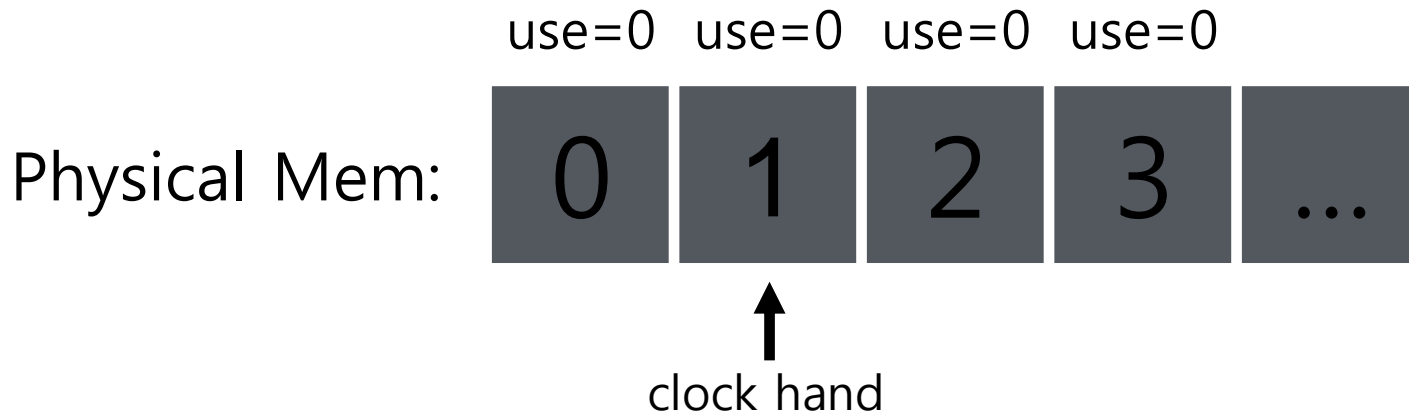
Clock: Look For a Page



Clock: Look For a Page



Clock: Look For a Page



evict **page 1** because it has not been recently used

Other factors

Assume page is both in RAM **and** on disk

Do we have to write to disk for eviction?

- not if page is **clean**
- track with **dirty bit**

Thrashing

A machine is **thrashing** when there is not enough RAM, and we constantly swap in/out pages

Solutions?

- admission control (like scheduler project)
- buy more memory
- Linux **out-of-memory killer**!

Clock Extensions

Replace multiple pages at once

- Intuition:
Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Add software counter ("chance")

- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment software counter if use bit is 0
- Replace when chance exceeds some specified limit

Use dirty bit to give preference to dirty pages

- Intuition: More expensive to replace dirty pages
 - Dirty pages must be written to disk, clean pages do not
- Replace pages that have use bit and dirty bit cleared

What if no Hardware Support?

What can the OS do if hardware does not have use bit (or dirty bit)?

- Can the OS “emulate” these bits?

Leading question:

- How can the OS get control (i.e., generate a trap) every time use bit should be set? (i.e., when a page is accessed?)

Conclusions

Illusion of virtual memory:

Processes can run when sum of virtual address spaces $>$ amount of physical memory

Mechanism:

- Extend page table entry with "present" bit
- OS handles page faults (or page misses) by reading in desired page from disk

Policy:

- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) perform approximation of LRU