Swapping and Demand Paging

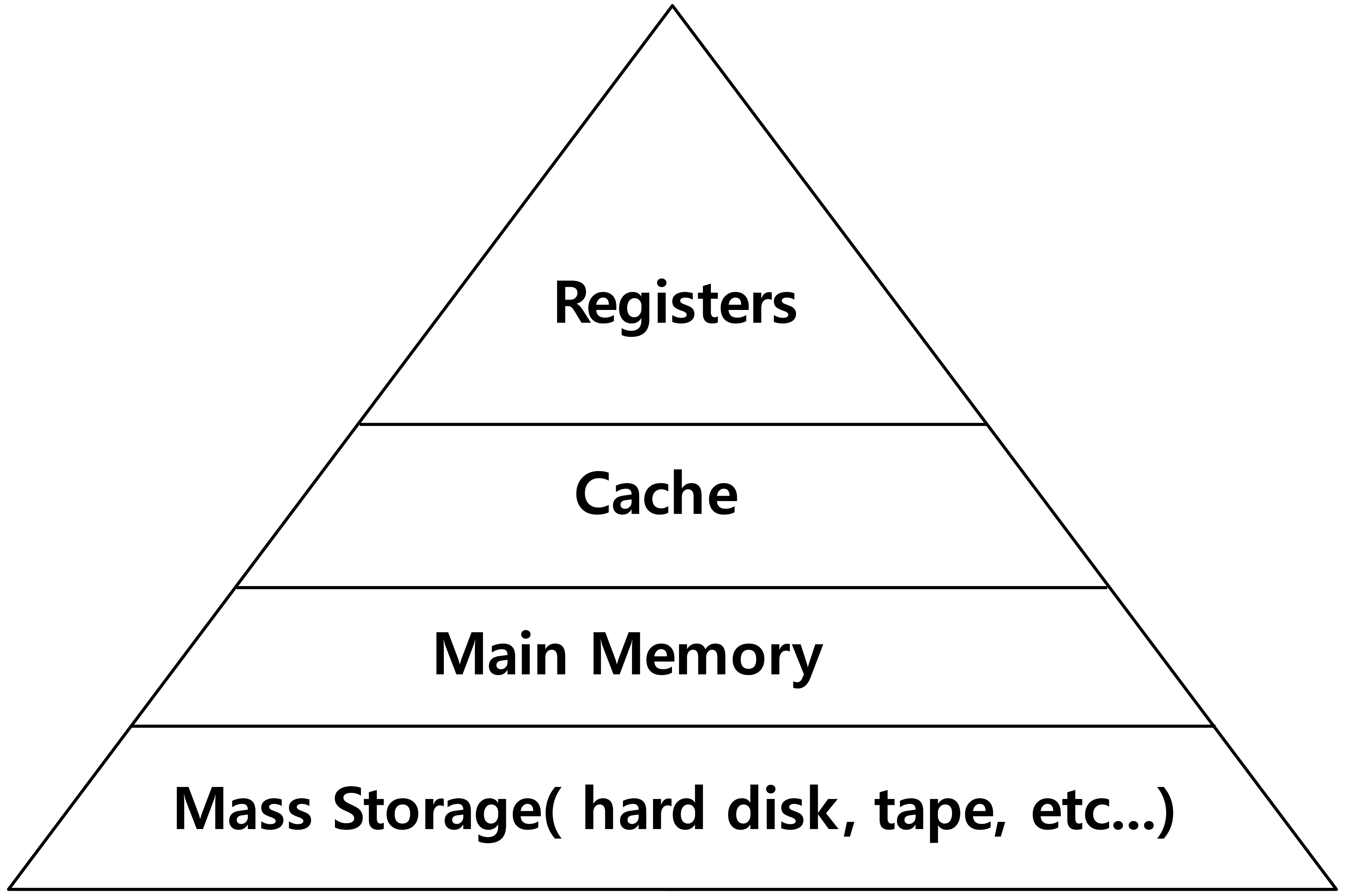
02/28/2023

Professor Amanda Bienz

Textbook pages 376-378, 389-412

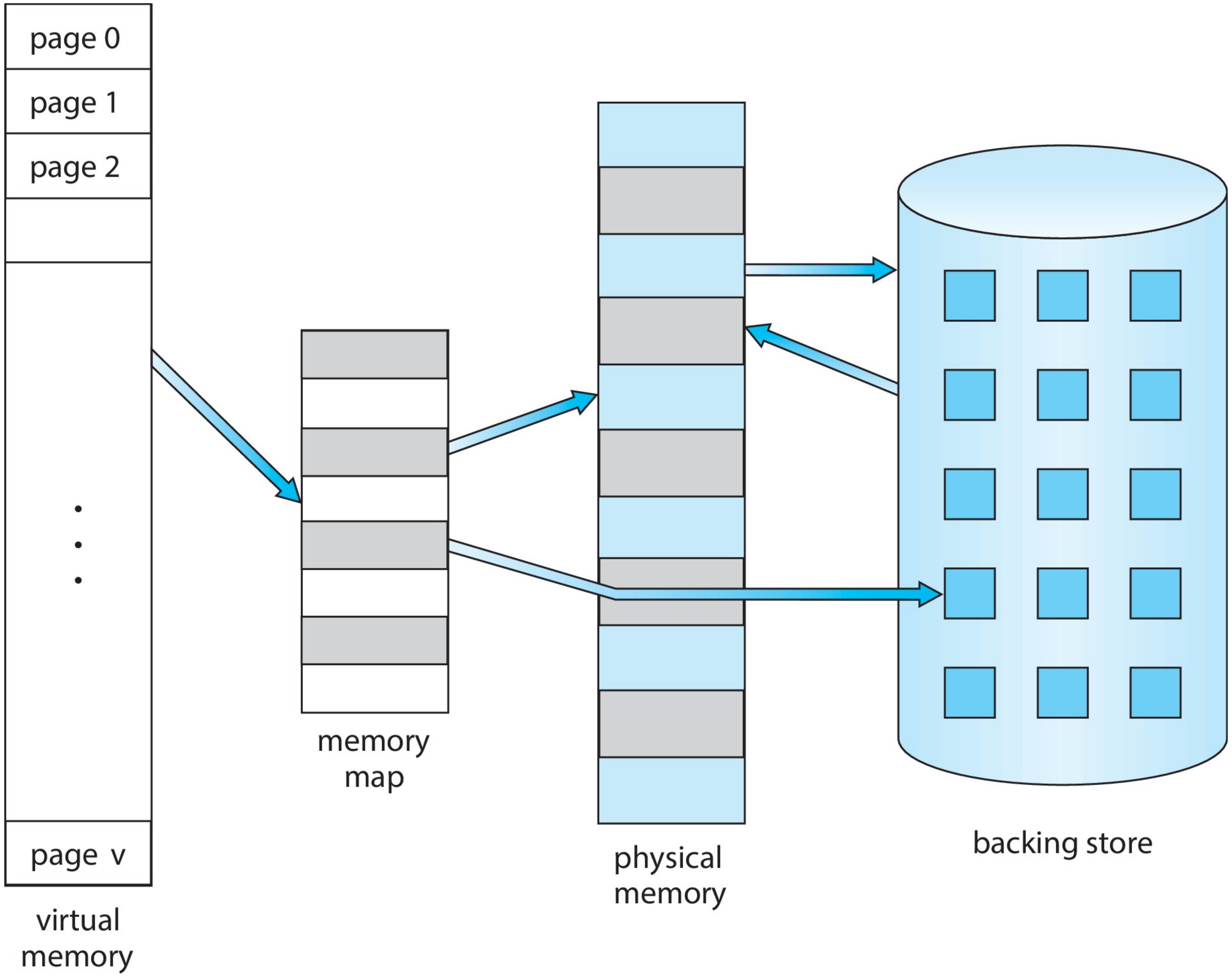
Beyond Physical Memory : Mechanisms

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



Memory Hierarchy in modern system

Virtual Memory Larger than Physical Memory



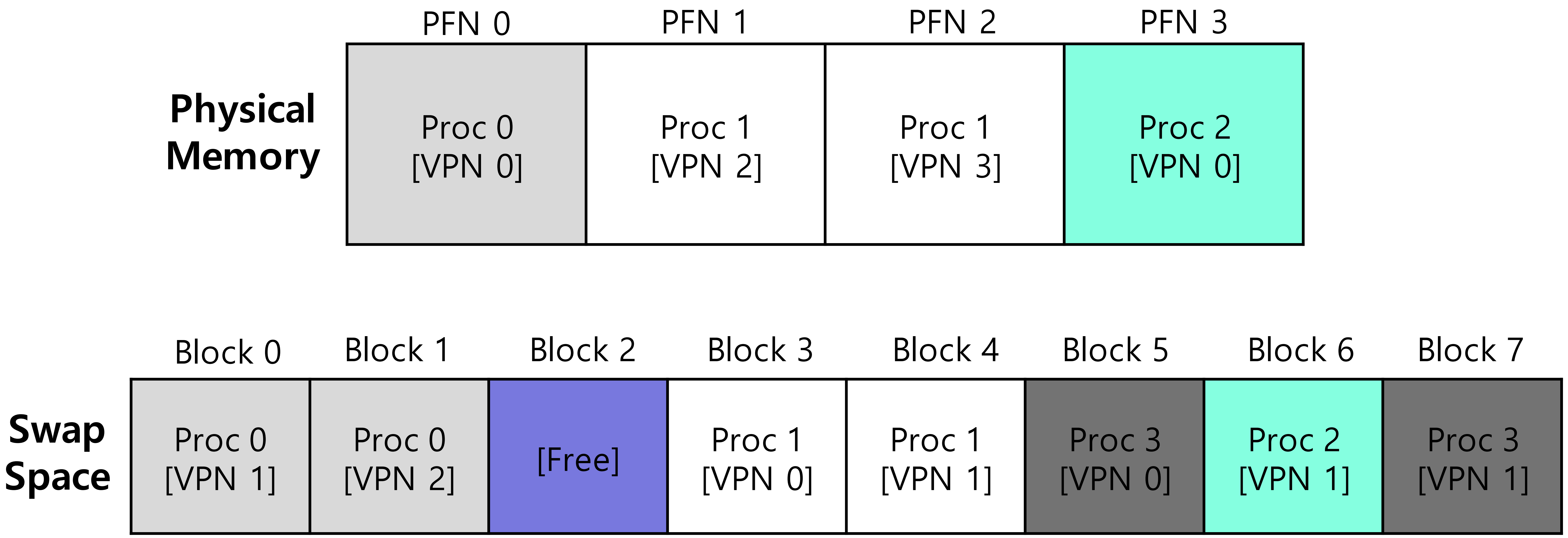
Swapping

- Process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store -** fast disk large enough to accommodate copies of all memory images for all users
- **Roll out, roll in -** swapping variant used for priority-based scheduling algorithms; lower priority process is swapped out so higher priority process can be loaded and executed
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swap Space

- Reserve some space on disk for moving pages back and forth

- Page-sized units



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

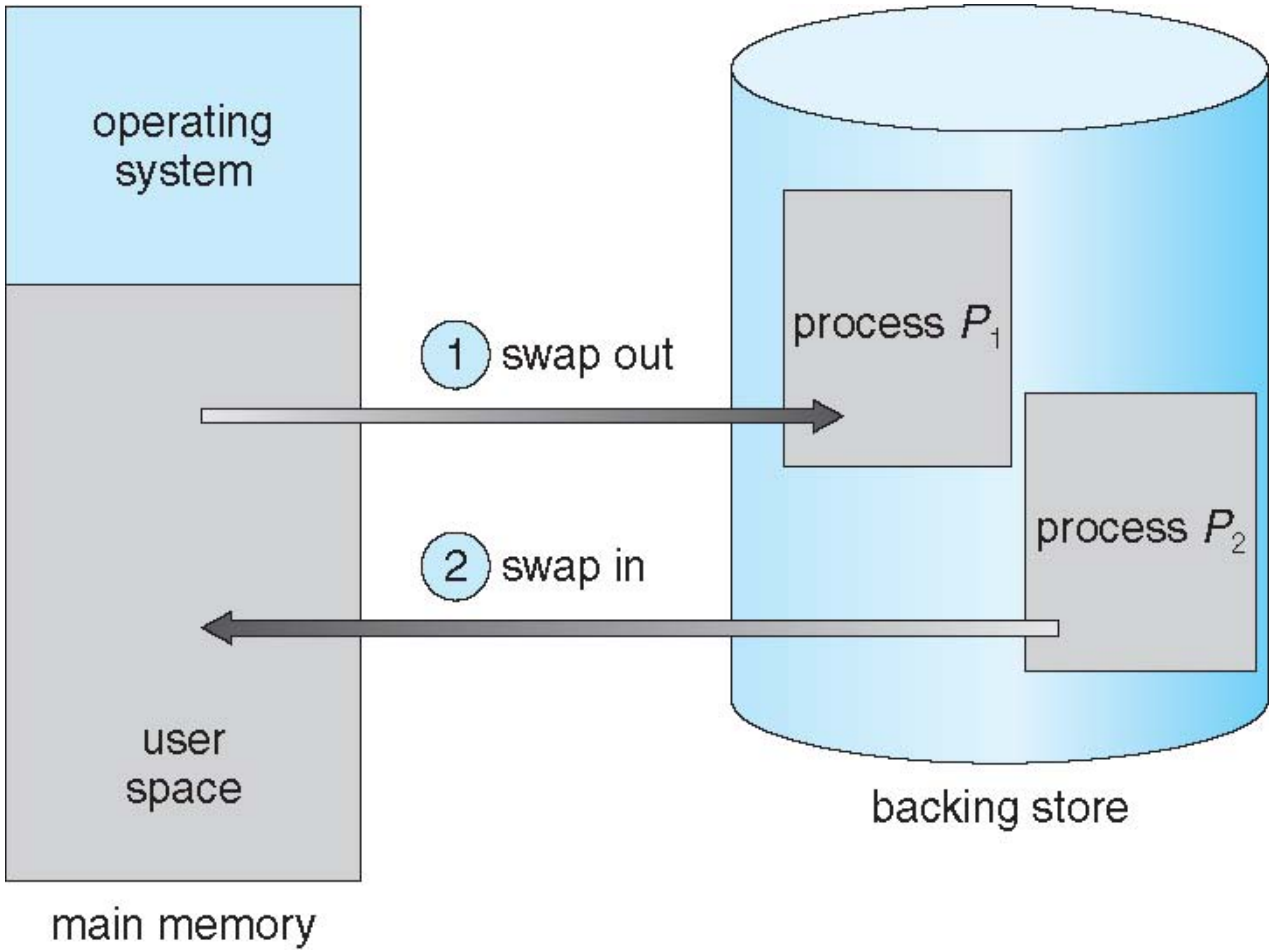
Swap Space	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

Physical Memory and Swap Space

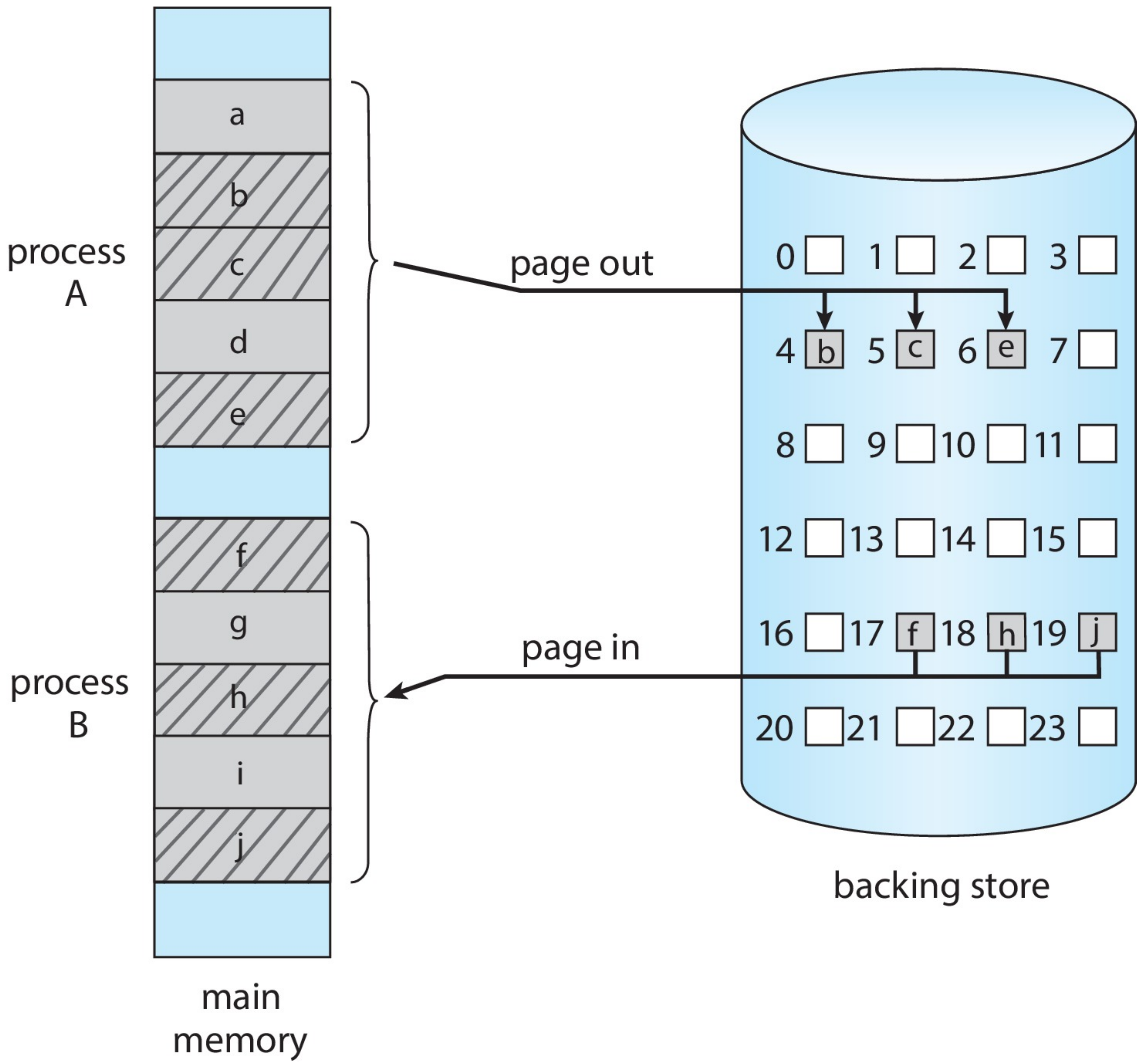
Swapping

- Modified versions of swapping found on main systems (UNIX, Linux, Windows)
    - Swapping disabled by default
    - Started if more than threshold amount of memory allocated
    - Disabled again once memory demand reduced below threshold

Schematic of Swapping

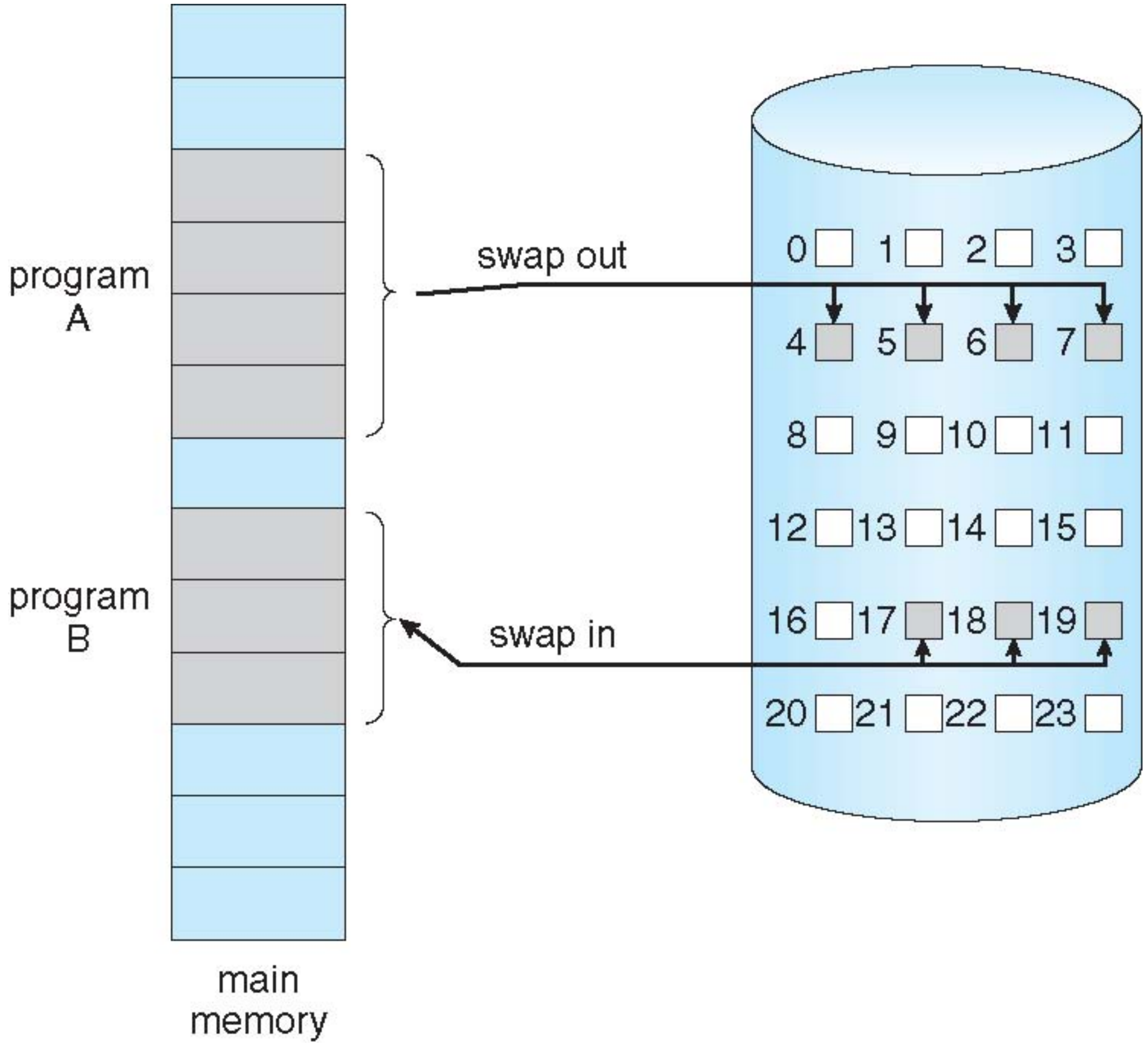


Swapping with Paging



Demand Paging

- Could bring entire process into memory at load time
- Instead, bring page into memory only when needed
  - Less I/O (none unnecessary)
  - Less memory required
  - Faster response time
- Page is needed — reference to it
  - Invalid reference — abort
  - Not in memory — bring to memory

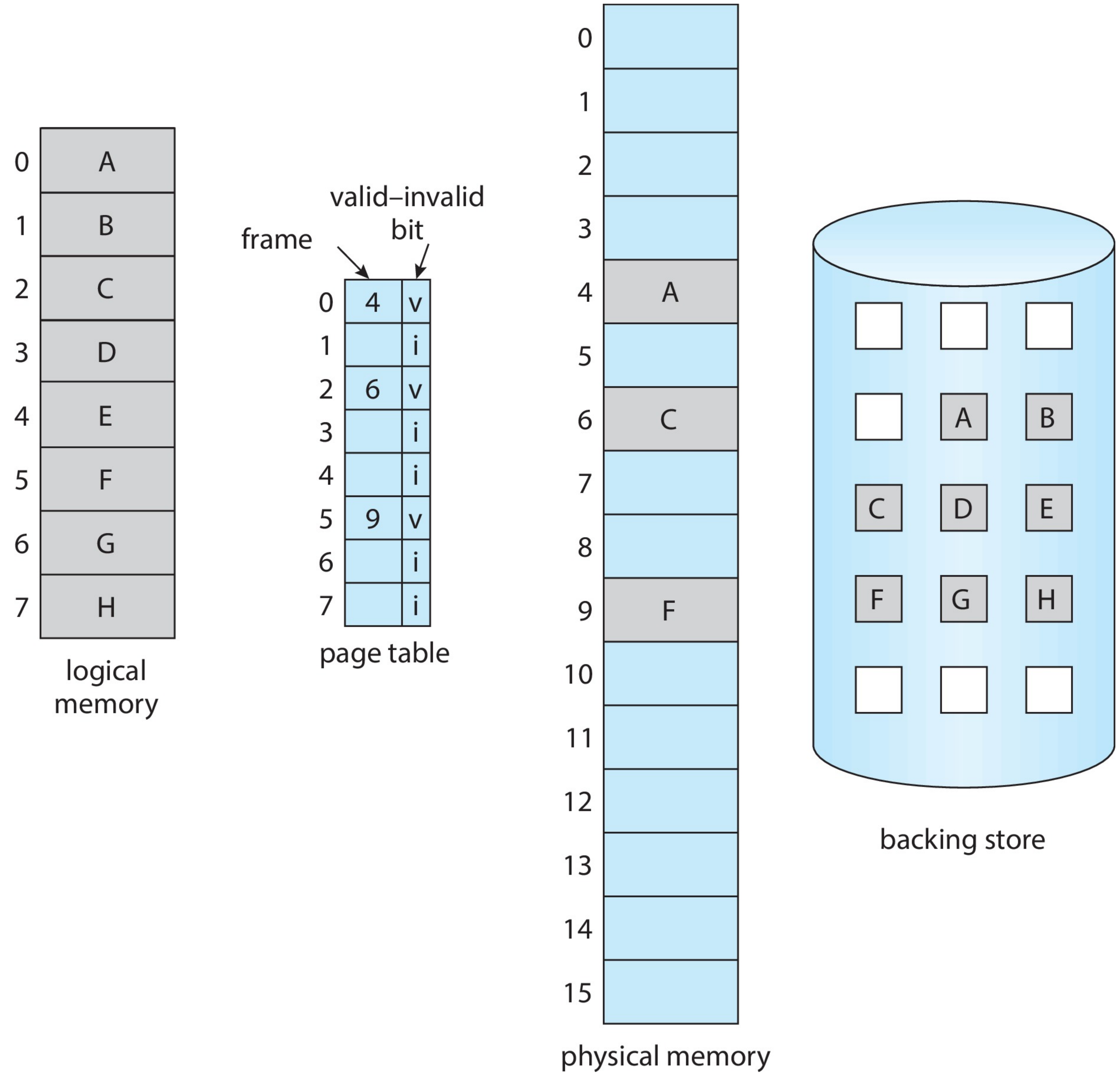


Present / Valid Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk
  - When hardware looks in Page Table Entry, 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.

Page Table with Invalid Bits



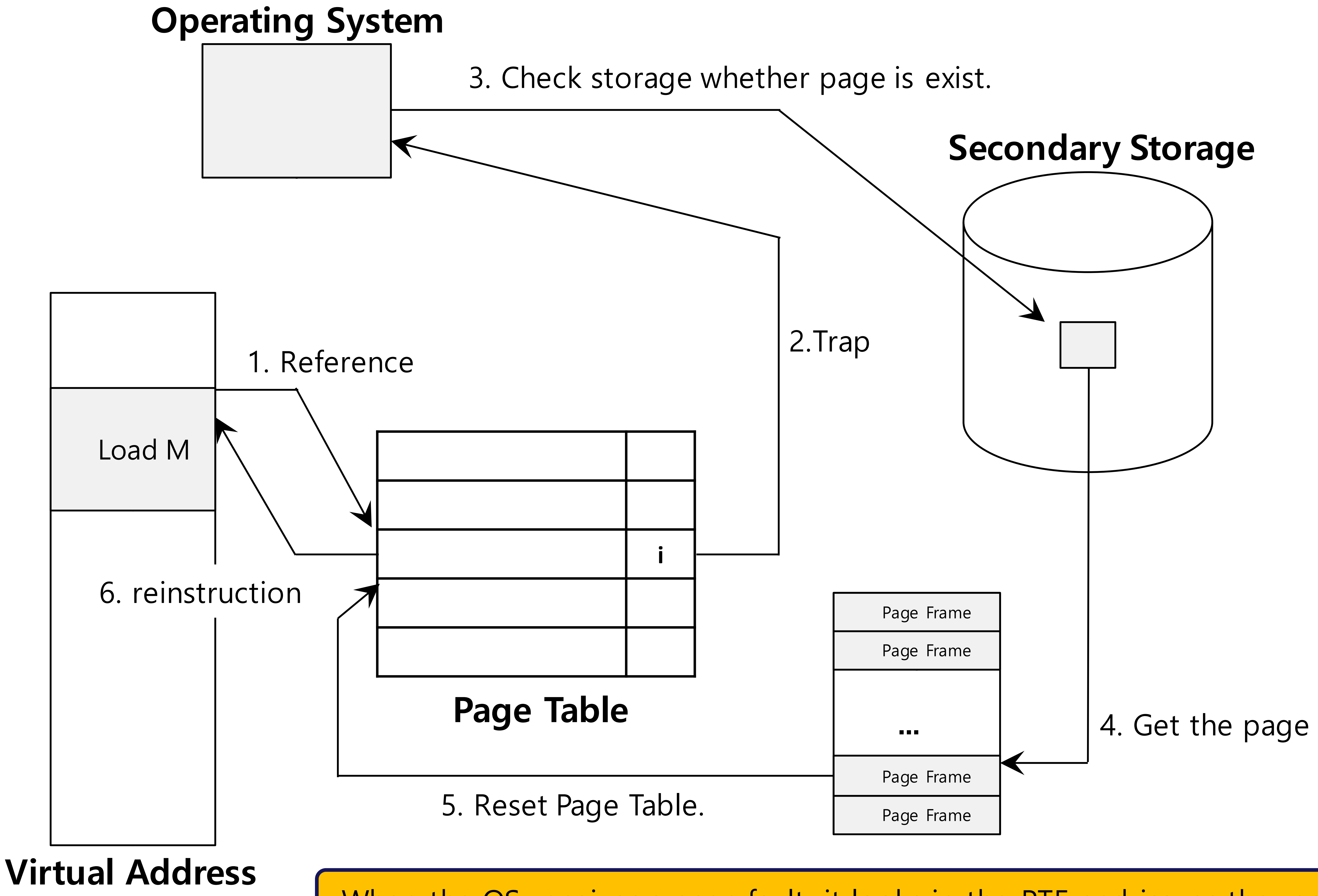
	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

Page Fault

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

Page Fault Control Flow

- Page table entry used for data such as physical frame number of the page for a disk address

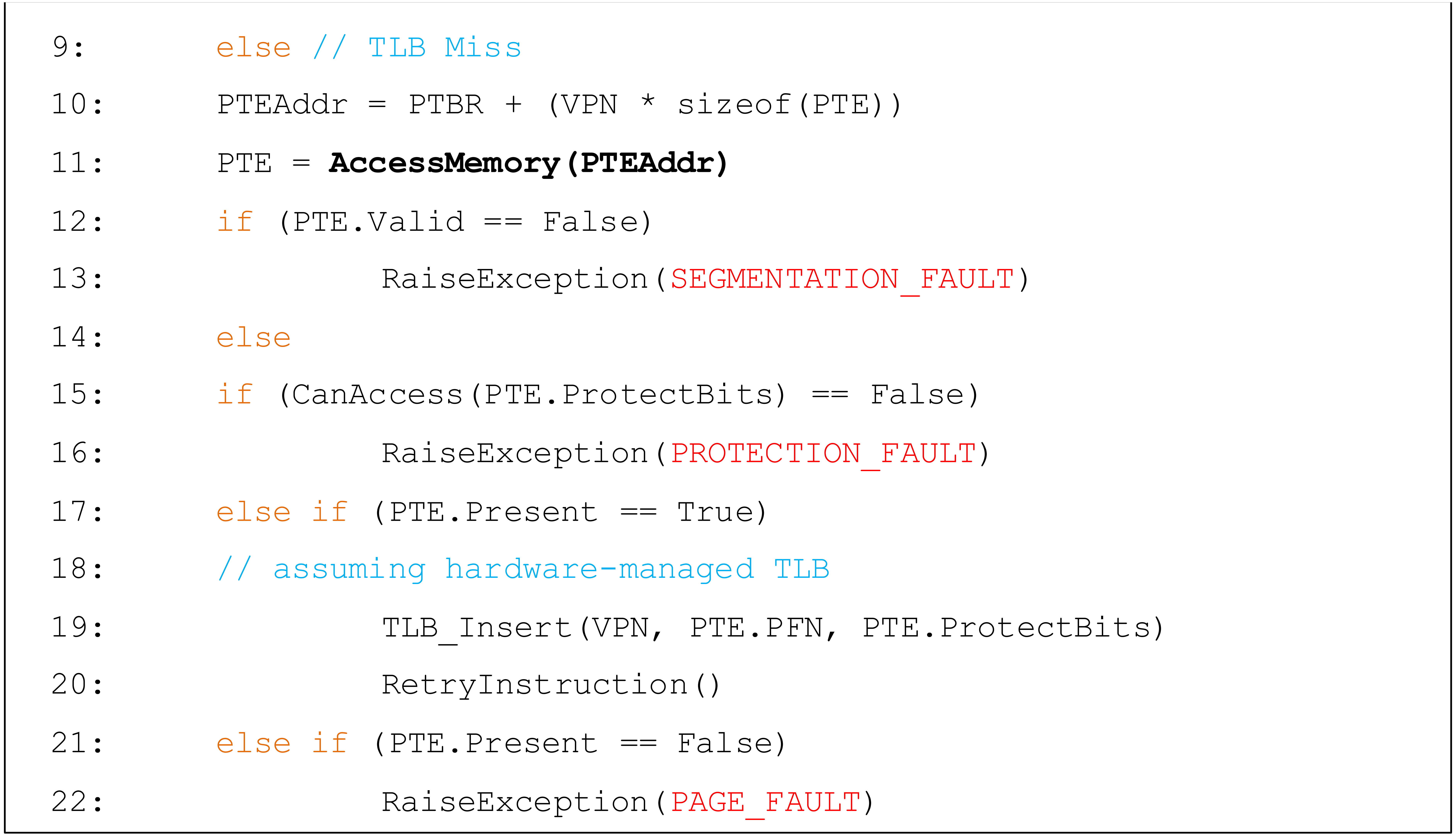


When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page Fault Control Flow - Hardware

```
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
```

- OS must find a physical frame for soon-be-paged-in page to reside within
- If no such page, wait for replacement algorithm to run and kick some pages out of memory

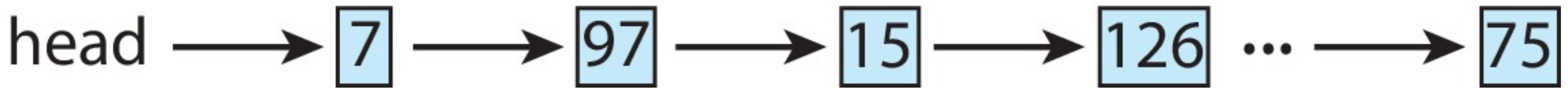
Aspects of Demand Paging

- Extreme case - start process with no pages in memory
  - OS sets instruction point to first instruction of process, non-memory-resident — page fault
  - **Pure demand paging**
- Actually, given instruction can access multiple pages — multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**

Free Frame List

- When a page fault occurs, OS must bring desired page from secondary storage into main memory

- Most operating systems maintain a free-frame list : a pool of free frames for satisfying such requests



head → 7 → 97 → 15 → 126 ••• → 75

- OS allocate free frames using a technique known as **zero-fill-on-demand** : the content of the frames zeroed-out before being allocated

- When a system starts up, all available memory is placed in the free-frame list

Stages of Demand Paging

1. Send trap to the OS
2. Save user registers and process state
3. Determine that interrupt was a page fault
4. Check that the page reference was legal and determine location of the page on the disk
5. Issue a read from the disk to a free frame
    1. Wait in a queue for this device until read request is serviced
    2. Wait for device seek and/or latency time
    3. Begin transfer of the page to a free frame

Stages of Demand Paging

6. While waiting, allocate CPU to some other process

7. Receive an interrupt from disk I/O subsystem (I/O completed)

8. Save registers and process state for other process

9. Determine that interrupt was from disk

10. Correct page table and other tables to show page is now in memory

11. Wait for CPU to be allocated to this process again

12. Restore user registers, process state, and new page table, and then resume interrupted instruction

Performance of Demand Paging

- Three major activities
  - Service the interrupt : several hundred instruction
  - Read the page : this takes a long time
  - Continue the process : very quick
- Page Fault Rate : 0 <= p <= 1
  - If p = 0, no page faults
  - If p = 1, every reference is a page fault

Effective Access Time (EAT)

- EAT = (I - p) x memory access
  + p x (page fault overhead + swap page out + swap page in)

- Example : memory access time is 200 nanoseconds
  average page-fault service time is 8 milliseconds

  - EAT = (1-p) * 200 + p * 8,000,000 = 200 + p*7,999,800

  - If one access out of 1,000 causes page fault, EAT = 8.2 microseconds
    Slowdown by a factor of 40!

- If want performance degradation < 10 percent, how often can we have page faults?

Effective Access Time (EAT)

- EAT = (I - p) x memory access
  + p x (page fault overhead + swap page out + swap page in)

- Example : memory access time is 200 nanoseconds
  average page-fault service time is 8 milliseconds

  - EAT = (1-p) * 200 + p * 8,000,000 = 200 + p*7,999,800

  - If one access out of 1,000 causes page fault, EAT = 8.2 microseconds
    Slowdown by a factor of 40!

- If want performance degradation < 10 percent, how often can we have page faults? **Less than 1 page fault every 400,000 memory accesses**

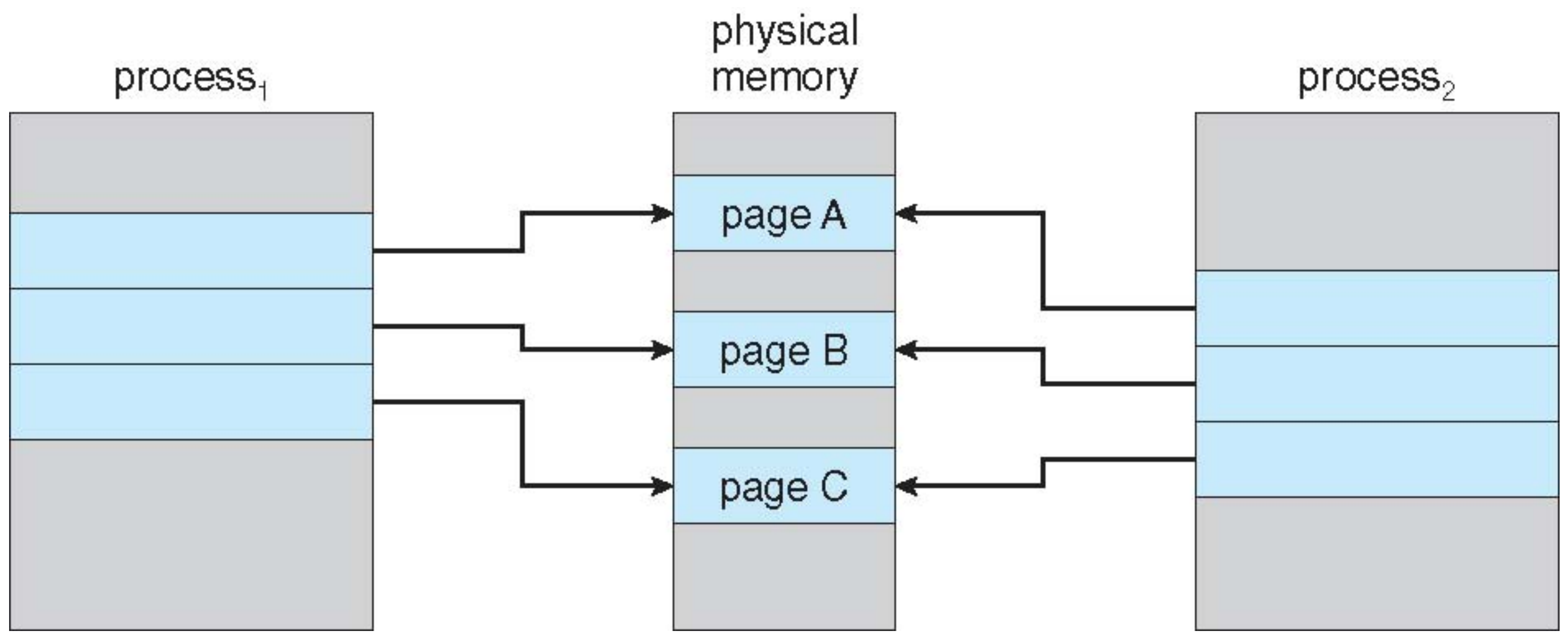
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older UNIX systems
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris
  - Still need to write to swap space (e.g. modified pages)

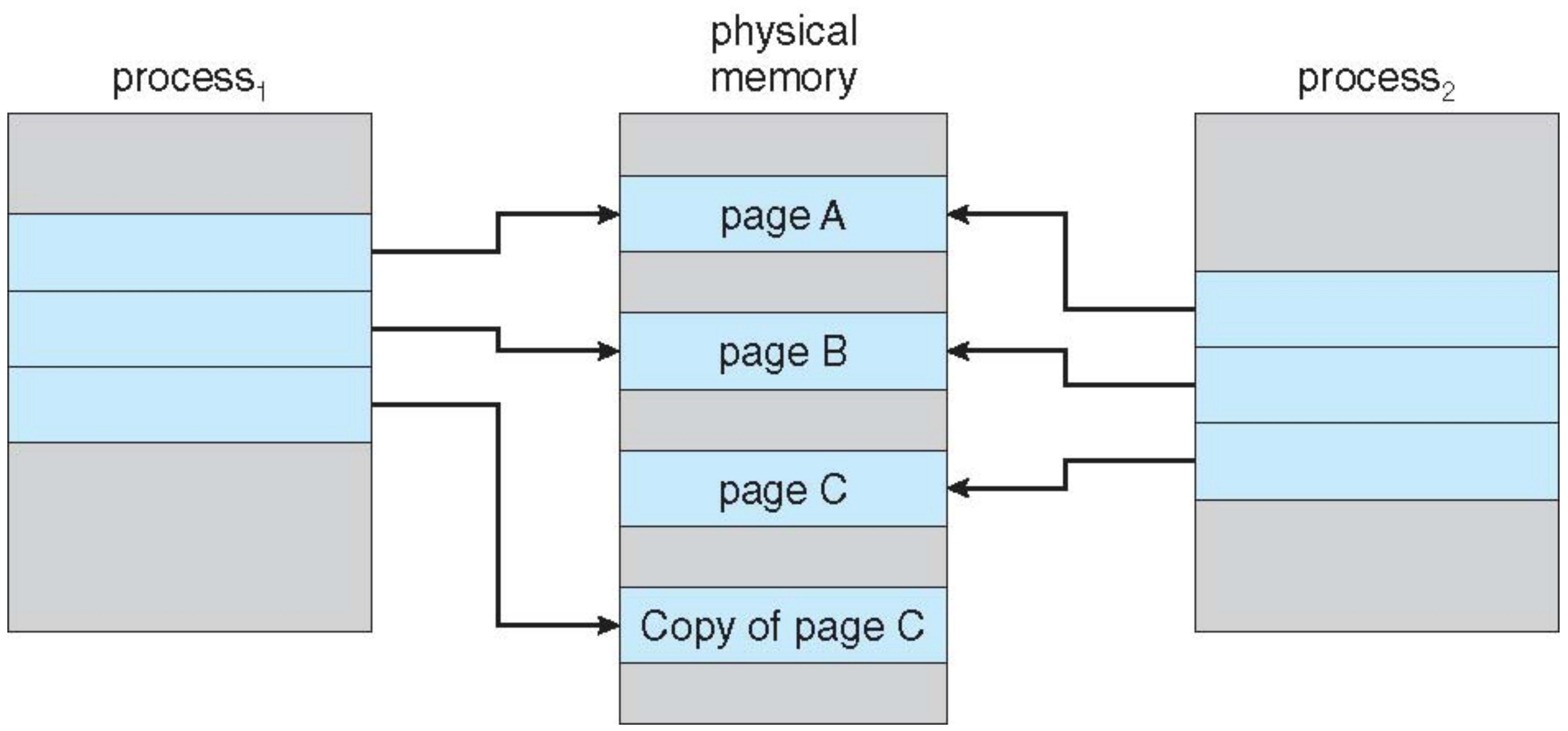
Copy-on-Write

- **Allows both parent and child processes to initially share same pages in memory**
  - If either process modifies a shared page, only then is the shared page copied
- Allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  - Pool should always have free frames for fast demand page execution
  - Don’t want to have to free a frame as well as other processing when you have a page fault
- Why zero-out a page before allocating it?

Copy on Write



Before Process 1 writes to page C



After Process 1 writes to page C

Copy-on-Write Performance

- COW allows more efficient process creation as only modified pages are copied
- If most pages will end up being copied, COW can be slower in total
- Even in those cases, time to execution of first instruction after fork (latency) is lower!

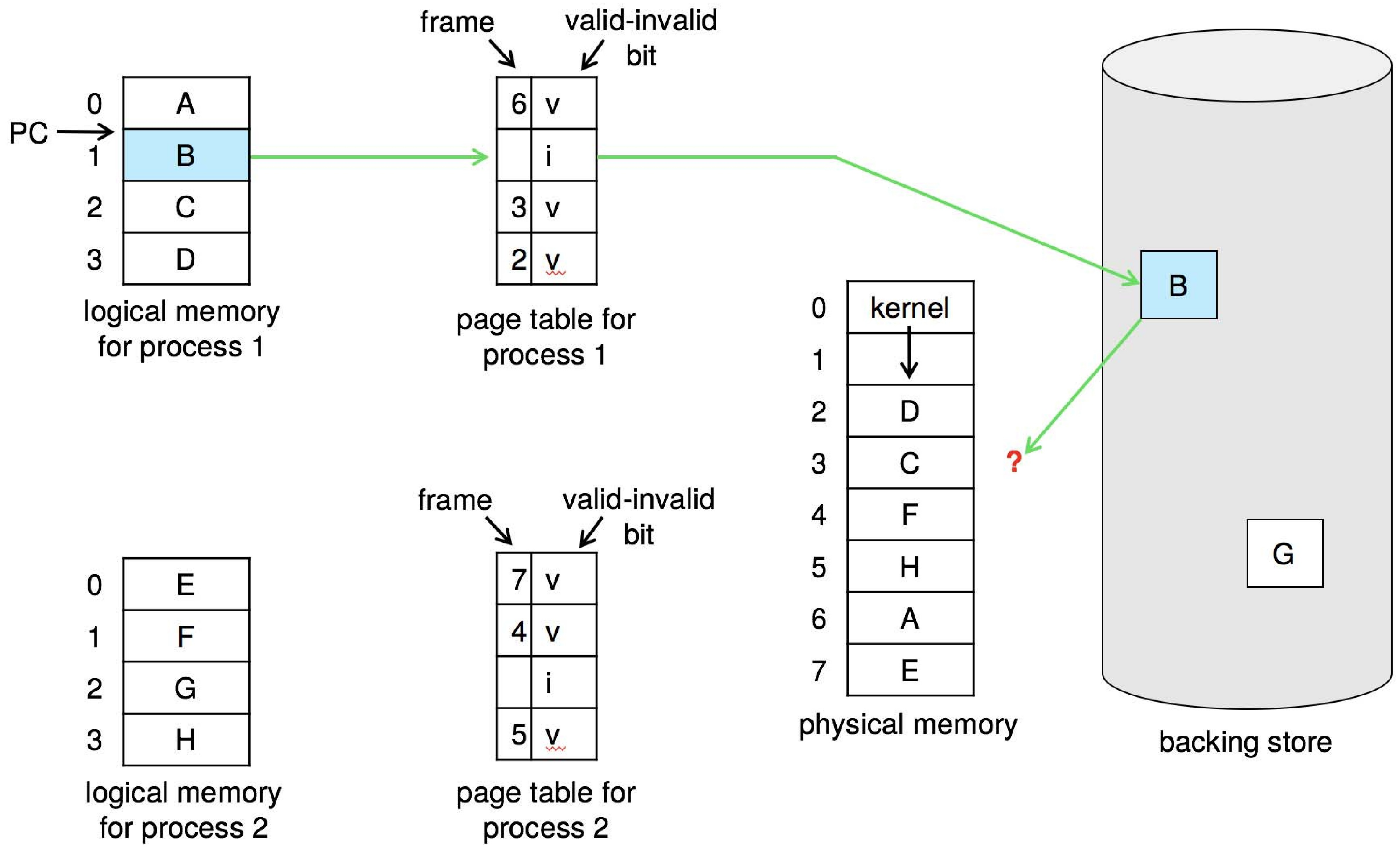
Copy on Write

- vfork() : variation of fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

What if Memory is Full?

- OS will page out pages to make room for new pages the OS is about to bring in
  - Process of picking a page to kick out, or replace, is known as page-replacement policy
- Use dirty bit to reduce overhead of page transfers (only modified pages written to disk)

Need for Page Replacement



When Replacements Really Occur

- OS waits until memory is nearly full, and only then replaces a page to make room for another page
    - This is a bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively
    - Generally keep a low water mark and high water mark on number of free physical pages desired
- Swap Daemon, Page Daemon
    - There are fewer than LW pages unavailable, a background thread is responsible for freeing memory runs
    - Thread evicts pages until there are HW pages available

Basic Page Replacement

1. Find location of desired page on disk
2. Find a free frame
    1. If there is a free frame, use it
    2. If there is no free frame, use a page replacement algorithm to select a victim frame
    3. Write victim frame to disk if dirty
3. Bring desired page into newly free frame and update page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note : potentially 2 page transfers per page fault - increasing EAT

Beyond Physical Memory : Policies

- Memory pressure forces 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 replacement policy for this cache is to minimize number of cache misses
- Number of cache hits and misses lets us calculate the average memory access time (AMAT)

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

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)

Optimal Replacement Policy

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

Tracing 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.

Simple Policy : FIFO

- Pages are placed in a queue when they enter the system
- When a replacement occurs, page on tail of queue (first in) is evicted
    - Simple to implement, but cannot determine 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 $\frac{Hits}{Hits+Misses} = \mathbf{36.4}\%$

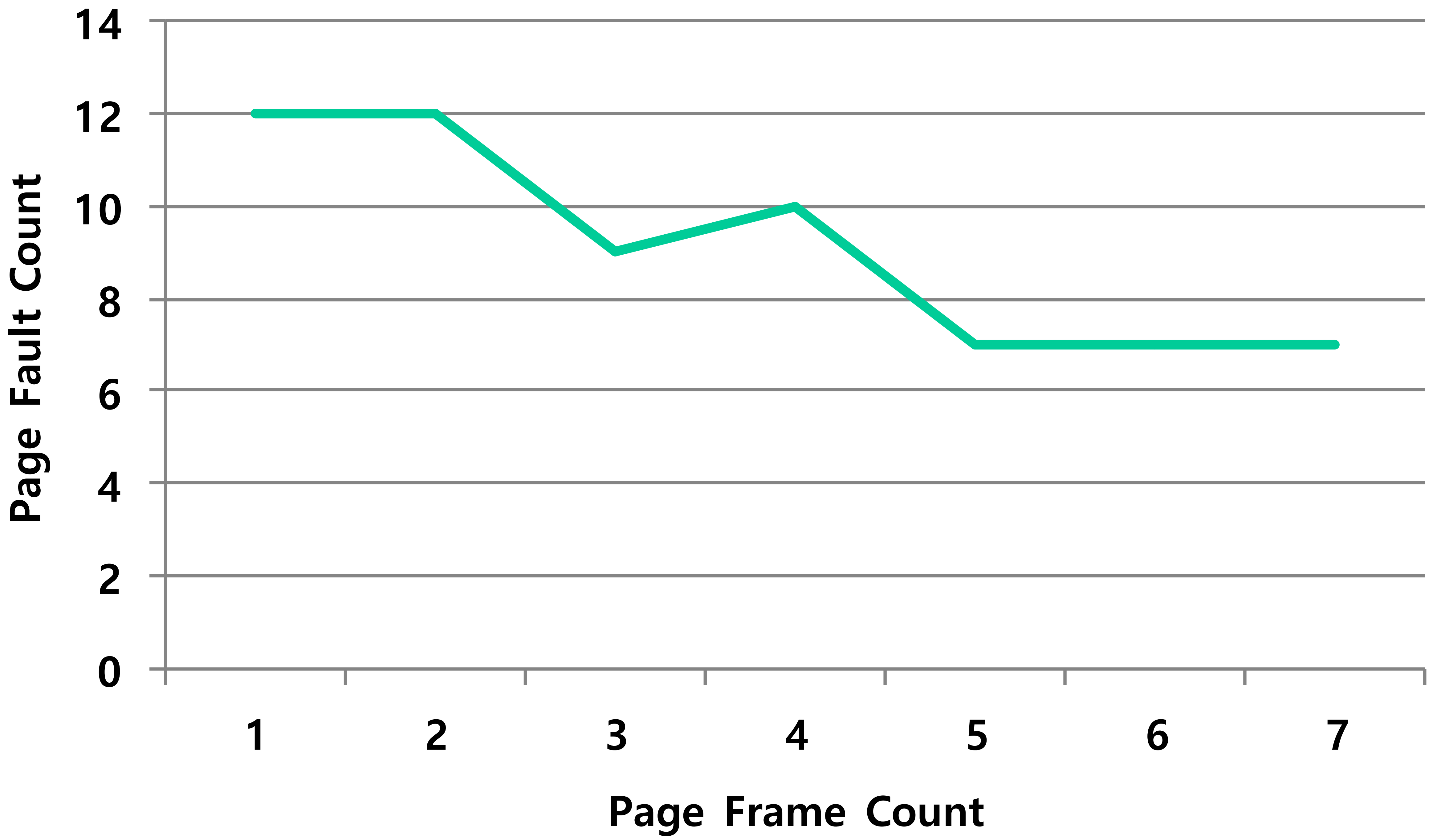
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

Belady's Anomaly

- We would expect cache hit rate to **increase** when the cache gets large. But with FIFO, it gets worse

Reference Row

1 2 3 4 1 2 5 1 2 3 4 5



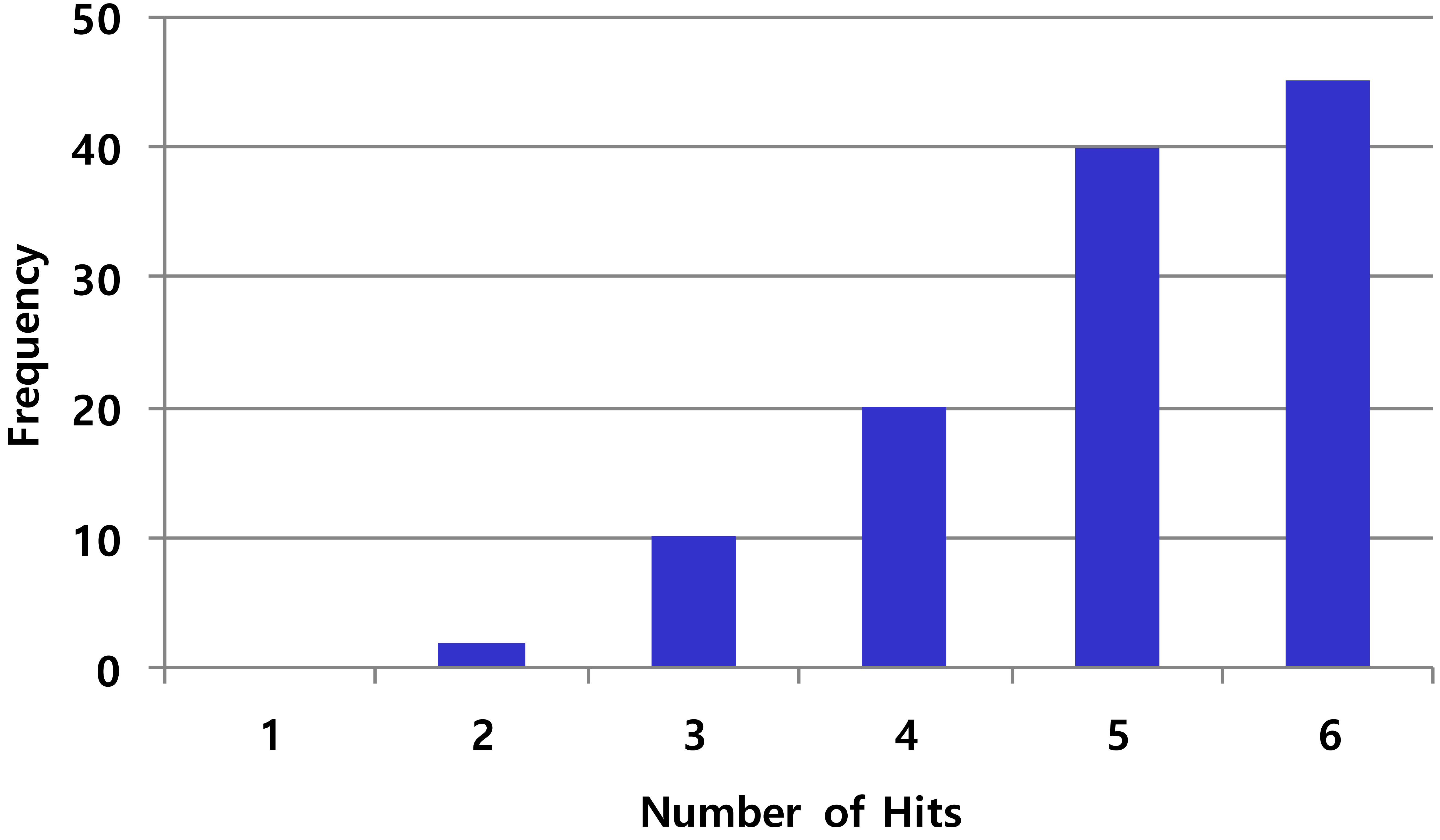
Another Simple Policy : Random

- Picks random page to replace under memory pressure
  - Doesn’t really try to be too intelligent in picking which blocks to evict
  - Performance depends entirely on how lucky it 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

- History can predict future

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 page that was least-recently used

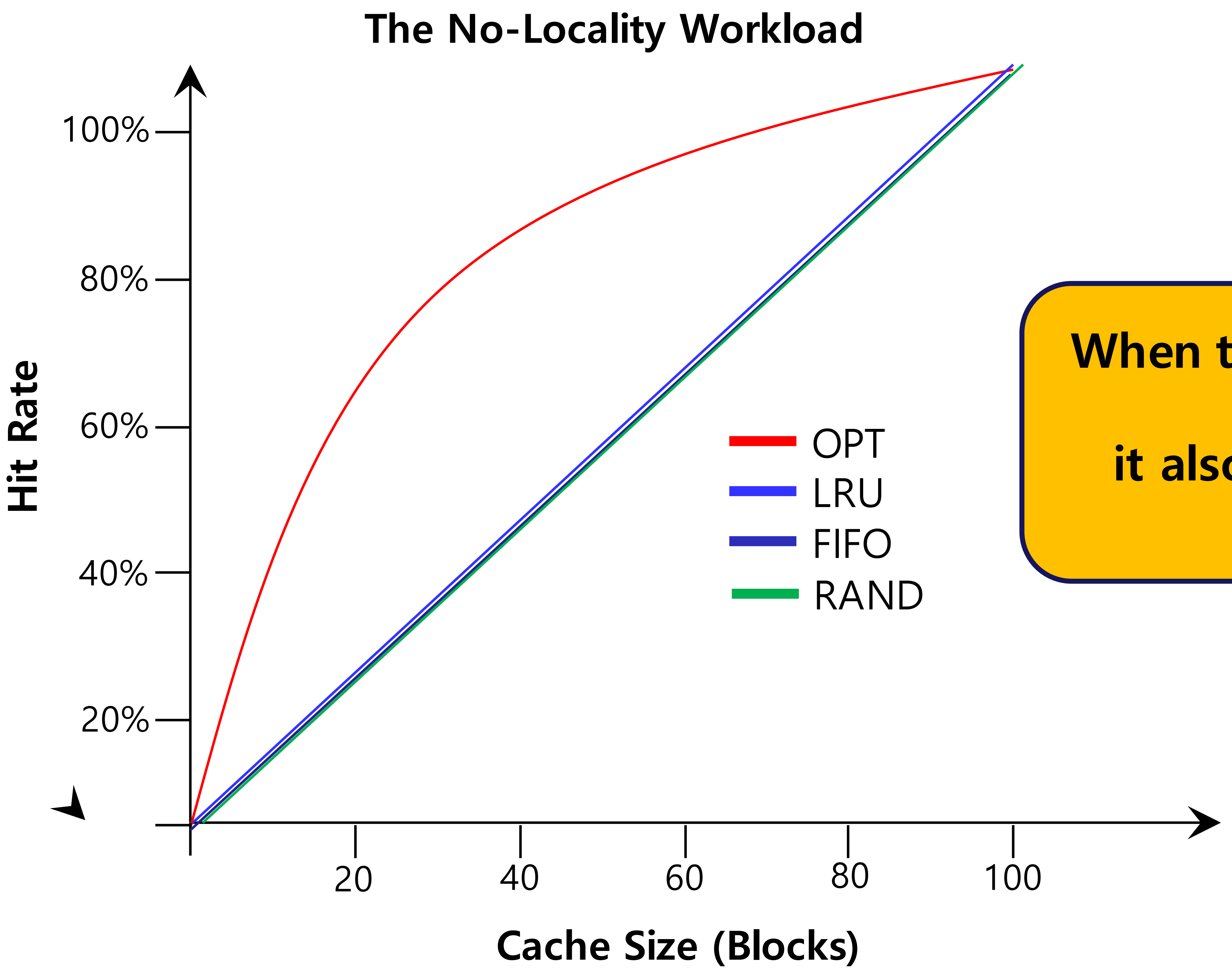
Reference Row

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 : No-Locality Workload

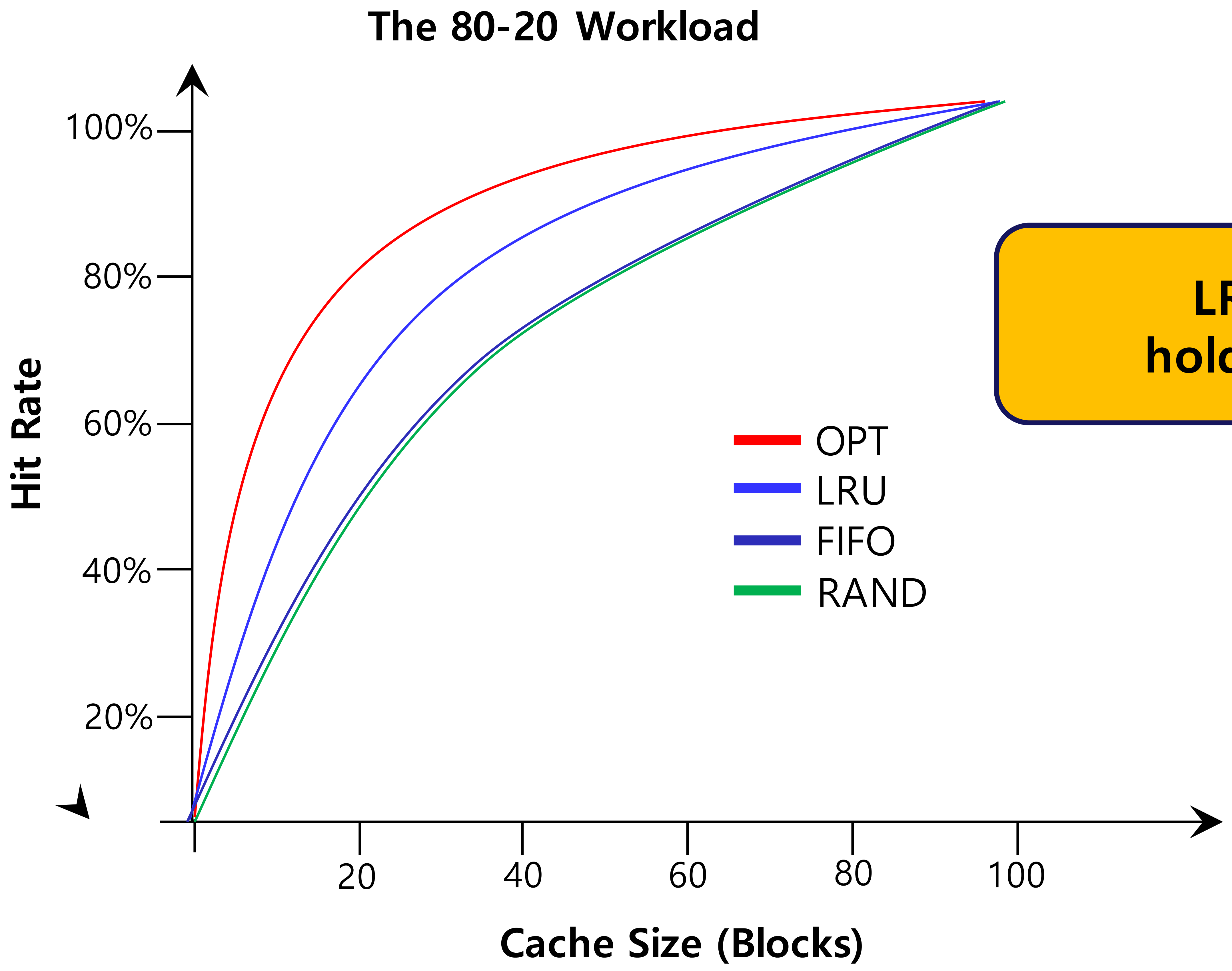
- Each reference is to a random page within the set of accessed pages
  - Workload accesses 100 unique pages over time
  - Choosing next page to refer to at random



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

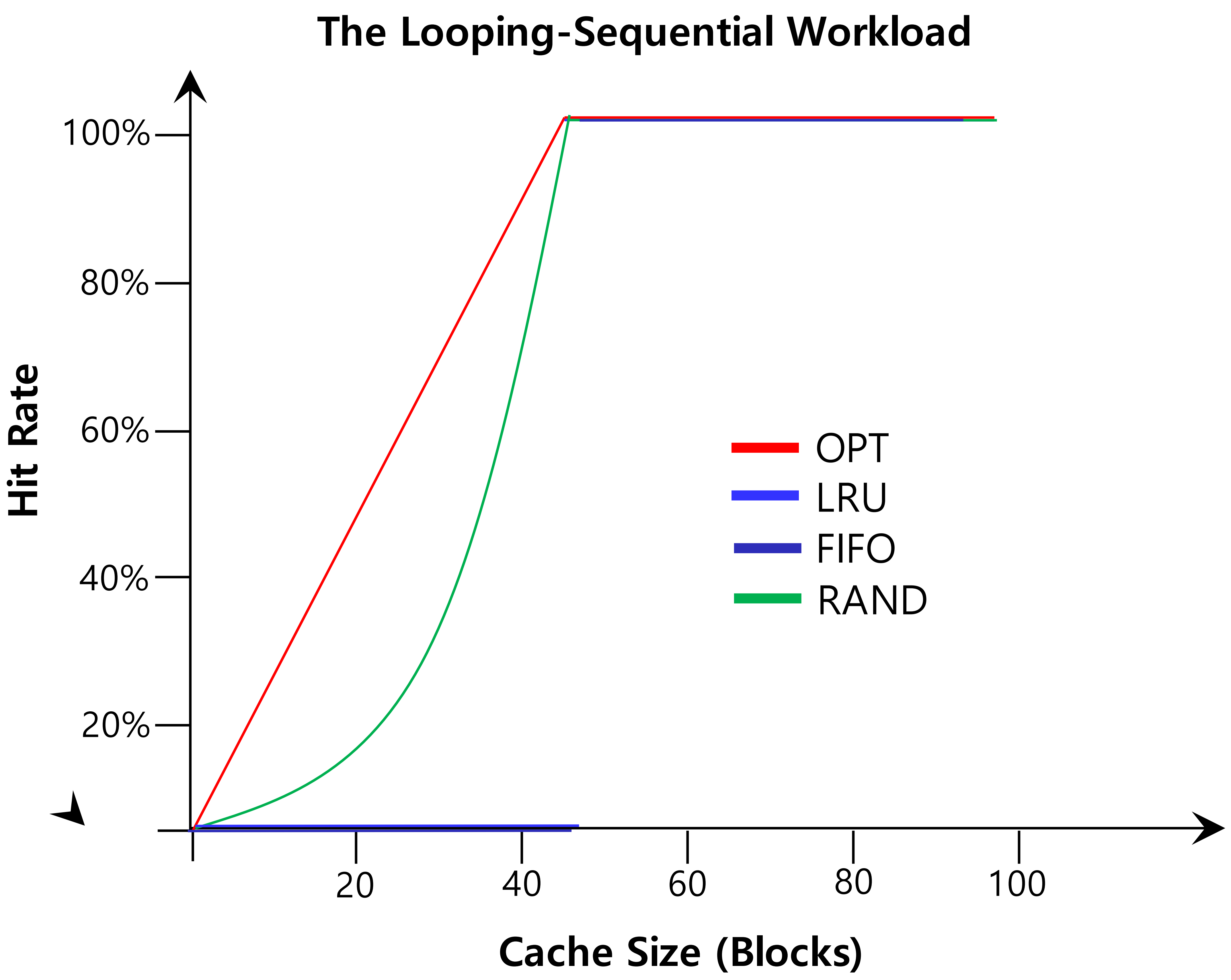
- Exhibits locality : 80% of references are made to 20% of pages
- Remaining 20% of references made to remaining 80% of pages



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, etc., up to page 49, and then we loop back, repeating those accesses
  - In total, 10,000 accesses to 50 unique pages



Implementing History-Based Algorithms

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

Approximating LRU

- How would we implement actual LRU?
    - Hardware has to act on every memory reference — update TLB, PTE
    - OS has to keep pages in some order or search big list of pages
    - Therefore, implementing pure LRU would be expensive

Approximating LRU

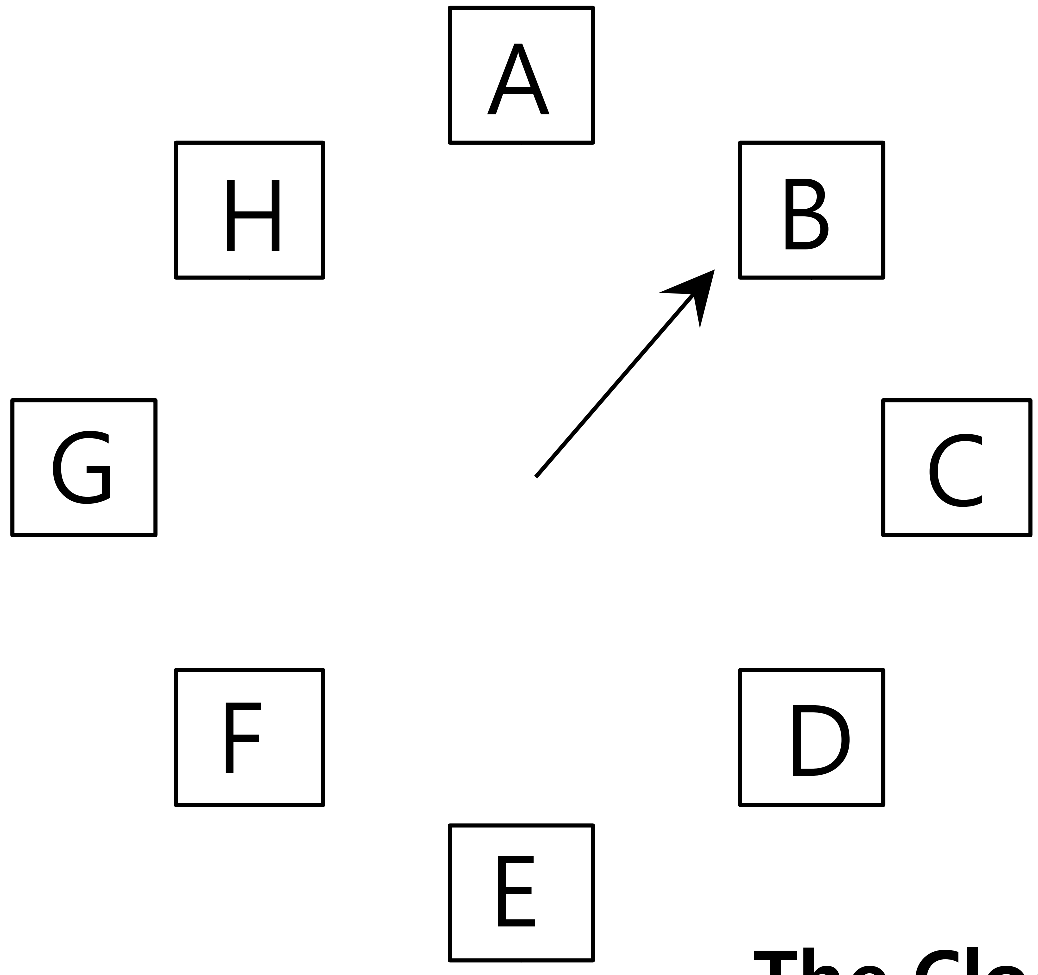
- Require some hardware support, in form of a **use bit**
  - Whenever page is referenced, use bit is set by hardware to 1
  - Hardware never clears the bit, though; that is responsibility of OS

Approximating LRU

- Clock algorithm - OS visits small number of pages
  - All pages of system arranged in circular list
  - A clock hand points to some particular page to begin with
  - Each page’s use bit examined once per trip around the “clock”

Clock Algorithm

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



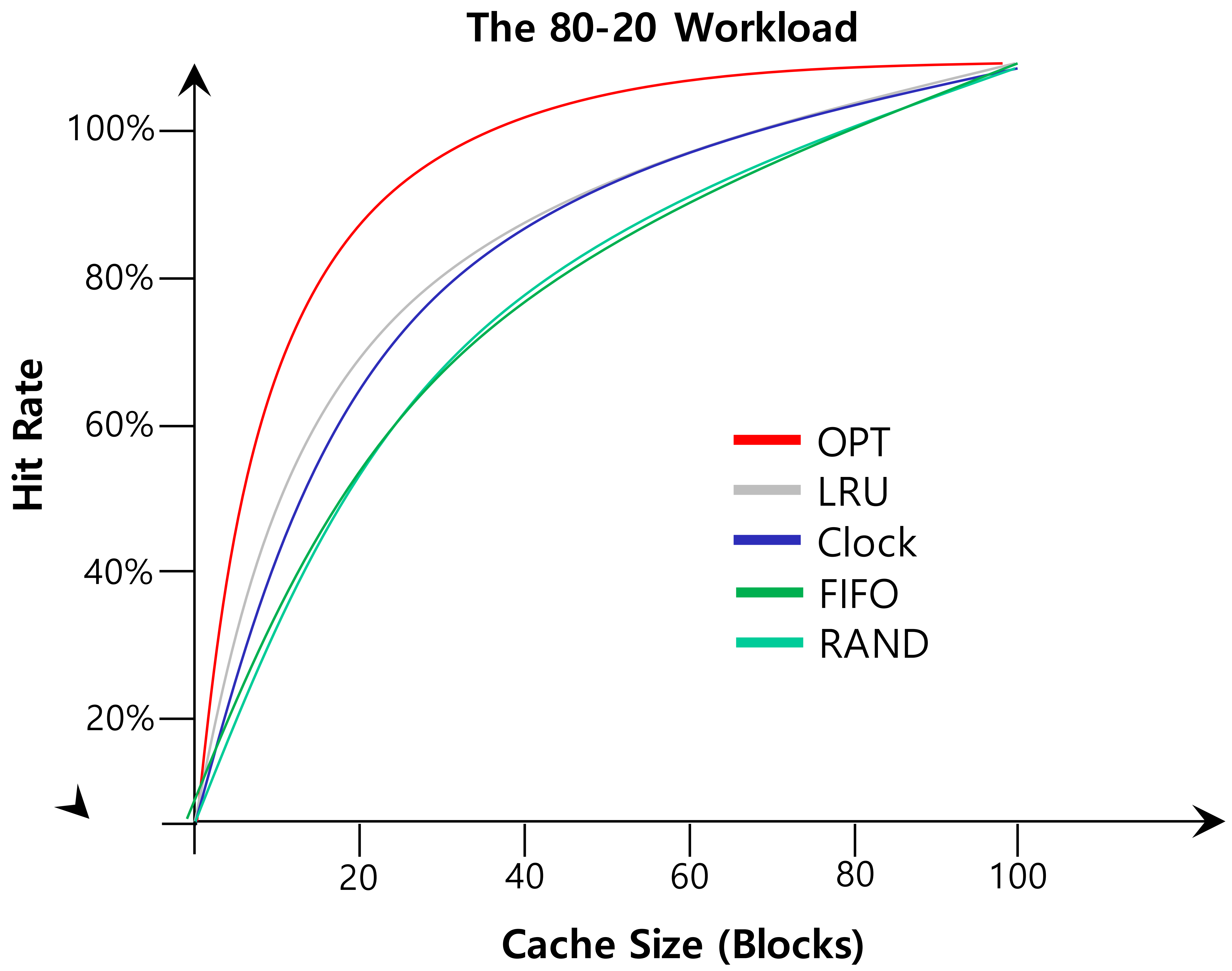
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

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, but it does better than approach that don't consider history at all



Considering Dirty Pages

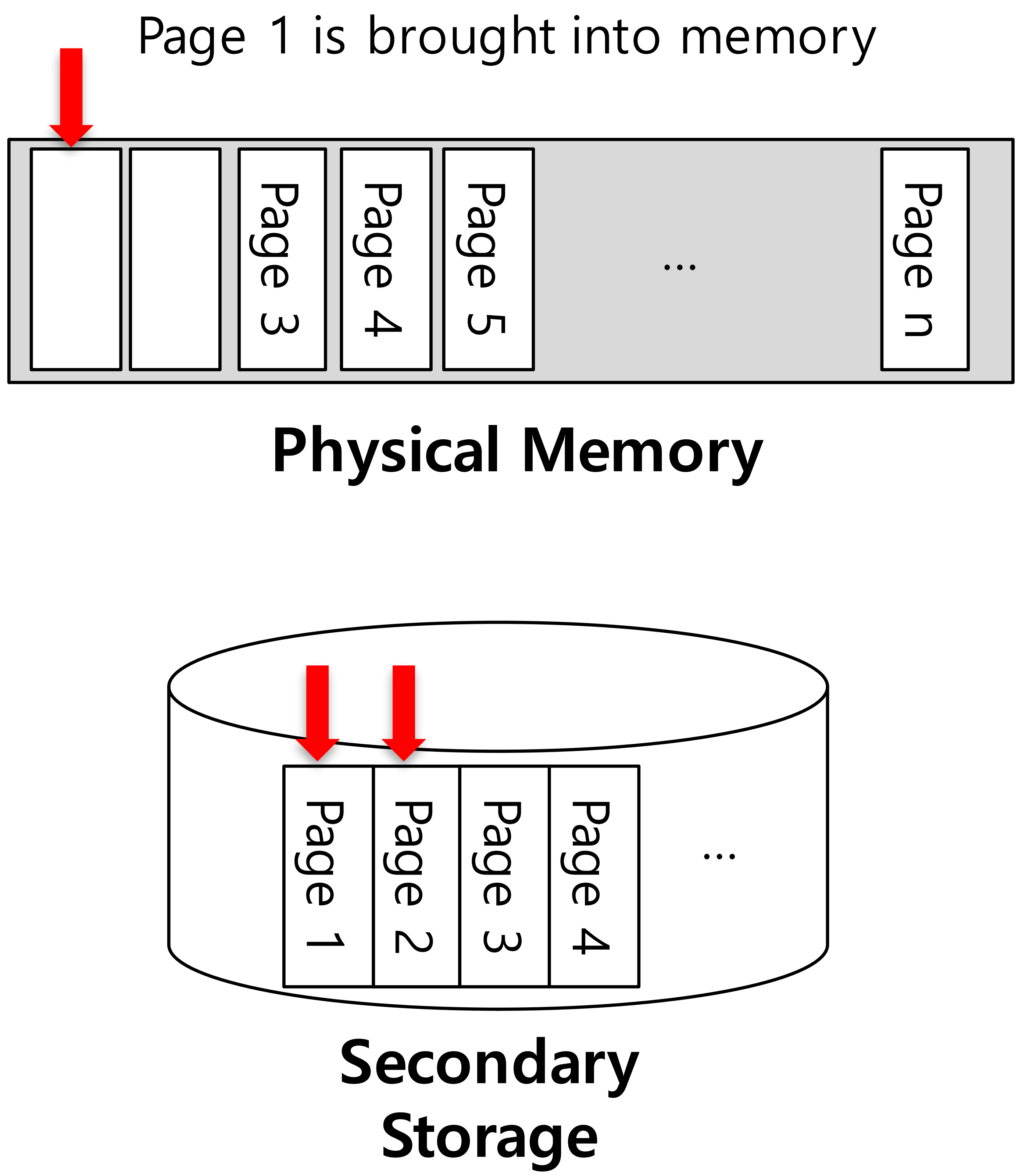
- Hardware includes a modified bit (or dirty bit)
    - Page has been modified and is thus dirty
        - Must be written back to disk to evict it
    - Page has not been modified — eviction is free

Page Selection Policy

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

Prefetching

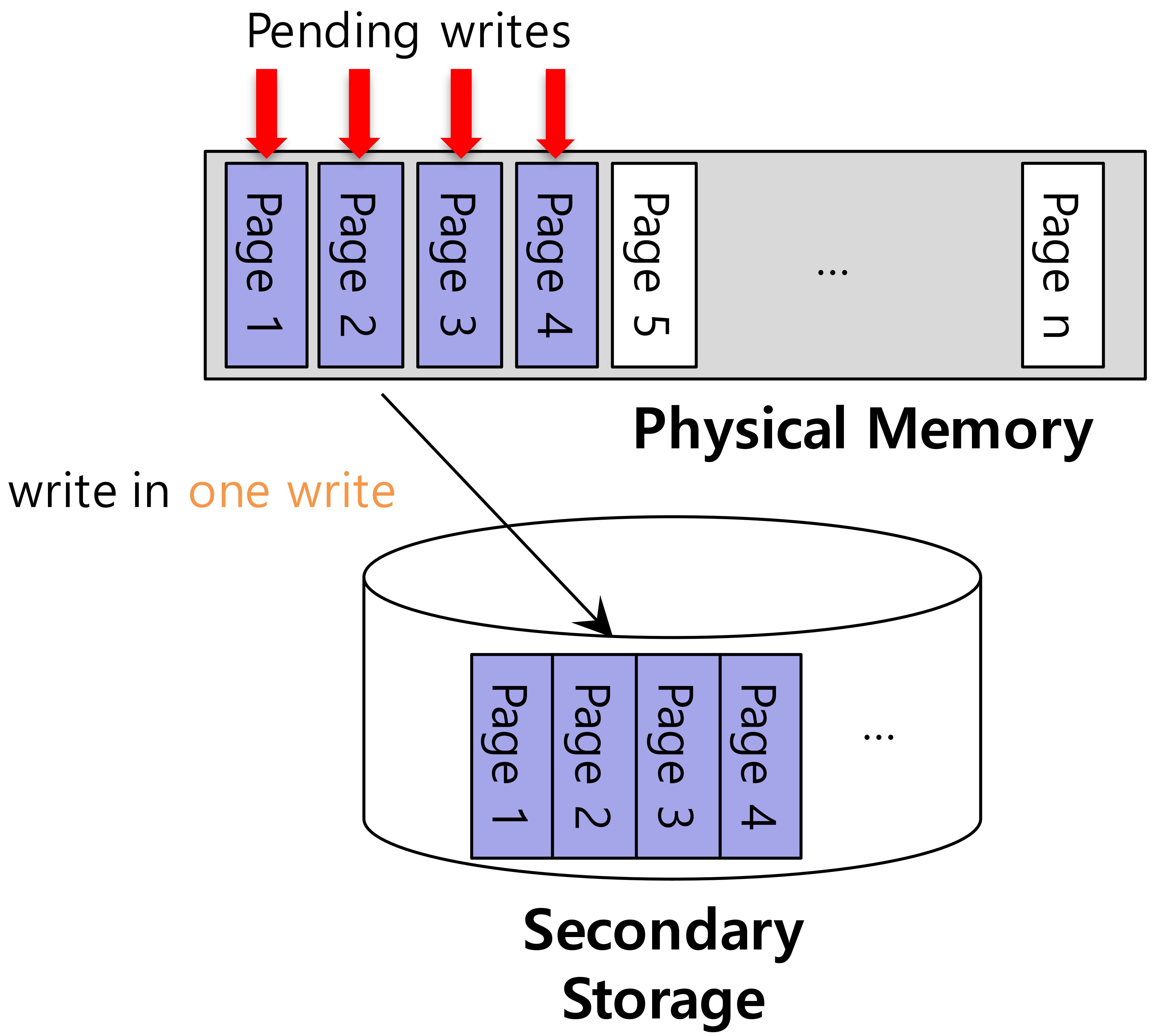
- OS guess that a page is about to be used, and thus bring it in ahead of time



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

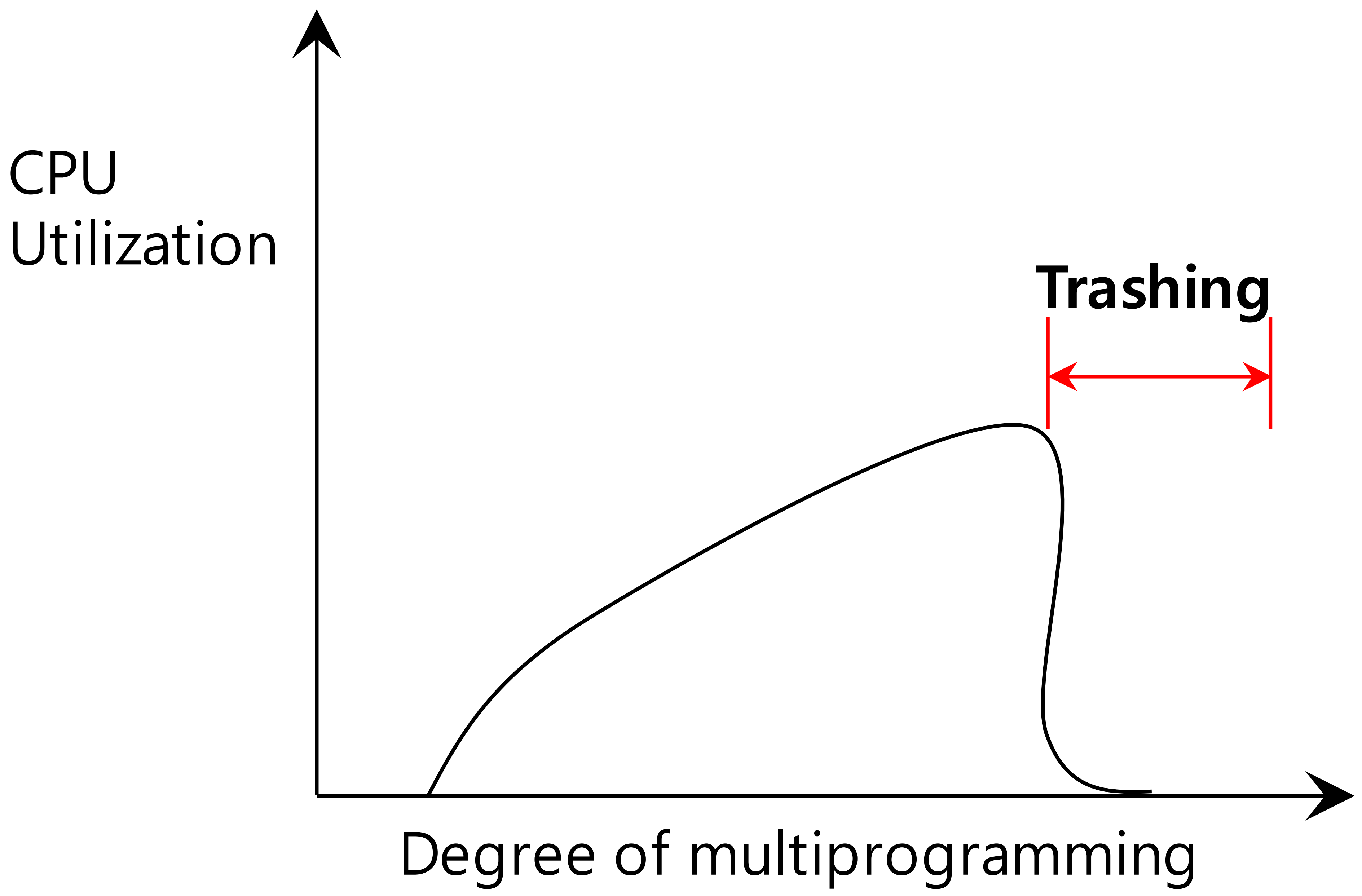
- Should OS allocate more address space than physical memory + swap space?

- What should we do when memory is oversubscribed

  - Memory demands of the set of running processes (the working set) exceeds the available physical memory

  - Decide not to run a subset of processes

  - Reduced set of process's working sets fit in memory



Memory-Mapped Files

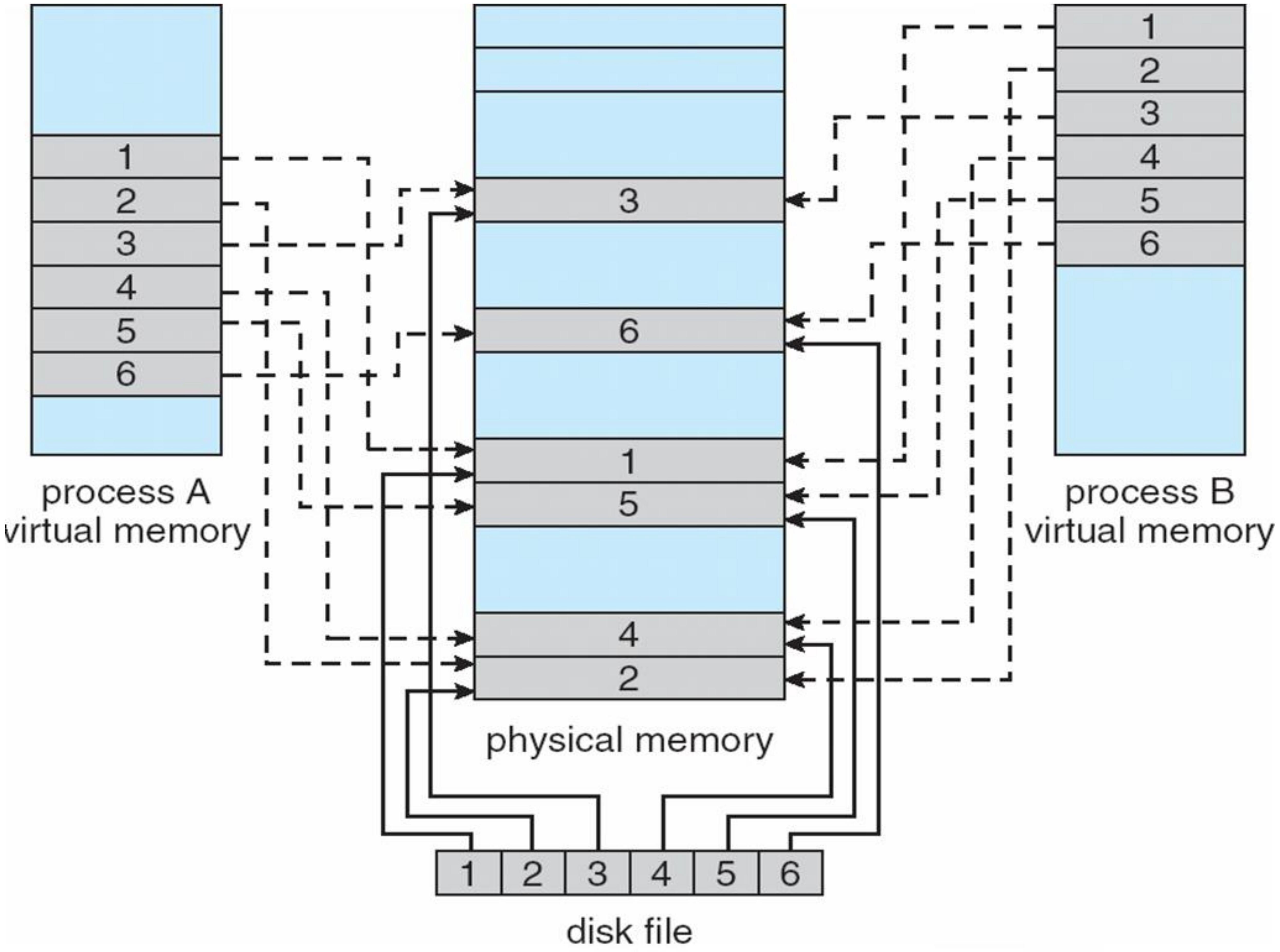
- Allow file I/O to be treated as routine memory access by mapping a disk block to a page in memory

- Simplifies file access by treating file I/O through direct memory access rather than read() and write() system calls

- File access is managed with demand paging

- Several processes may map the same file into memory as shared data

Memory Mapped Files



Key mmap mechanisms

- Valid bit used to demand page in file.
- Remaining bits can contain disk block number in an invalid PTE
- Dirty bit is important : how OS knows if process wrote to the page, so it needs to be written back to the file when it evicts the page

Example of Linux mmap

```
#define NUMINTS  1000
#define FILESIZE    NUMINTS * sizeof(int)
   fd = open(FILEPATH, O_RDWR | O_CREAT | O_TRUNC,
                  0600);
   result = lseek(fd, FILESIZE-1, SEEK_SET);
   result = write(fd, "", 1);
   map = mmap(0, FILESIZE, PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, 0);
   for (i = 1; i <=NUMINTS; ++i)  map[i] = 2 * i;
   munmap(map, FILESIZE);
   close(fd);
```

For next class

- Read pgs 159-171 : introduction to concurrency