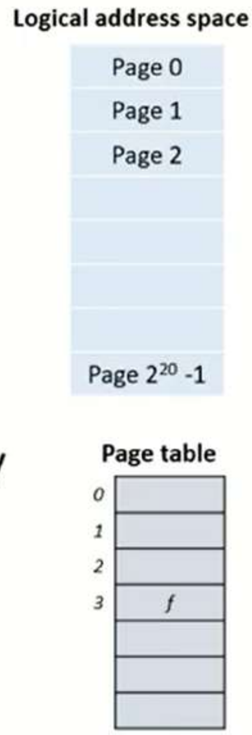
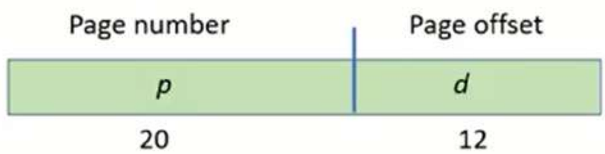
Page Table

- Consider a 32-bit logical address space
- Page size: 4 KB (2^{12})
- Number of pages = $2^{32} / 2^{12} = 2^{20}$
- Page table would have 2^{20} entries
- If each entry is 4 bytes:
  - Space for page table for each process = 2^{20} * 4 bytes = 4 MB
- Do not want to allocate space contiguously in main memory
- Solutions:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



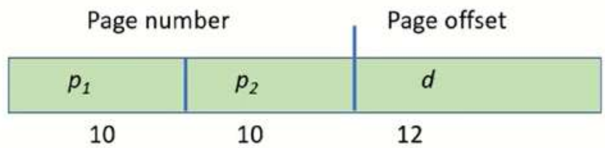
Hierarchical Page Tables

- Page the page table
  - Multi-level page table
- 32-bit logical address, 4K page size, is divided into:
  - Page number: 20 bits
  - Page offset: 12 bits

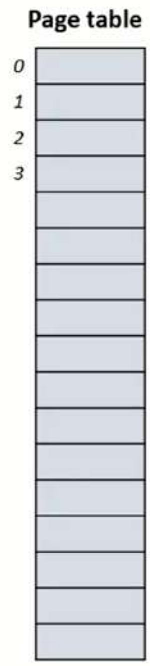


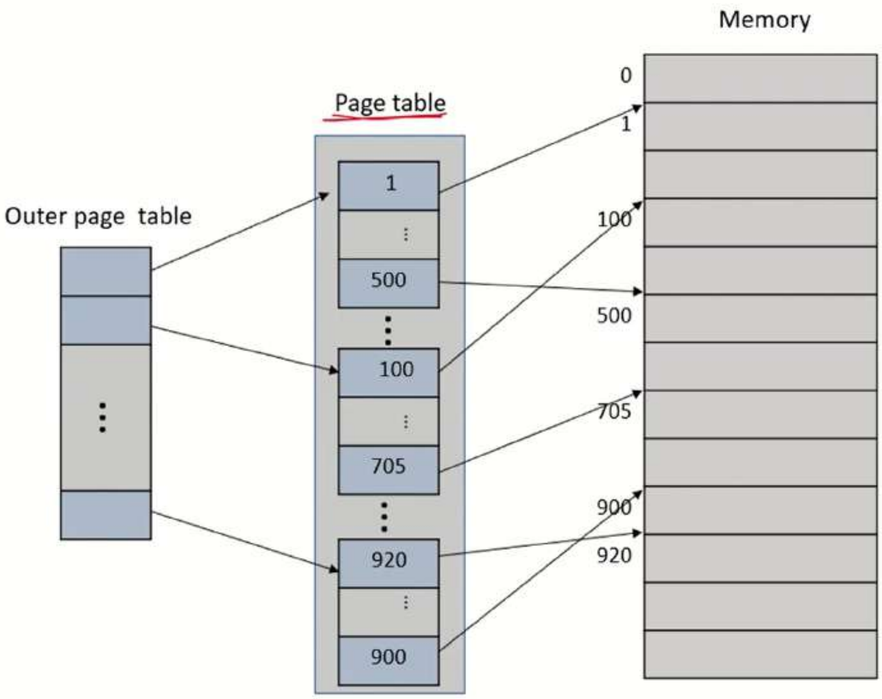
Hierarchical Page Tables

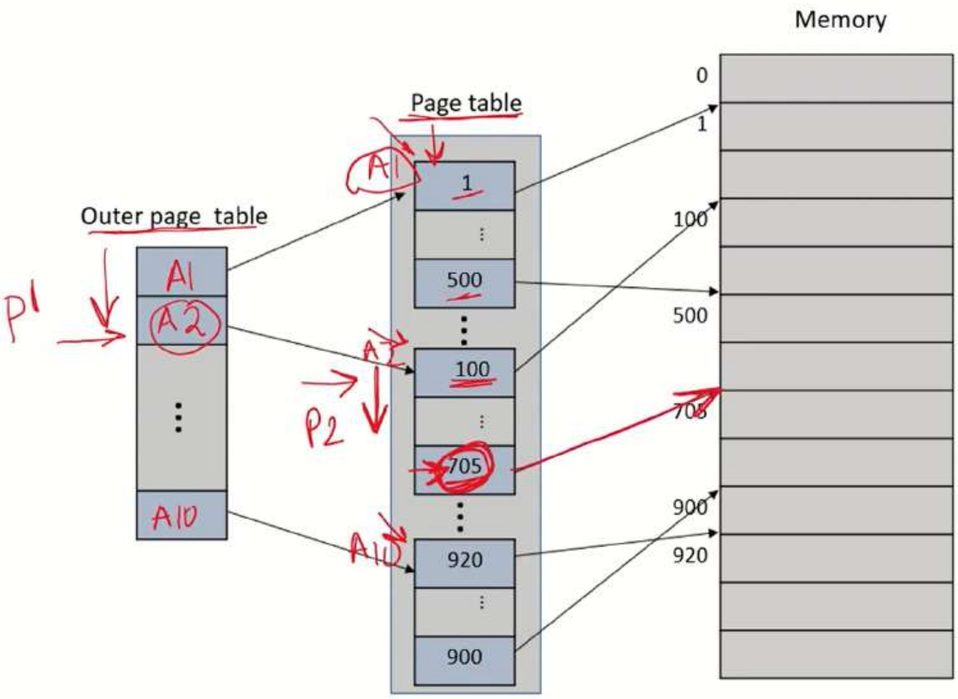
- Page size = 4 KB, size of each entry = 4 bytes
  - No. of entries of page table in one page = 4 KB / 4B = 2^{10}
- To page the page table, page number further divided into:
  - 10-bit page number
  - 10-bit page offset
- Thus, a logical address is as follows:

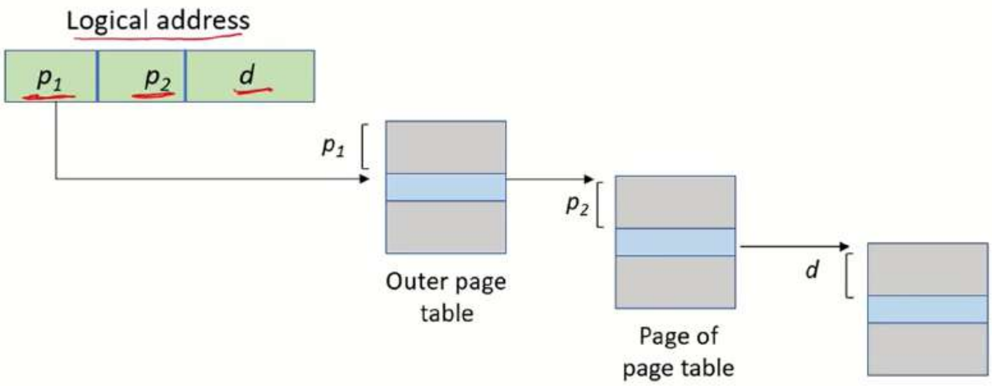


- p_1 is an index into outer page table
- p_2 is displacement within page of inner page table



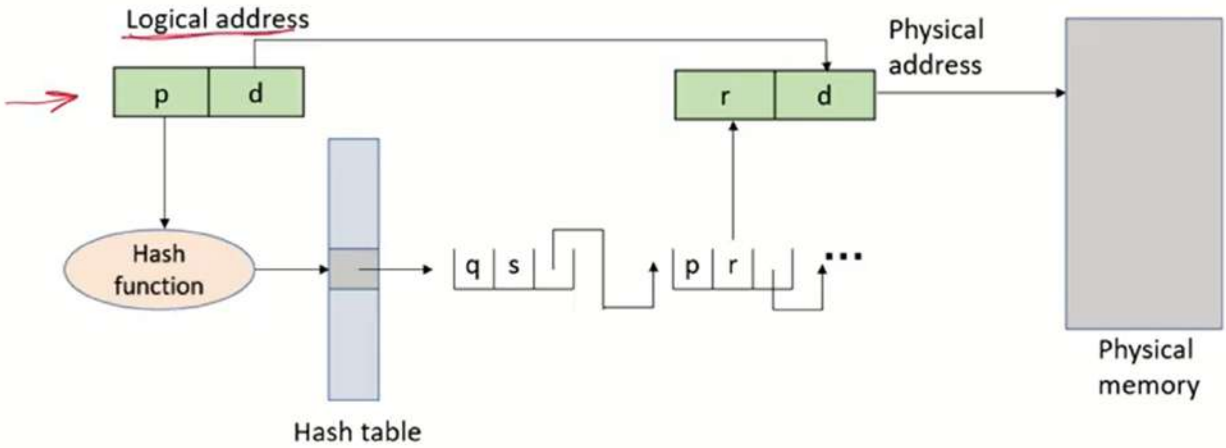

Memory
Page table
Outer page table
1
500
100
705
920
900
0
1
100
500
705
900
920

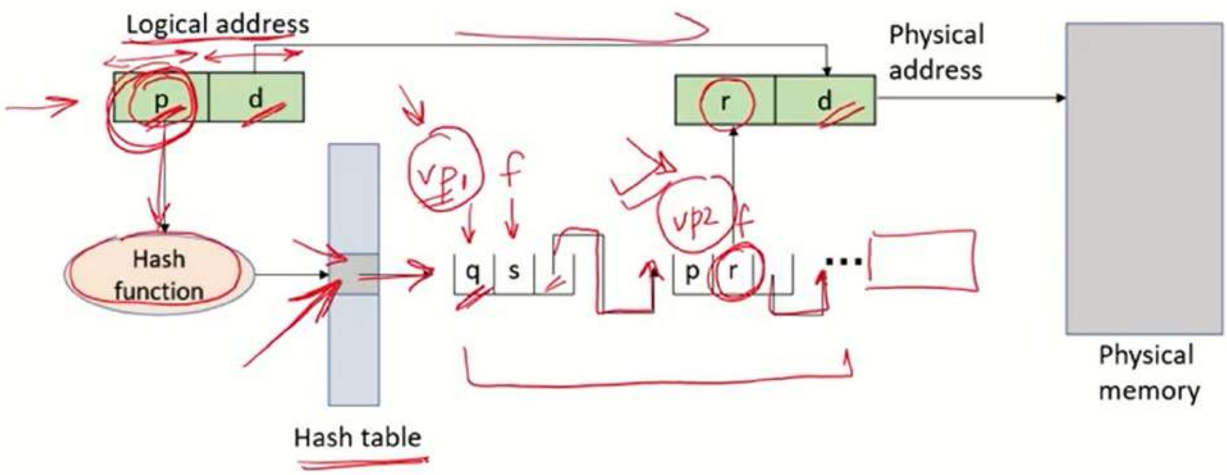

Memory
0
1
100
500
705
900
920
Page table
1
500
100
705
920
900
Outer page table
A1
A2
A10
P1
P2
A1
A2
A10


Logical address
p_1
p_2
d
p_1
p_2
d
Outer page table
Page of page table

Hashed Page Tables

- Common in address spaces > 32 bits
- Virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to same location
- Each element contains:
  - Virtual page number
  - Value of mapped page frame
  - Pointer to next element
- Virtual page numbers are compared in this chain to look for a match
  - If a match is found, corresponding physical frame is extracted


Logical address
p
d
Physical
address
r
d
Hash
function
q
s
p
r
...
Hash table
Physical
memory



Logical address
p
d
Physical address
r
d
Hash function
vp1
f
q
s
vp2
f
p
r
Hash table
Physical memory

Inverted Page Table

- Each process has its own page table with entry for each logical page

- Rather than each process having a page table and keeping track of all possible logical pages, track page frames
  - One entry for each frame of memory
  - Entry consists of virtual address of page stored in frame along with information about process that owns that page

- Only one page table in the system

Inverted Page Table

0	4
1	0
2	1
3	
4	
5	
6	
7	

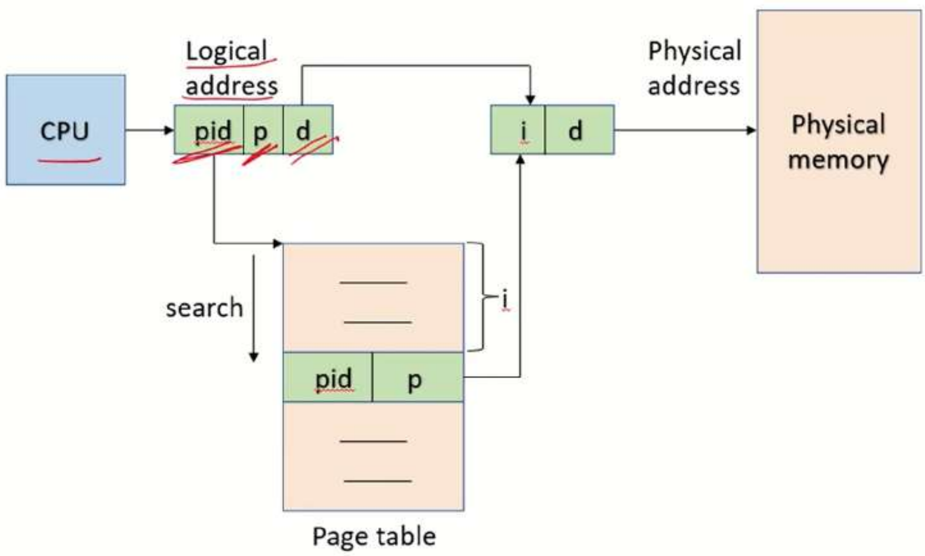
Page Table for P1

0	2
1	3
2	5
3	
4	
5	
6	
7	

Page Table for P2

	pid	Page no.
0	1	1
1	1	2
2	2	0
3	2	1
4	1	0
5	2	2

Inverted Page Table


CPU
Logical address
pid
p
d
i
d
Physical address
Physical memory
search
i
pid
p
Page table

Inverted Page Table

- Decreases memory needed to store each page table
  - But increases time needed to search table when a page reference occurs
- Use hash table to limit search to one — or at most a few — page-table entries
  - TLB can accelerate access

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Improves main memory utilization
  - Swapped-out processes stored in swap space
- Backing store – fast disk:
  - Large enough to accommodate copies of all memory images for all users
  - Must provide direct access to these memory images
- Swap out, swap in (Roll out, roll in):
  - Swapping variant used for priority-based scheduling algorithms
  - Lower-priority process swapped out so higher-priority process can be loaded and executed

Swapping

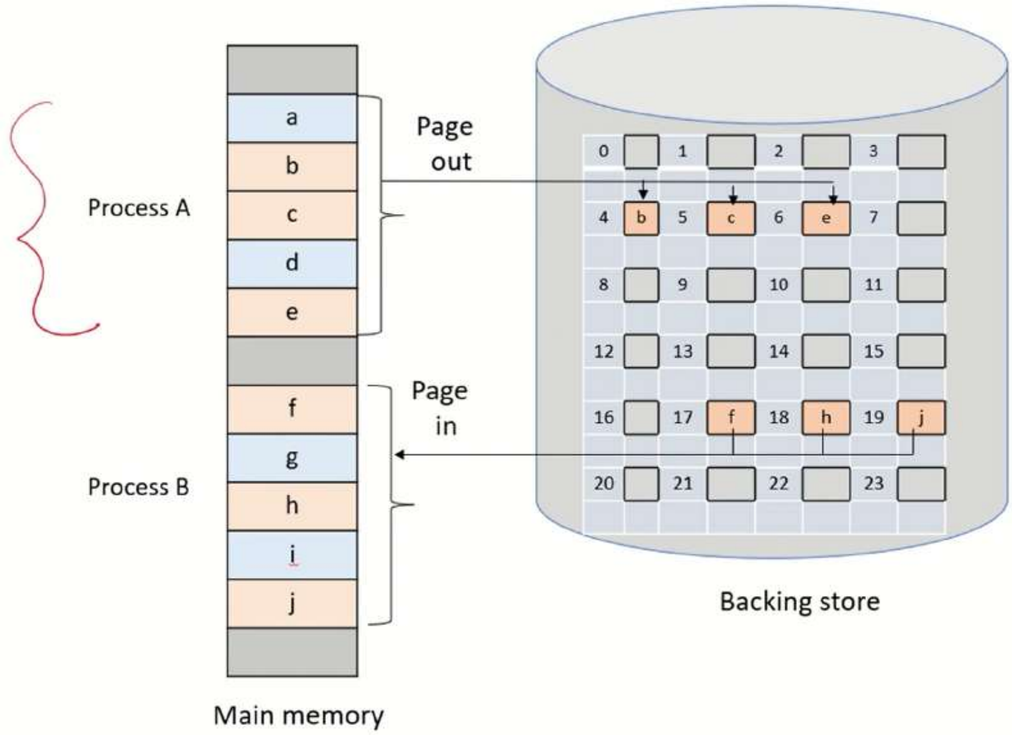
- Swapping makes possible for total physical address space of all processes to exceed real physical memory of system:
  - Increasing degree of multiprogramming in a system
- Does swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Swapping with Paging

- Standard swapping not used in modern operating systems
  - Swap only when free memory extremely low
- Most systems use a variation of swapping:
  - Pages of a process—rather than an entire process—can be swapped
  - A **page out** operation moves a page from memory to the backing store
  - Reverse process is known as a **page in**

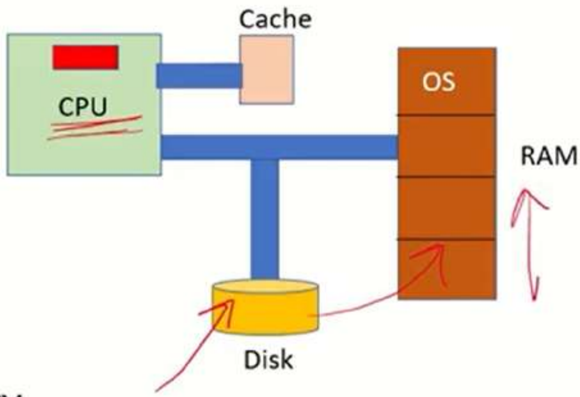
Swapping with Paging

- Standard swapping not used in modern operating systems
  - Swap only when free memory extremely low
- Most systems use a variation of swapping:
  - Pages of a process—rather than an entire process—can be swapped
  - A **page out** operation moves a page from memory to the backing store
  - Reverse process is known as a **page in**


Process A
Process B
a
b
c
d
e
f
g
h
i
j
Page
out
Page
in
0
1
2
3
4
b
5
c
6
e
7
8
9
10
11
12
13
14
15
16
17
f
18
h
19
j
20
21
22
23
Backing store
Main memory

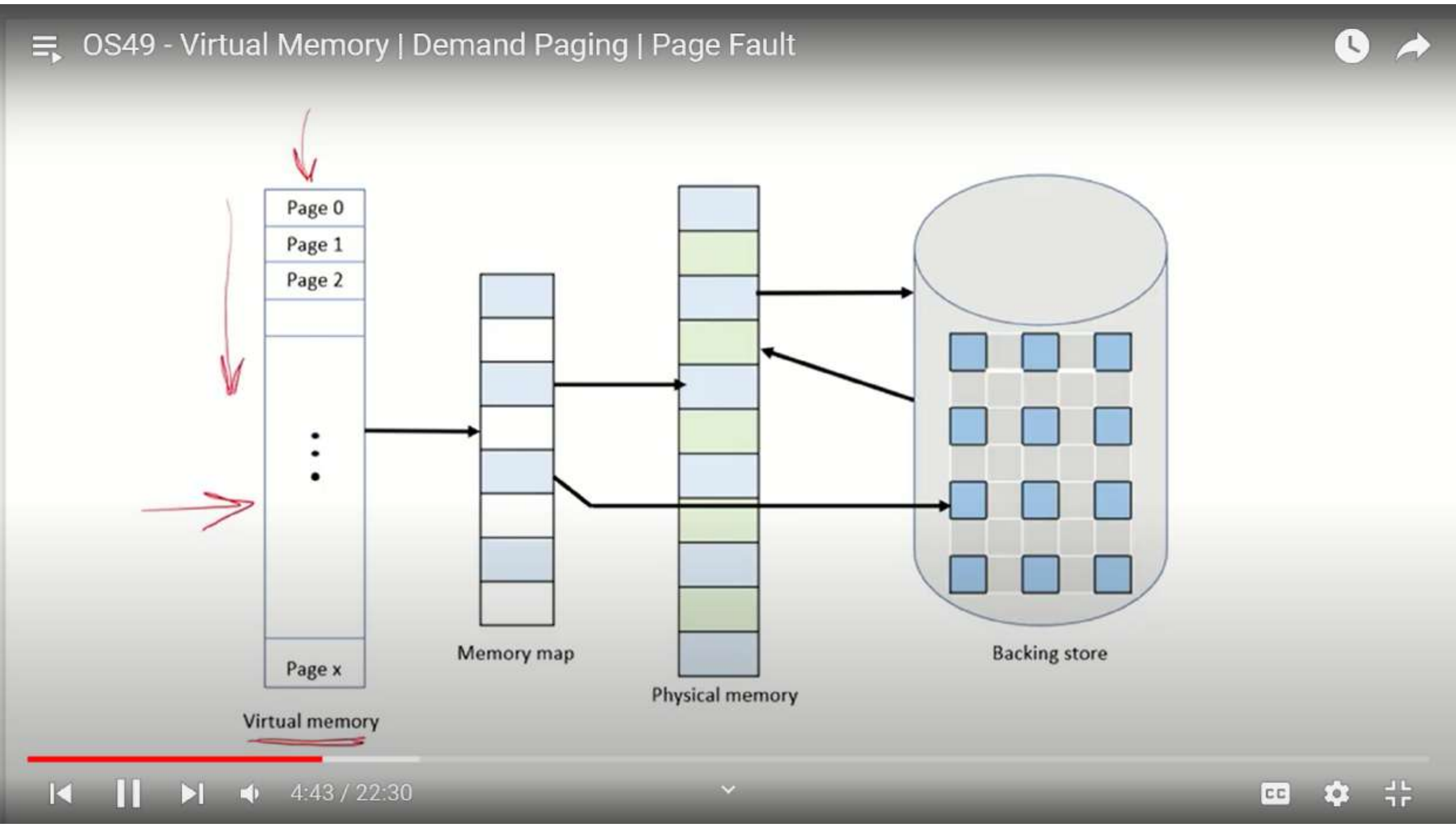
Virtual memory

- Instructions must be in physical memory to be executed
- It limits size of a program to size of physical memory
- In many cases, entire program is not needed
    - Codes to handle error conditions
    - Static memory allocation to arrays
    - Certain features of programs
- Entire program may not be needed in memory at same time

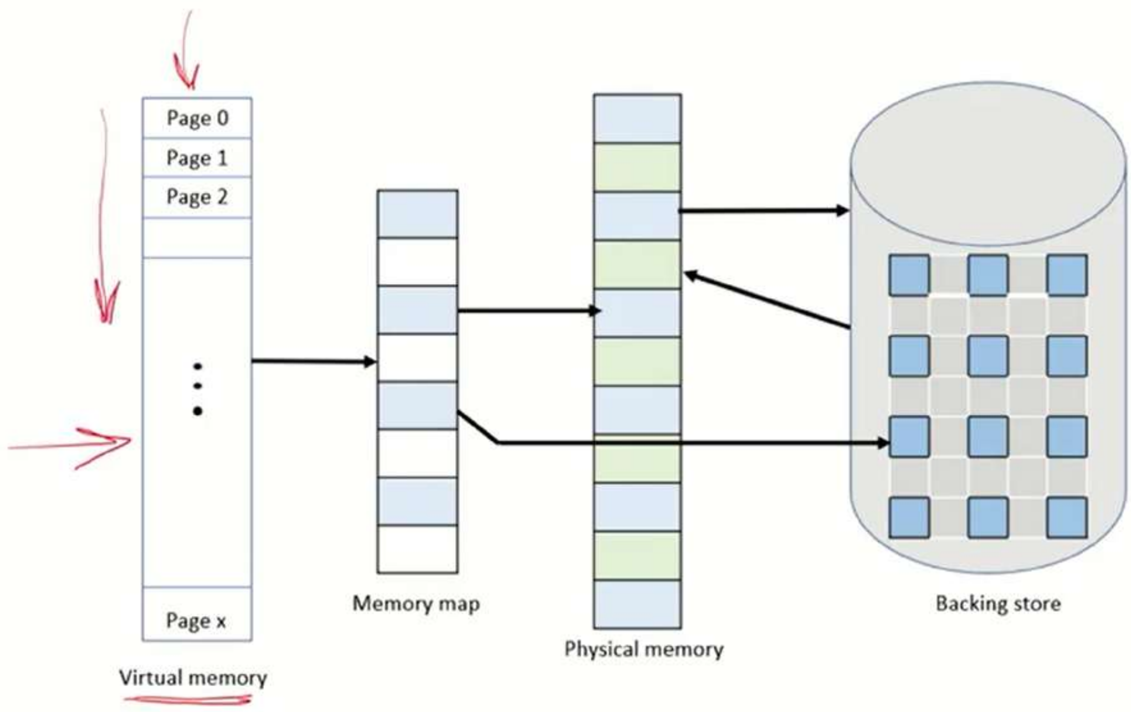


Virtual memory

- Ability to execute a program only partially in memory can have many benefits:
  - Program no longer constrained by amount of available physical memory
  - If each program takes less physical memory, more programs can run at the same time - increase in CPU utilization and throughput
  - Less I/O needed to load or swap portions of programs into memory
- **Virtual memory** involves separation of logical memory as perceived by developers from physical memory
  - Allows a large virtual memory to be provided for programmers even if smaller physical memory is available



OS49 - Virtual Memory | Demand Paging | Page Fault
Page 0
Page 1
Page 2
Page x
Virtual memory
Memory map
Physical memory
Backing store
4:43 / 22:30


Page 0
Page 1
Page 2
Page x
Virtual memory
Memory map
Physical memory
Backing store

Virtual memory

- Separation of user logical memory from physical memory
  - Only part of program needs to be in memory for execution
  - Logical address space can be much larger than physical address space
  - Allows address space to be shared by several processes
  - More programs running concurrently
- Logical view of how process is stored in memory
  - Contiguous addresses, consisting of pages
- Physical memory organized in page frames
  - MMU must map logical to physical

Demand Paging

- Can bring entire process into memory at load time
  - May not initially need entire program in memory
- **OR** bring a page into memory only when it is needed (**demand**ed)
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - Invalid reference $\Rightarrow$ abort
  - Not-in-memory $\Rightarrow$ bring to memory

Demand Paging

- Basic concept - load a page in memory only when it is needed
    - While a process is executing - some pages in memory, and some in secondary storage
    - Need hardware support to distinguish between the two
- With each page table entry, **valid–invalid bit** is associated
    - ***v*****:** in-memory – **memory resident**
    - ***i*****:** not-in-memory
    - Initially valid–invalid bit set to *i* on all entries
- During MMU address translation, if valid–invalid bit is *i*, this results in **page fault**

	Frame #	*v*/*i* bit
0		
1	5	*v*
2	3	*v*
3	10	*v*
4		*i*
	. . .	
		i
		i

Page table

Demand Paging

- Basic concept - load a page in memory only when it is needed
  - While a process is executing - some pages in memory, and some in secondary storage
  - Need hardware support to distinguish between the two
- With each page table entry, **valid–invalid bit** is associated
  - ***v*:** in-memory – **memory resident**
  - ***i*:** not-in-memory
  - Initially valid–invalid bit set to *i* on all entries
- During MMU address translation, if valid–invalid bit is *i*, this results in **page fault**

	Frame #	*v*/*i* bit
0		
1	5	*v*
2	3	*v*
3	10	*v*
4		*i*
	. . .	
		i
		i

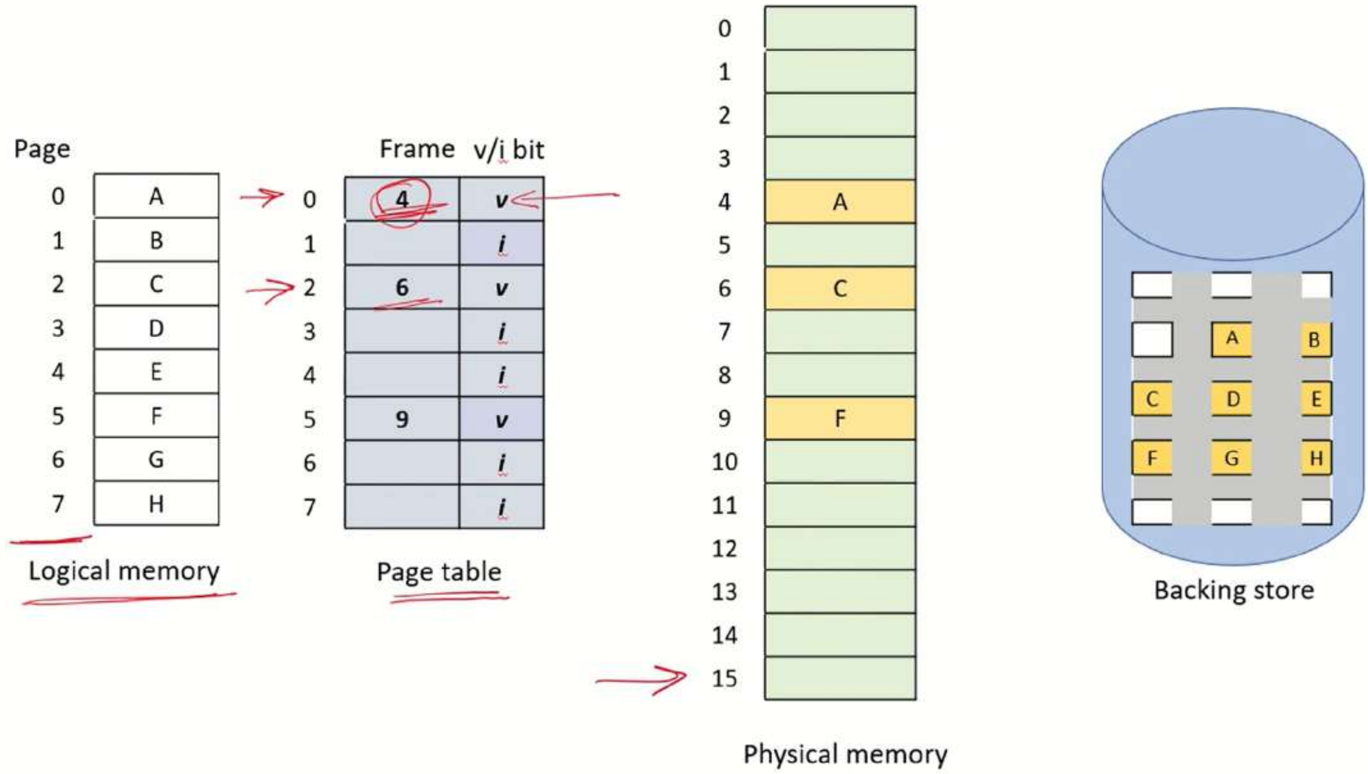
Page table

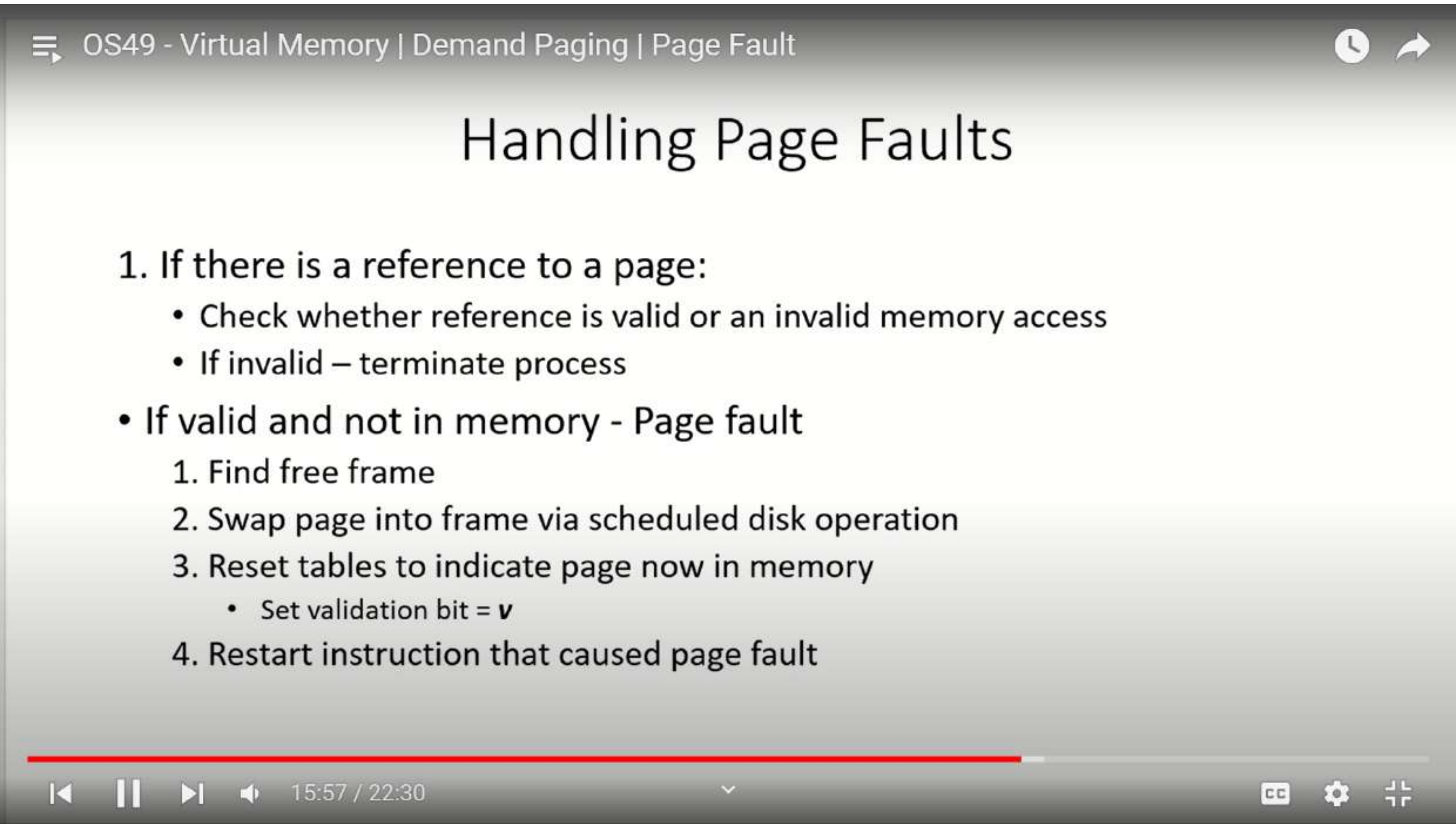
Demand Paging

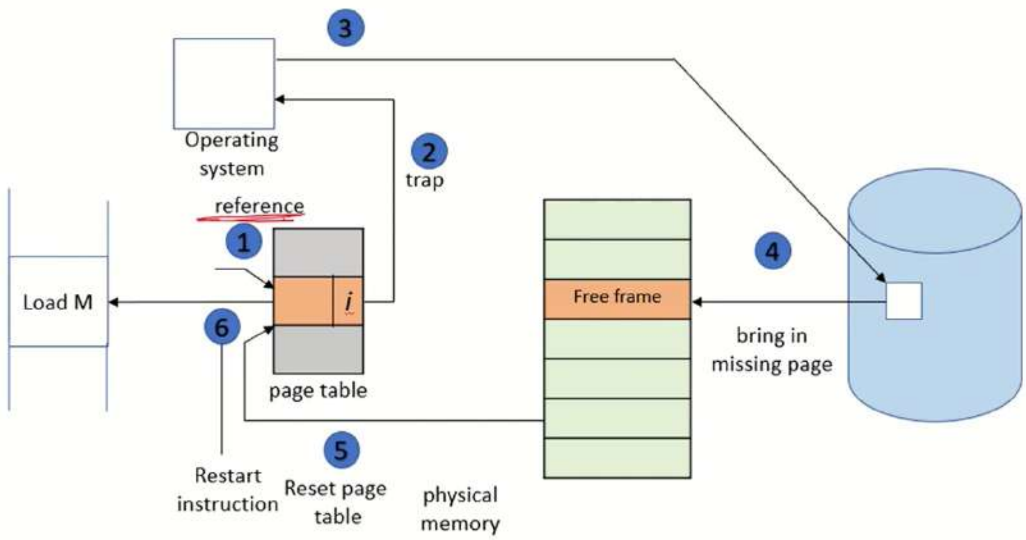
- Basic concept - load a page in memory only when it is needed
  - While a process is executing - some pages in memory, and some in secondary storage
  - Need hardware support to distinguish between the two
- With each page table entry, **valid–invalid bit** is associated
  - ***v***: in-memory – **memory resident**
  - ***i***: not-in-memory
  - Initially valid–invalid bit set to *i* on all entries
- During MMU address translation, if valid–invalid bit is *i*, this results in **page fault**

	Frame #	*v*/*i* bit
0		
1	5	*v*
2	3	*v*
3	10	*v*
4		*i*
	...	
		i
		i

Page table


Page
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
Logical memory
Frame v/i bit
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
Page table
0
1
2
3
4 A
5
6 C
7
8
9 F
10
11
12
13
14
15
Physical memory
A B
C D E
F G H
Backing store


OS49 - Virtual Memory | Demand Paging | Page Fault
Handling Page Faults
1. If there is a reference to a page:
• Check whether reference is valid or an invalid memory access
• If invalid – terminate process
• If valid and not in memory - Page fault
1. Find free frame
2. Swap page into frame via scheduled disk operation
3. Reset tables to indicate page now in memory
• Set validation bit = v
4. Restart instruction that caused page fault
15:57 / 22:30


3
Operating system
2
trap
reference
1
Load M
i
6
Free frame
4
bring in missing page
page table
5
Restart instruction
Reset page table
physical memory

Free-Frame List

- When page fault occurs – OS must bring desired page from secondary storage into main memory
- Most OS maintain a **free-frame list** -- a pool of free frames for satisfying such requests
  - Typically allocate free frames using **zero-fill-on-demand** technique - content of frames erased before being allocated
- When system starts up, all available memory placed on free-frame list
- As free frames are requested, size of free-frame list shrinks

Performance of Demand Paging

- If no page faults: effective access time = memory access time
- If page fault: first read relevant page from secondary storage and then access desired address
- Let:
  - Memory-access time: *ma*
  - Probability of a page fault: p ($0 \le p \le 1$)
  - Effective access time = $(1 - p) \times ma + p \times$ page fault service time
- Ex.: ma = 200 ns, page fault service time = 8 ms

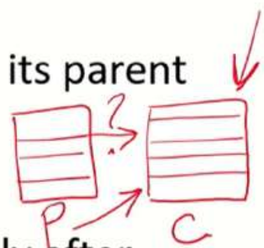
  Effective access time = $(1 - p) \times 200 + p \times 8000000 = 200 + 7999800p$
- If one out of 1,000 accesses causes page fault, effective access time is 8.2 μs
  - Slowing down of system by a factor of 40

Copy-on-Write (CoW)

- *fork()* system call creates child process that is duplicate of its parent
- Copy of parent's address space created for child
  - Duplicating pages belonging to parent
- Many child processes invoke *exec()* system call immediately after creation - copying of parent's address space may be unnecessary
- **Copy-on-write** – allows parent and child processes initially to share same pages
  - Shared pages marked as copy-on-write pages
  - Only pages that can be modified marked as copy-on-write
  - Pages that cannot be modified – ex. pages containing executable code - can be shared by parent and child

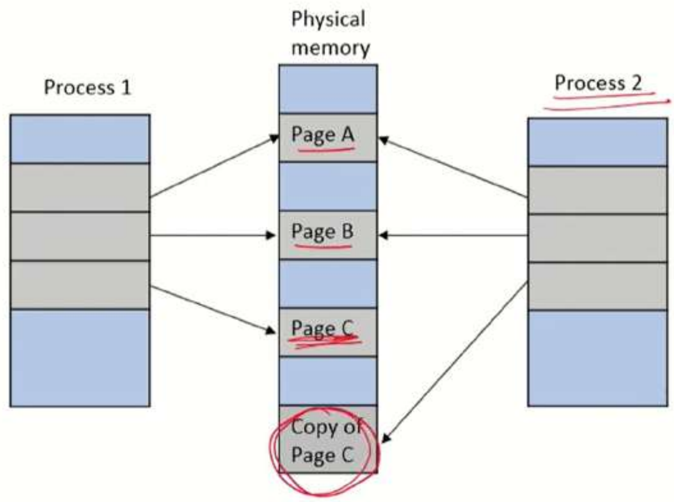
Copy-on-Write (CoW)

- *fork()* system call creates child process that is duplicate of its parent
- Copy of parent's address space created for child
  - Duplicating pages belonging to parent
- Many child processes invoke *exec()* system call immediately after creation - copying of parent's address space may be unnecessary
- **Copy-on-write** – allows parent and child processes initially to share same pages
  - Shared pages marked as copy-on-write pages
  - Only pages that can be modified marked as copy-on-write
  - Pages that cannot be modified – ex. pages containing executable code - can be shared by parent and child



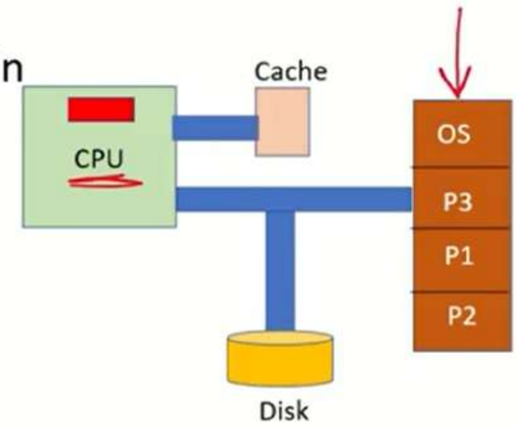
Copy-on-Write (CoW)

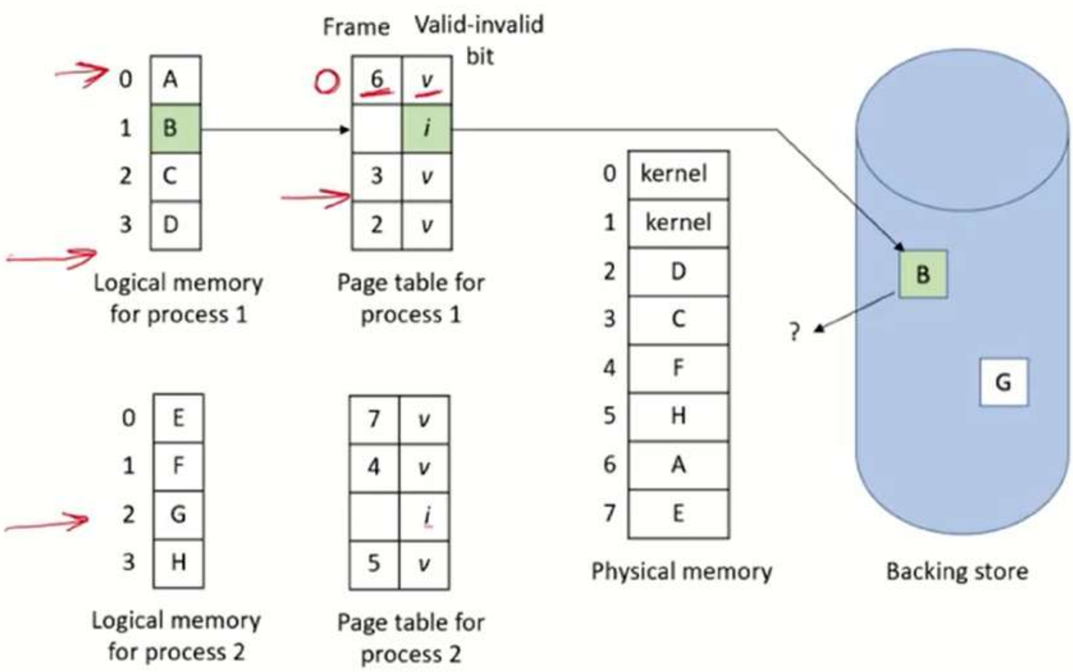
- If either process writes to a shared page:
  - A copy of shared page is created
  - Unmodified pages shared by parent and child processes
- Allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution


Physical memory
Process 1
Process 2
Page A
Page B
Page C
Copy of Page C

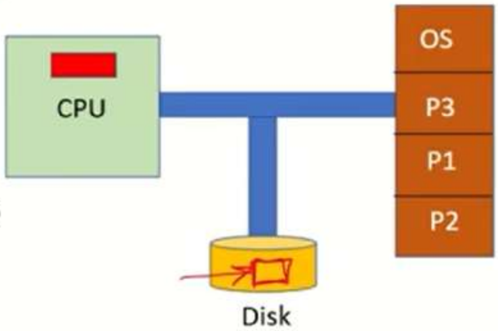
Page Replacement

- When a **page fault** occurs, OS must bring desired page from secondary storage into main memory in a **free frame**
- What happens if there is no free frame?
  - Used up by process pages, kernel, I/O buffers, etc.
- **Page replacement** – find some page in memory, currently not in use and free it




Frame
Valid-invalid bit
0
1
2
3
A
B
C
D
6
v
i
3
v
2
v
Logical memory for process 1
Page table for process 1
0
1
2
3
E
F
G
H
7
v
4
v
i
5
v
Logical memory for process 2
Page table for process 2
0 kernel
1 kernel
2 D
3 C
4 F
5 H
6 A
7 E
Physical memory
B
?
G
Backing store

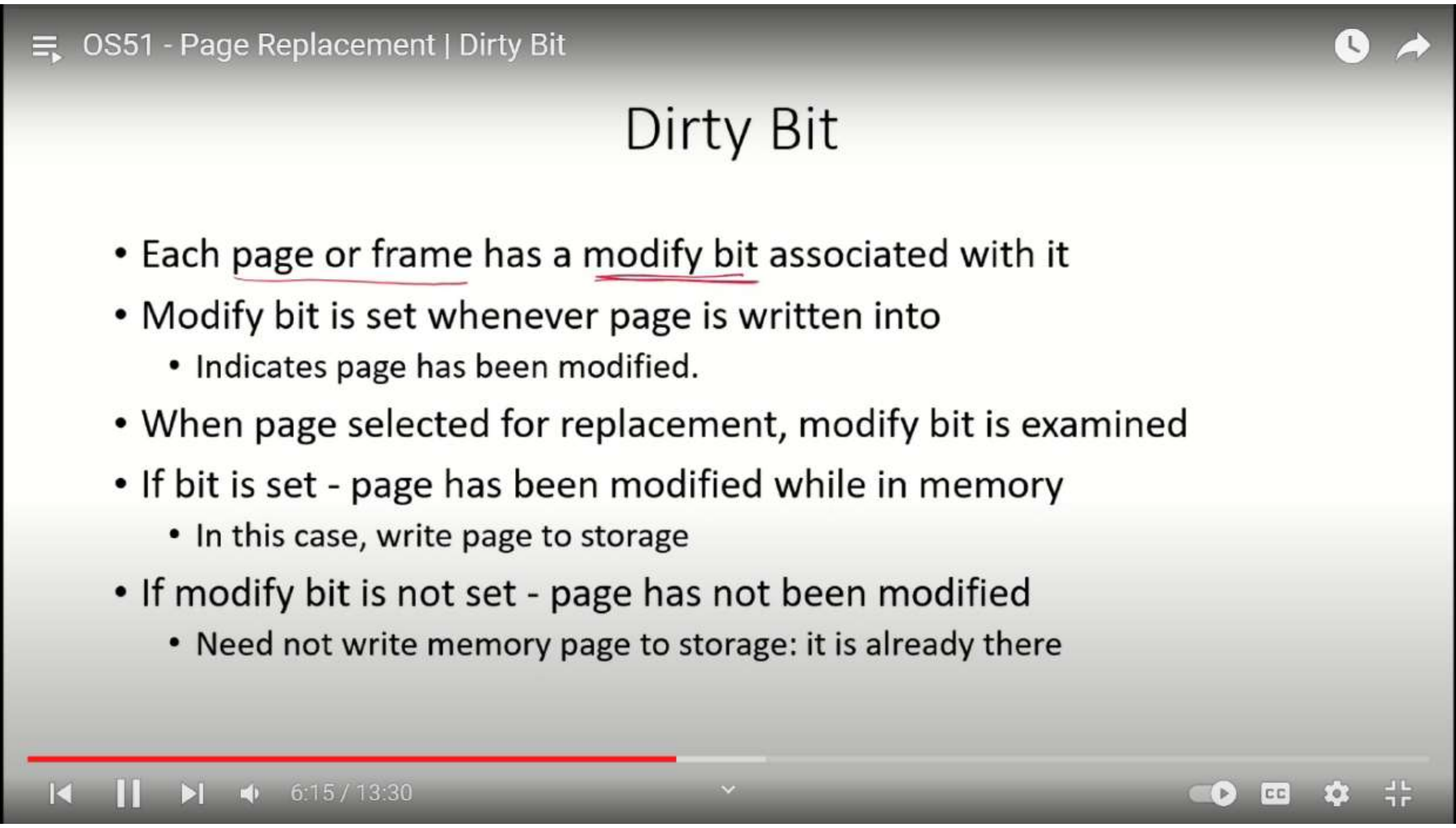
Page Fault Service Routine

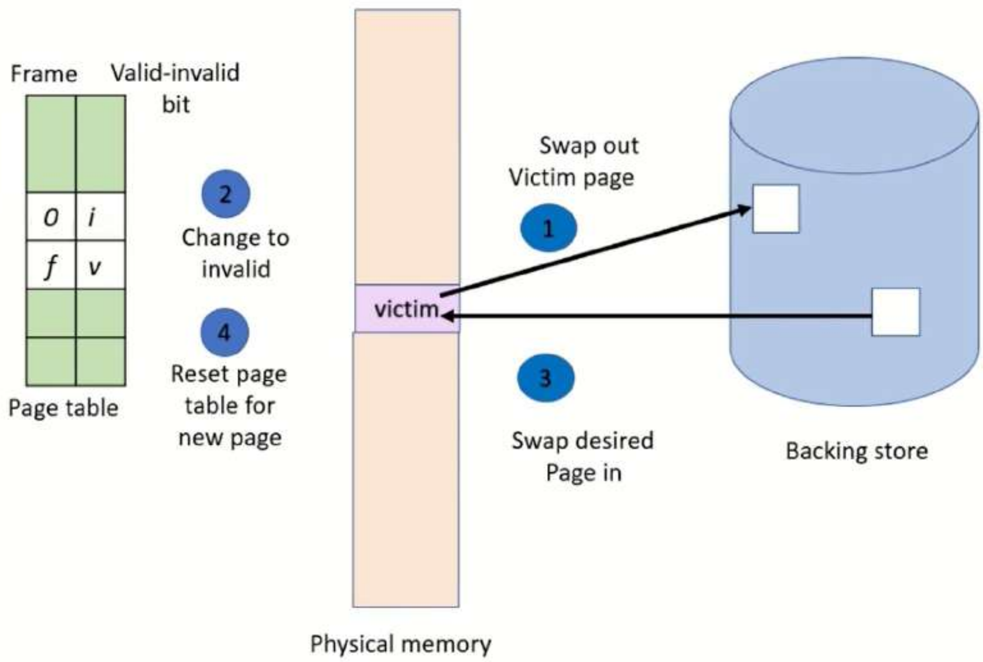


1. Find location of desired page on disk
2. Find a free frame:
   - If free frame available, use it
   - If no free frame: use page replacement algorithm to select **victim frame**
3. Bring desired page into (newly) free frame
   - Update page and frame tables
4. Continue process by restarting instruction that caused the trap

If no free frame: 2 page transfers for page fault

- Increases effective access time
- Can reduce overhead by using a **dirty** bit (or modify bit)


OS51 - Page Replacement | Dirty Bit
Dirty Bit
• Each page or frame has a modify bit associated with it
• Modify bit is set whenever page is written into
• Indicates page has been modified.
• When page selected for replacement, modify bit is examined
• If bit is set - page has been modified while in memory
• In this case, write page to storage
• If modify bit is not set - page has not been modified
• Need not write memory page to storage: it is already there
6:15 / 13:30



Frame
Valid-invalid bit
o
i
f
v
Page table
2
Change to invalid
4
Reset page table for new page
victim
Physical memory
Swap out Victim page
1
3
Swap desired Page in
Backing store

Dirty Bit

- Technique also applies to read-only pages (ex., pages of binary code)
  - Such pages cannot be modified
  - Can be discarded when desired
- Significantly reduces time required to service a page fault
  - Reduces I/O time by one-half if page has not been modified
- many different page-replacement algorithms
- How do we select a particular replacement algorithm?
  - Evaluate an algorithm by running it on a particular string of memory references and computing number of page faults
  - String of memory references is called a **reference string**