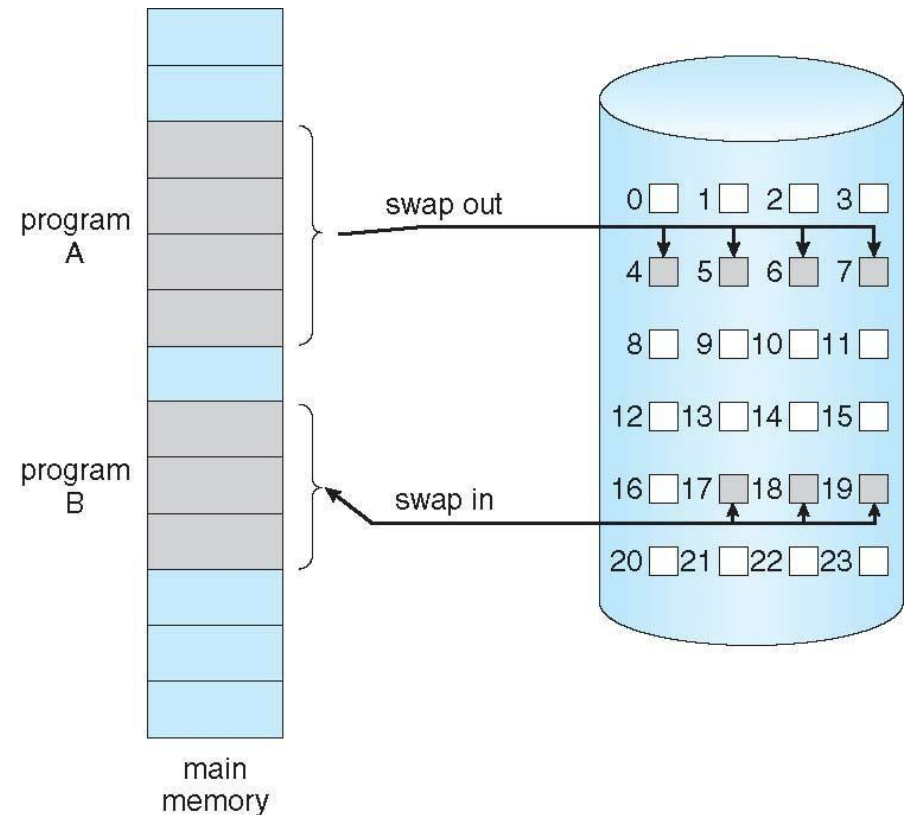




# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a

**pager**





# Basic Concepts



- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code



# Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v**  $\Rightarrow$  in-memory – **memory resident**, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

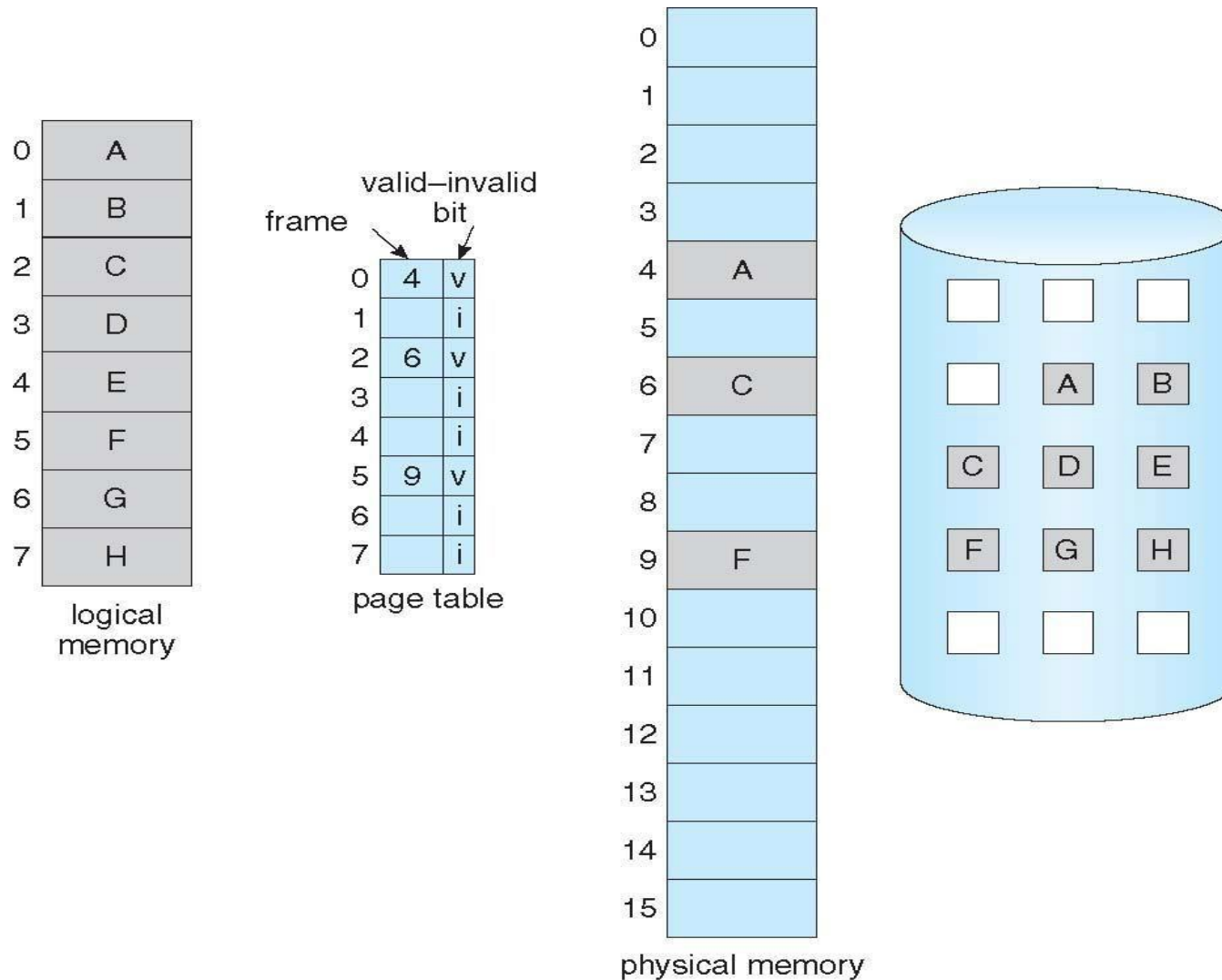
Frame #	valid-invalid bit
	<b>v</b>
	<b>v</b>
	<b>v</b>
	<b>i</b>
...	
	<b>i</b>
	<b>i</b>

page table

During MMU address translation, if valid–invalid bit in page table entry is **i**  $\Rightarrow$  page fault



# Page Table When Some Pages Are Not in Main Memory





# Page Fault



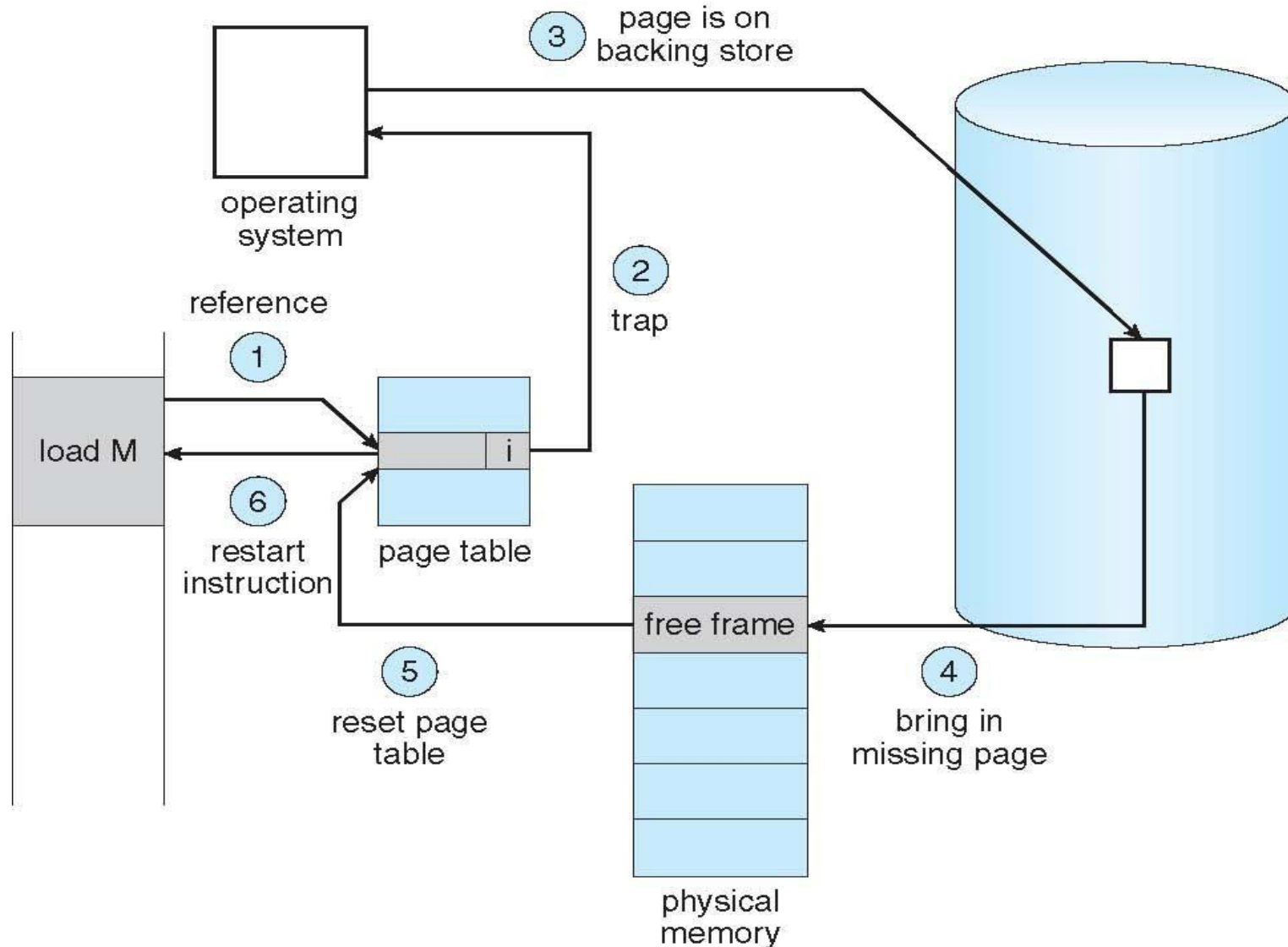
- If there is a reference to a page, first reference to that page will trap to operating system:

## page fault

1. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory Set validation bit = **v**
5. Restart the instruction that caused the page fault



# Steps in Handling a Page Fault





# Aspects of Demand Paging



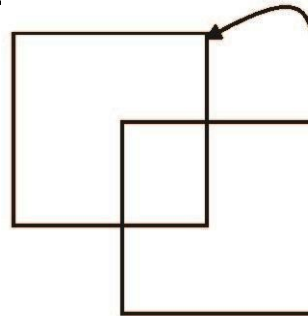
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple 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**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart



# Instruction Restart

- Consider an instruction that could access several different locations

– block move



- auto increment/decrement location
- Restart the whole operation?
  - What if source and destination overlap?





# Performance of Demand Paging

- Stages in Demand Paging (worse case)
  1. Trap to the operating system
  2. Save the user registers and process state
  3. Determine that the interrupt was a page fault
  4. Check that the page reference was legal and determine the 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 the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame



# Performance of Demand Paging

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



# Performance of Demand Paging (Cont.)



- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate  $0 \leq p \leq 1$ 
  - if  $p = 0$  no page faults
  - if  $p = 1$ , every reference is a fault
- Effective Access Time (EAT)  
$$\text{EAT} = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$



# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$   
 $= (1 - p) \times 200 + p \times 8,000,000$   
 $= 200 + p \times 7,999,800$
- If one access out of 1,000 causes a page fault, then  
EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

- If want performance degradation < 10 percent
  - $220 > 200 + 7,999,800 \times p$   
 $20 > 7,999,800 \times p$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses



# Demand Paging Optimizations



- Swap space I/O faster than file system I/O even if on the 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 BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)



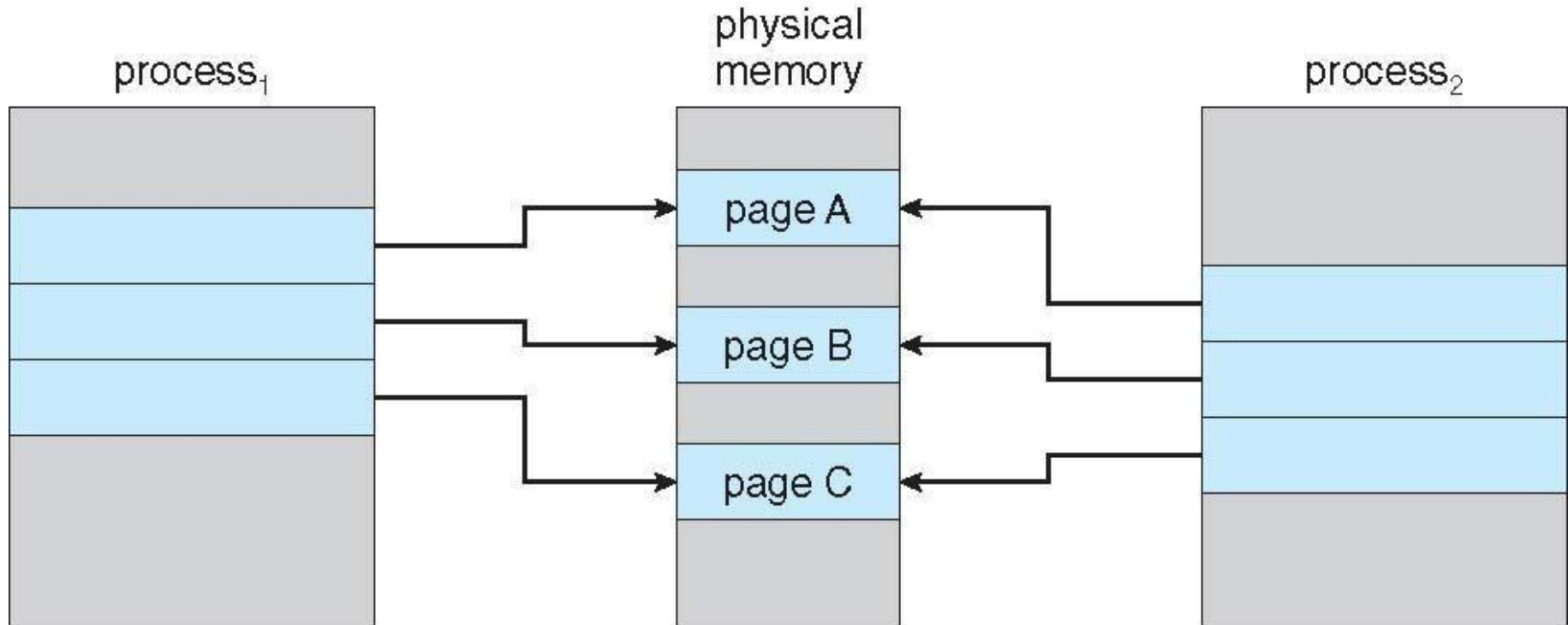
# Copy-on-Write



- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW 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 on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

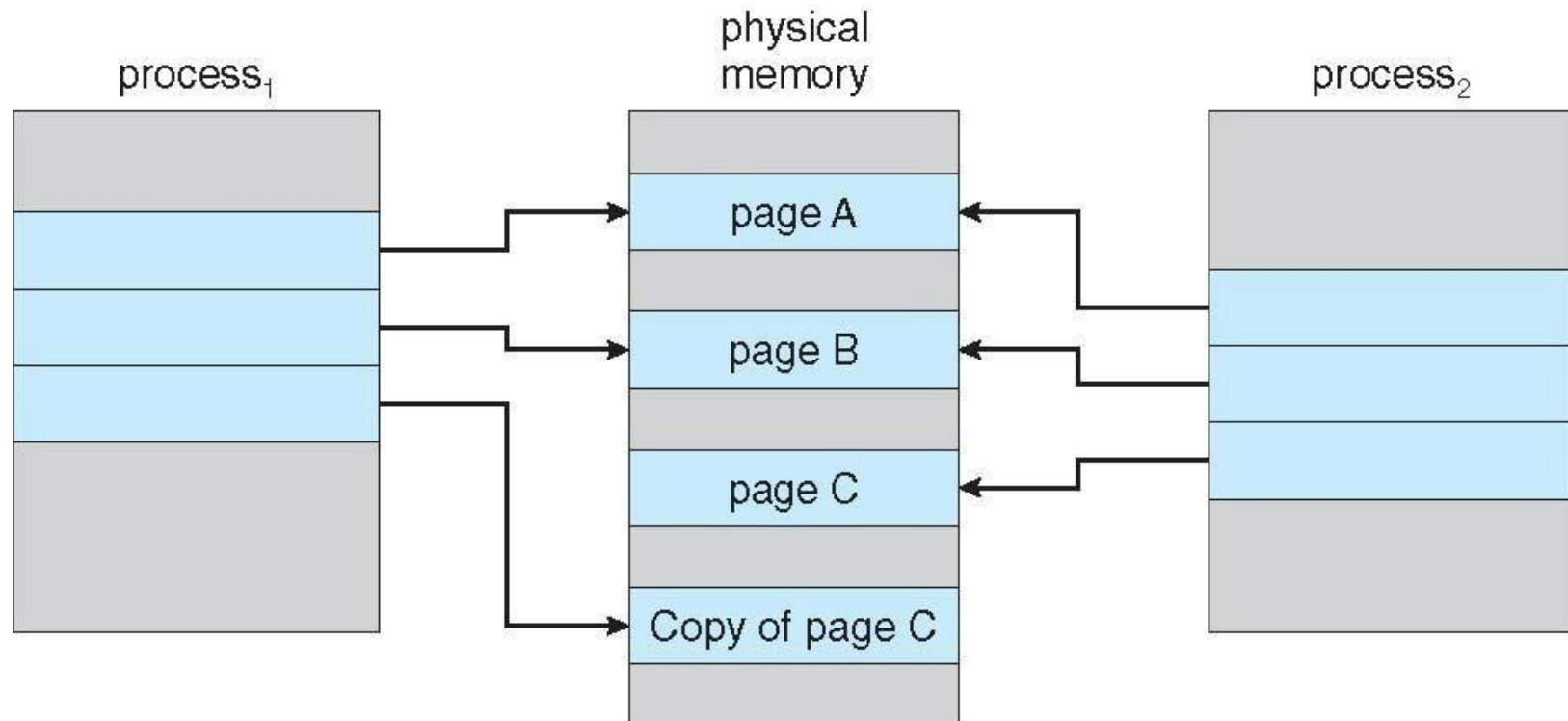


# Before Process 1 Modifies Page C





# After Process 1 Modifies Page C







# What Happens if There is no Free Frame?



- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times