Operating Systems
CS 415

Lecture 12:
Virtual Memory

Suyash Gupta
Assistant Professor
Distopia Labs and ONRG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 3 is out!

- **Deadline → June 3, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Final → June 10, 2026 at 12:30pm PST, STB 145**
  - Closed book, no cheat sheets, no discussions.

Last Class

- Main Memory (Chapter 9)
- Next, we move to Chapter 10!

How can we execute the programs that are larger than physical memory?

Virtual Memory

- Programs can be larger than physical memory.
- Facilitates execution of processes that are not completely in memory.
- Frees programmers from the concerns of memory-storage limitations.
- Allows processes to share files and libraries, and to implement shared memory.

Memory Resident Programs

- Placing the entire logical address space of a process in physical memory, when possible, is a good idea but often unnecessary.
  - limits the size of a program to the size of physical memory.

- Consider the following:
  - Programs often have code to handle unusual error conditions.
  - Errors seldom, if ever, occur in practice, this code is almost never executed.

  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.

Supporting Multiprogramming

- Executing a program that is only partially in memory would has benefits:
  - A program would no longer be constrained by the amount of physical memory that is available.
  - As each program takes less physical memory, more programs can run at the same time, with a corresponding increase inCPU utilization and throughput.
  - Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

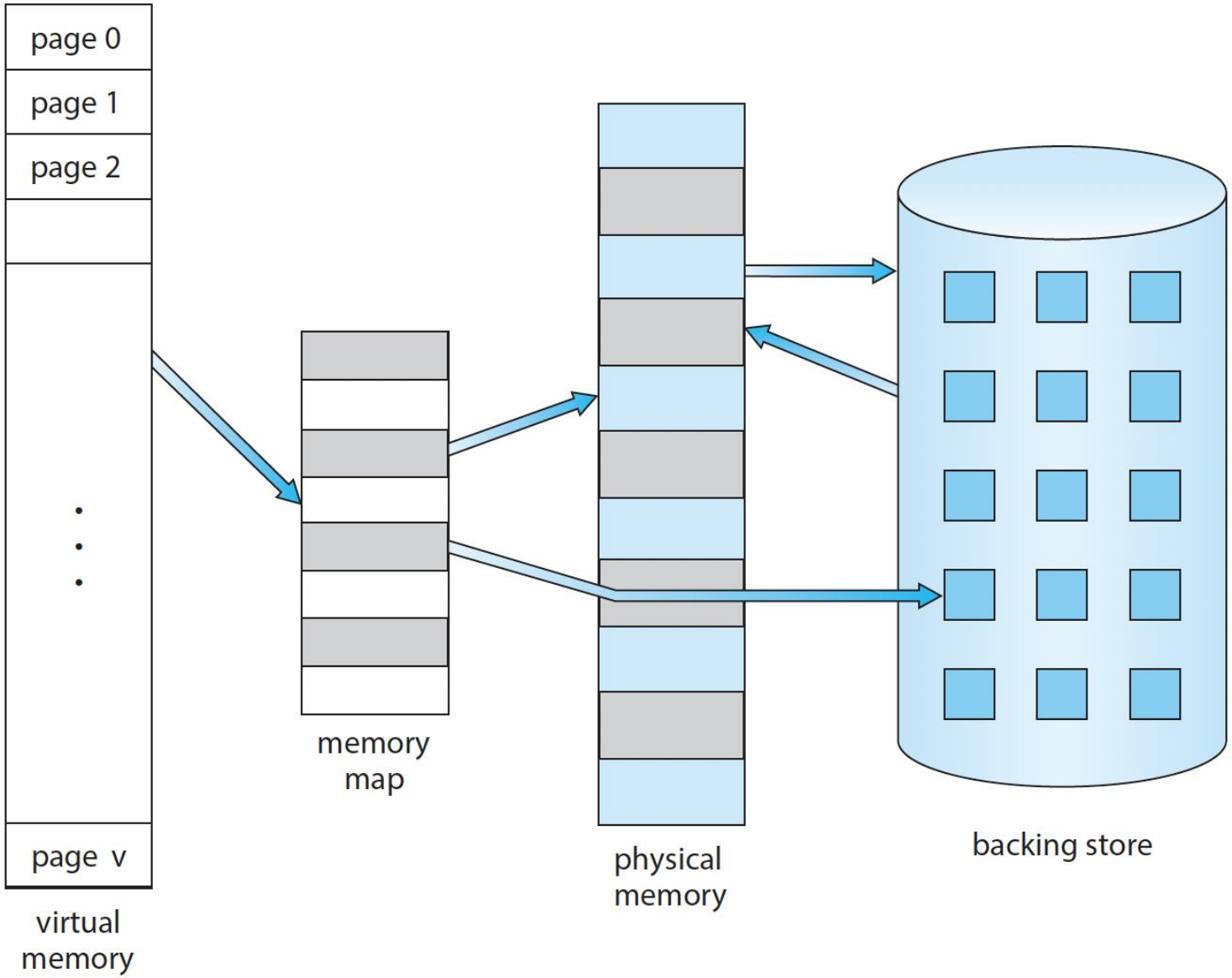
Virtual Memory

- Virtual memory → separates the logical memory as perceived by developers from physical memory.

- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

- Virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
  - Each process begins at a certain logical address - say, address 0 - and exists in contiguous memory.

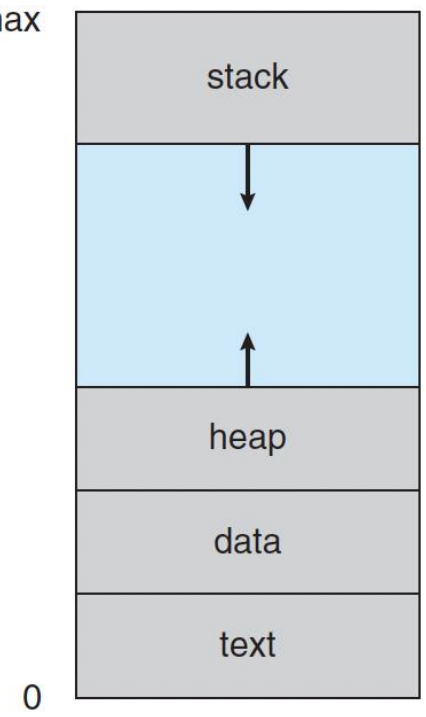
Virtual Memory



Process Memory

- On creating a process, OS sets up a **virtual address space layout**.
  - Decides where the code, data, heap, and stack regions will start.
  - But, it doesn't actually map most of it yet.
  - Even the heap starts at size zero, and the stack gets just a small initial chunk of physical frames.

- The gap between heap and stack is the unallocated virtual address space.
  - **Allocated virtual memory** → the OS acknowledges a region exists (e.g., calling malloc()) but has not given it a physical frame yet.
  - **Unallocated hole** → completely unmapped, no records, no page table entries, totally empty.



How to share system libraries like libc

How to share system libraries like libc

- **Without virtual memory**:
  - Every process needs its own physical copy of libc loaded at a specific fixed location in RAM.
  - Or, load the library once at a fixed physical address and have all processes access it. Every program would need to be compiled knowing exactly where in RAM that library lives.

- **With virtual memory**:
  - OS loads libc once into physical RAM (say frame 200–250), and then maps it into every process's virtual address space at whatever virtual address each process expects.
  - All processes share the same physical frames, but each thinks it has its own private copy at its own virtual address.

Shared Memory Communication

- **Without virtual memory**:
  - Shared memory communication is hard.
  - Since all processes share one flat physical address space, they have to agree on a fixed physical address for the shared region at compile time, ensure no other process uses that range, and manually coordinate access.
  - It's fragile, unsafe, and doesn't scale to multiple processes.

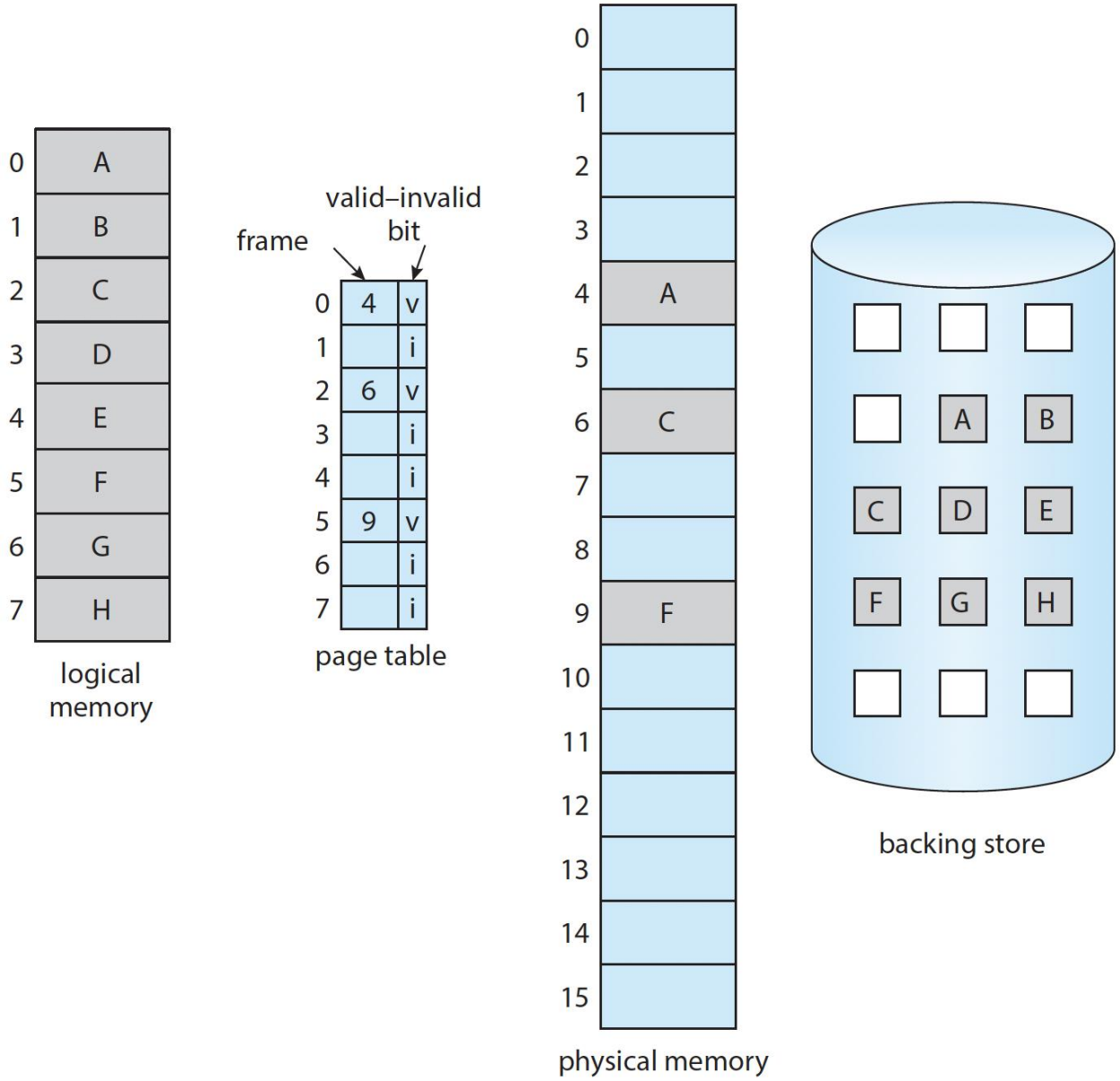
- **With virtual memory**:
  - OS simply maps the same physical frames into two different processes' page tables at potentially different virtual addresses.
  - Ex: Process A writes to virtual 0x5000, Process B reads from virtual 0x8000.

Implementing Virtual Memory

Demand Paging

- Helps to supporting virtual memory.
- Pages are loaded only when they are demanded during program execution.
  - Pages that are never accessed are never loaded into physical memory.
- Hardware support to distinguish between pages in memory and on disk:
  - Valid–invalid bit.
  - If bit is set to “valid,” the associated page is both legal and in memory.
  - If the bit is set to “invalid,” the page either is not valid (not in the logical address space of the process) or is valid but is currently in secondary storage.
  - If the process tries to access a page that is not in memory → Page fault

Demand Paging



Pure Demand Paging

- Never bring a page into memory until it is required.
- Start executing a process with no pages in memory.
- When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.

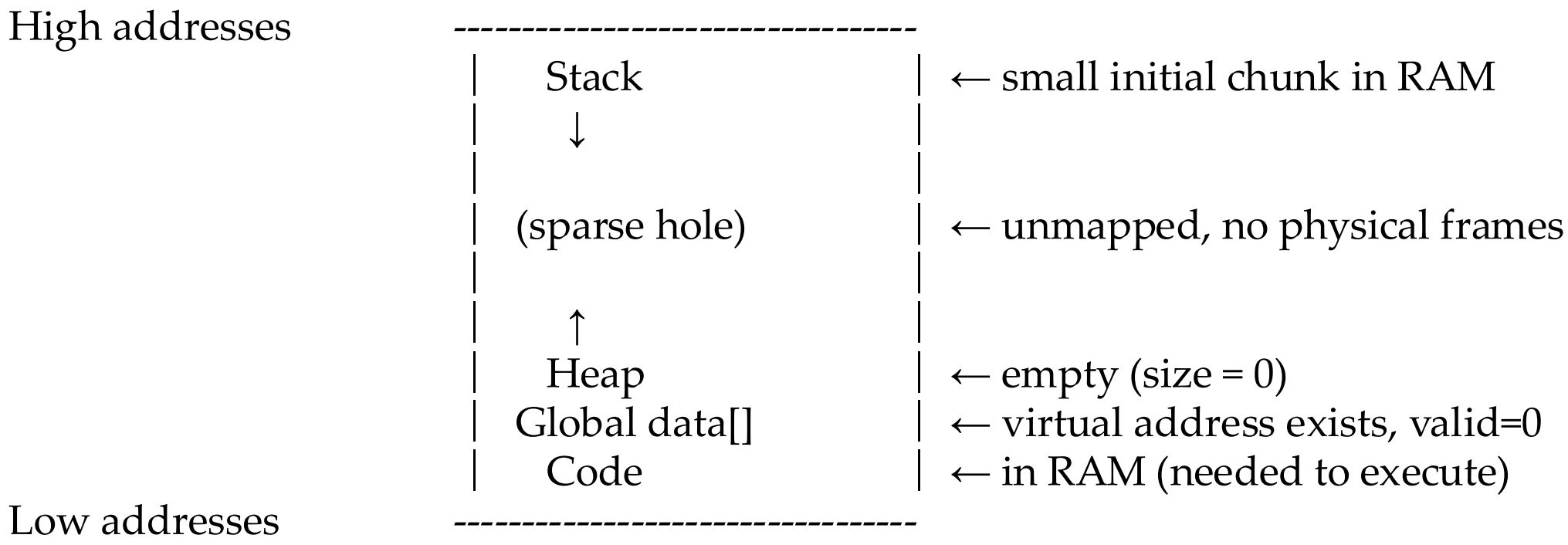
Page Fault

- **Process A** (ASID = 3) has just been created and started running.

```
#include <stdio.h>
int data[1000];
int main() {
        data[500] = 42;         // Line 1
        printf("done\n");       // Line 2
        return 0;
}
```

- OS has set up Process A's virtual address space into physical RAM: the code segment and a small initial stack.
- The heap is empty, and data[] (a global array) has been noted in the virtual layout but no physical frame has been assigned yet.

Page Fault


High addresses
Stack
↓
(sparse hole)
↑
Heap
Global data[]
Code
Low addresses
← small initial chunk in RAM
← unmapped, no physical frames
← empty (size = 0)
← virtual address exists, valid=0
← in RAM (needed to execute)

Page Fault

- **Step 1: CPU Executes Line 1: data[500] = 42.**
  - The compiler has translated **data[500]** into a virtual address, say **0x00402010**.
  - The CPU hands this to the MMU for translation .
  - Virtual address breakdown (4KB pages, 2-level page table):
    - p1 (outer index) = 1
    - p2 (inner index) = 2
    - offset = 16

Page Fault

- **Step 2: TLB lookup (Translation Lookaside Buffer):**
  - Look for entry where VPN matches and ASID = 3
  - TLB Miss!
    - Process A just started; nothing is cached yet
    - The MMU must now walk the 2-level page table .

Page Fault

- **Step 3: Outer Page Table Walk:**
  - MMU reads CR3.
  - Goes to the outer page table (physical address 5000) → uses p1 = 1.

Outer Page Table (physical addr 5000):
Index 1: [Frame 17 | Valid=1]

  - Valid = 1 → inner page table is in physical frame 17.
  - Physical address = 17 × 4096 = 69632.

Page Fault

- **Step 4: Inner Page Table Walk:**
  - MMU goes to physical address 69632 → uses p2 = 2.

Inner Page Table (physical addr 69632):
Index 2: [Frame ?? | Valid=0] X **PAGE FAULT!**

  - Valid = 0 → this page has never been loaded into RAM.
  - The hardware raises a page fault trap.
  - Process A is suspended mid-instruction.

Page Fault

- **Step 5: OS Page Fault Handler:**
  - The OS inspects its internal memory map for Process A.
  - "Is virtual address 0x00402010 a legal address for Process A?"
  - YES!
    - It falls in the global data segment (data[] array).
    - This is a **demand paging fault** → the page exists on disk (in the program's binary/swap) but hasn't been brought into RAM yet .
  - No!
    - If this were an address in the sparse hole (wild pointer), the OS would send SIGSEGV and kill the process .

Page Fault

- **Step 6: Find a Free Physical Frame:**
  - The OS scans its frame allocation table and finds physical frame 33 is free.
  - If no frame were free, the OS would run a page replacement algorithm (e.g., LRU).
  - Pick the least recently used page from any process, write it to disk if its dirty bit = 1, then reclaim that frame .

Page Fault

- **Step 7: Load Page from Disk into Frame 33:**
  - The OS reads the page containing **data[500]** from the executable file on disk into physical frame **33**.
  - Process A is placed in waiting state during this I/O (takes milliseconds).
  - The CPU context-switches to run another process in the meantime.

Page Fault

- **Step 8: Update the Page Table:**
  - Once loaded, the OS updates the inner page table entry :

Inner Page Table (physical addr 69632):
Index 2: [Frame 33 | Valid=1] → was 0, now 1.

Page Fault

- **Step 9: Update the TLB:**
  - The new translation is inserted into the TLB, tagged with ASID = 3:

TLB:
[VPN=(p1=1,p2=2) | Frame=33 | ASID=3 | Valid=1 | Dirty=0]

  - Future accesses to this same page by Process A will get a TLB hit.
    - No page table walk needed!

Page Fault

- **Step 10: Update the TLB:**
  - The new translation is inserted into the TLB, tagged with ASID = 3:

TLB:
[VPN=(p1=1,p2=2) | Frame=33 | ASID=3 | Valid=1 | Dirty=0]

  - Future accesses to this same page by Process A will get a TLB hit.
    - No page table walk needed!

Cost of Demand Paging

- **Page Fault Rate** (p), $0 \le p \le 1$
  - if p = 0, no page faults
  - if p = 1, every reference is a fault

- **Effective Access Time (EAT)**

  EAT = (1 – p) * memory access
  + p * (page fault overhead + swap out + swap in + memory access)

Cost of Demand Paging

- Example:
  - Assume memory access time = 200 nanoseconds.
  - Average page-fault service time = 8 milliseconds.

- EAT = (1 – p) * 200 + p (8 milliseconds)
  = 200 – p * 200 + p * 8,000,000 = 200 + p x 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 x p
  20 > 7,999,800 x p
  p < .0000025; one page fault in every 400,000 memory accesses.

What happens if there is no free frame?

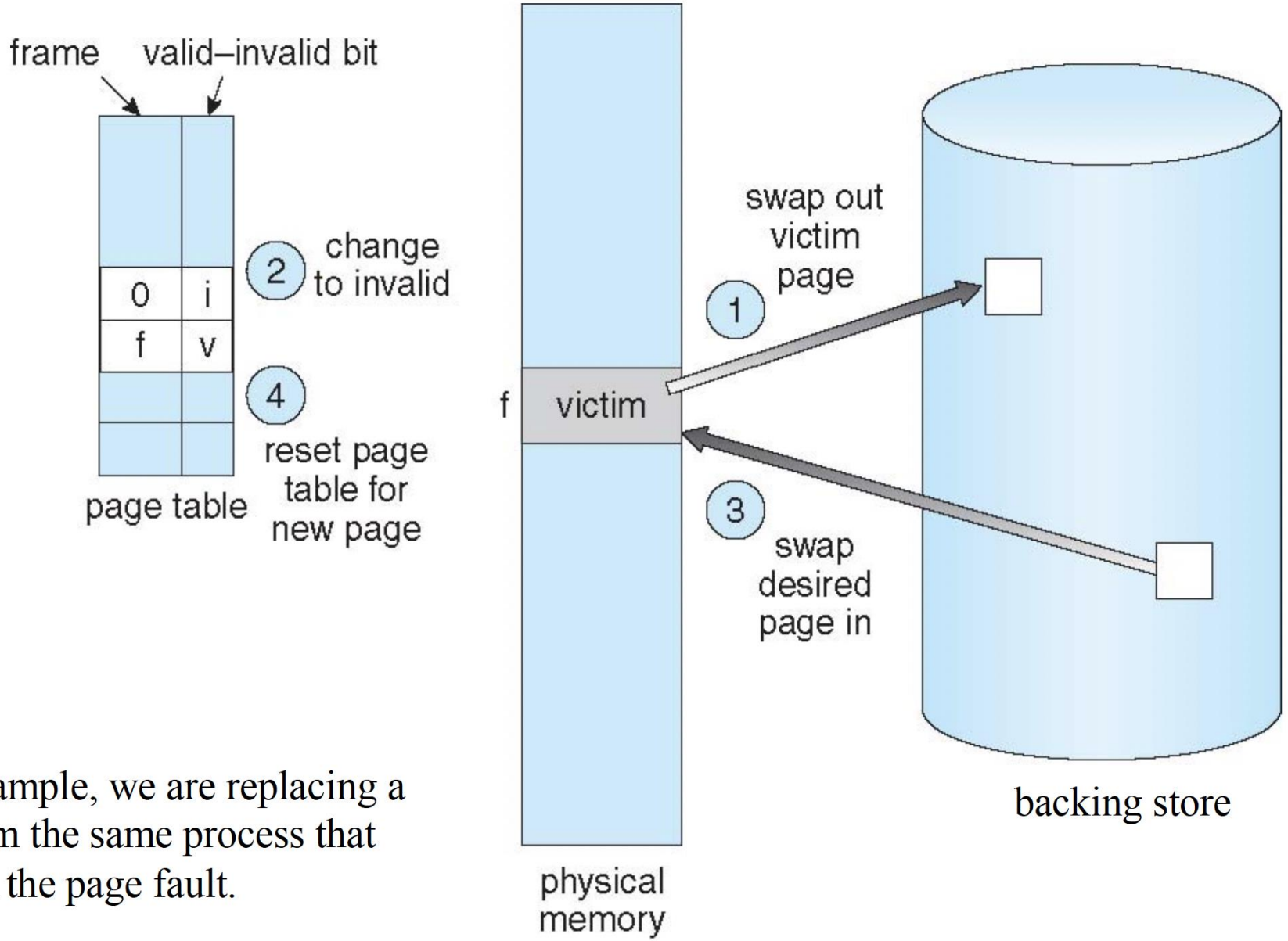
- **Page replacement**
  - Find some page in memory (what page?)
  - Algorithm: terminate? swap out? replace the page?
  - Performance: want an algorithm which will result in minimum number of page faults.

- Same page may be brought into memory several times during the lifetime of a process.

Basic Page Replacement

- Find the location of the desired page on disk.
- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame.
  - Write victim frame to disk if dirty.
- Bring the desired page into the (newly) free frame.
  - Need to update the page and frame tables.
- Continue the process by restarting the instruction that caused the trap (i.e., page fault).

Basic Page Replacement



In this example, we are replacing a pages from the same process that generated the page fault.

How to evaluate a Page Replacement algorithm?

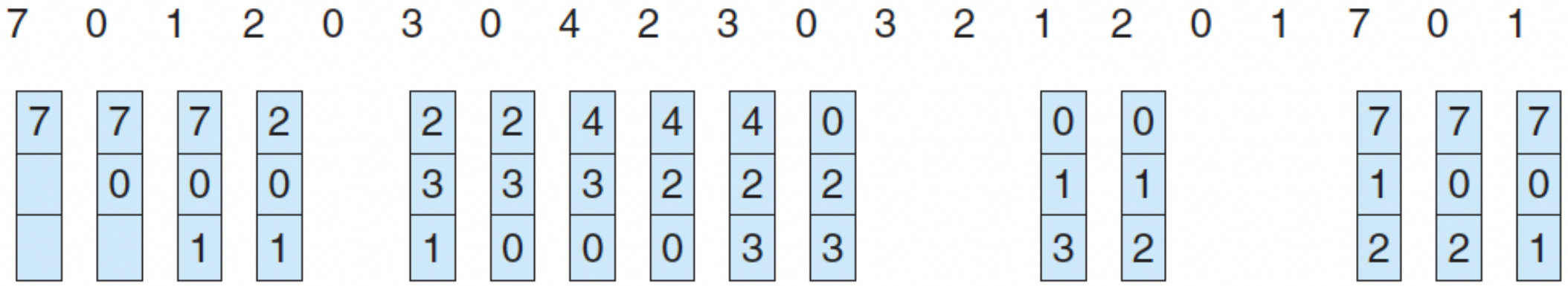
- Develop a frame-allocation algorithm and a page-replacement algorithm.
- If we have multiple processes in memory, we must decide how many frames to allocate to each process.
- Evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults → **Reference string**.
- Reference String: **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**

How to evaluate a Page Replacement algorithm?

- Develop a frame-allocation algorithm and a page-replacement algorithm.
- If we have multiple processes in memory, we must decide how many frames to allocate to each process.
- Evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults → **Reference string**.
- Reference String: **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**

FIFO Page Replacement

- Simplest page-replacement algorithm.
- Associate with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- Assume we have a total of **3 frames** initially.

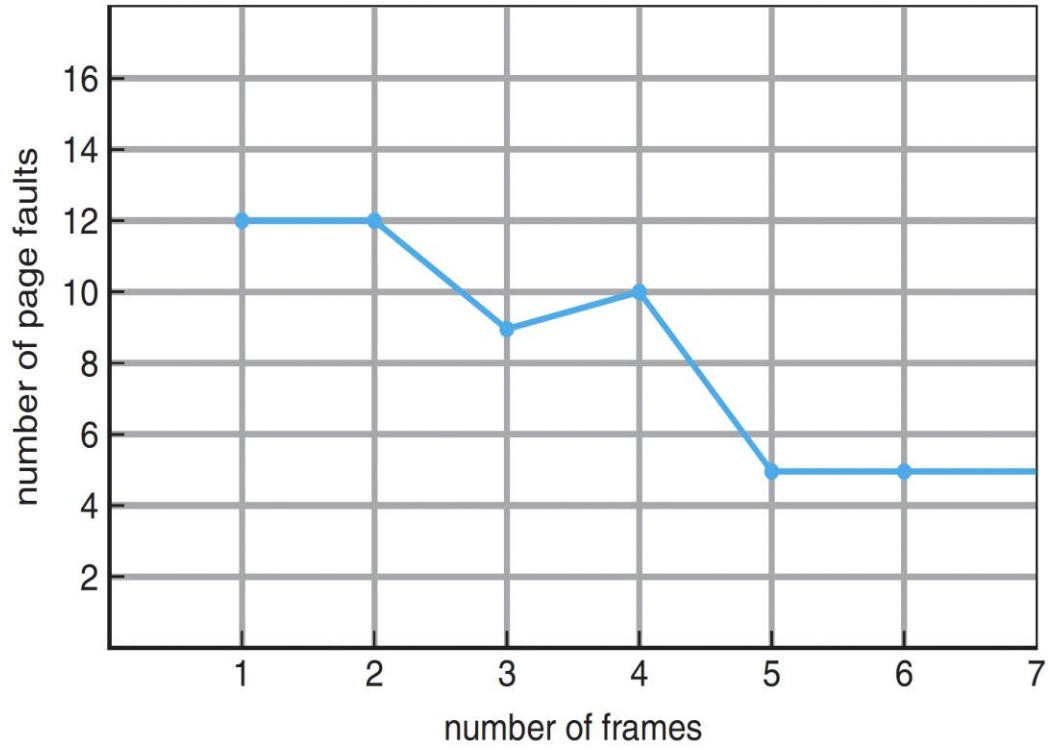


FIFO Disadvantages

FIFO Disadvantages

- An active page can get periodically replaced and need to be brought back!
  - A bad replacement choice increases the page-fault rate and slows process execution.

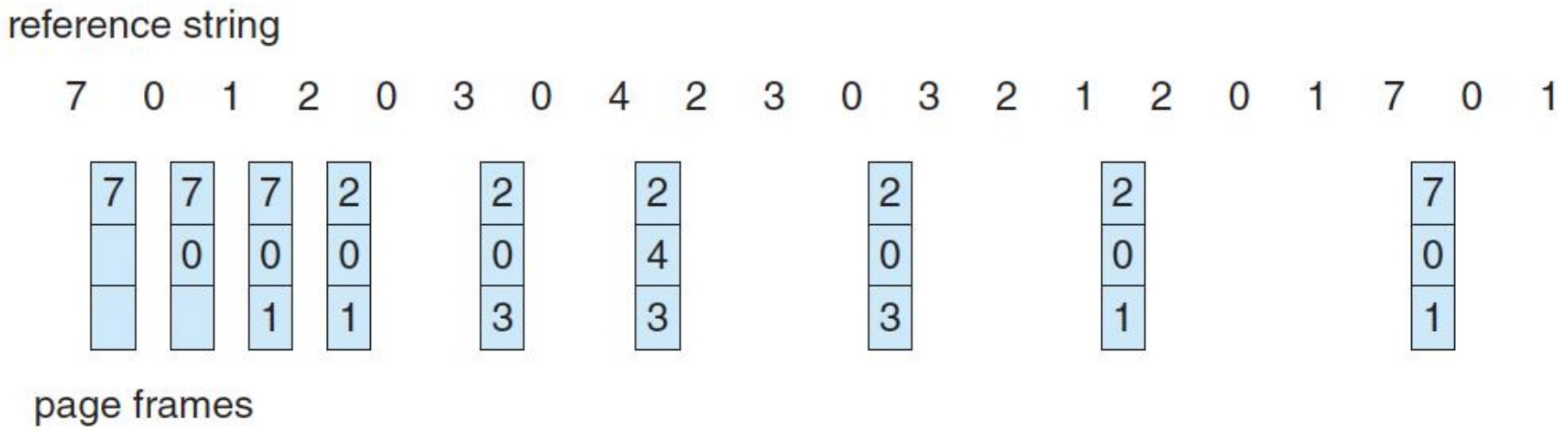
- Suffers **Belady's anomaly**.
  - Adding more frames can cause more page faults!
  - Consider this reference string: 1,2,3,4,1,2,5,1,2,3,4,5



Optimal Algorithm

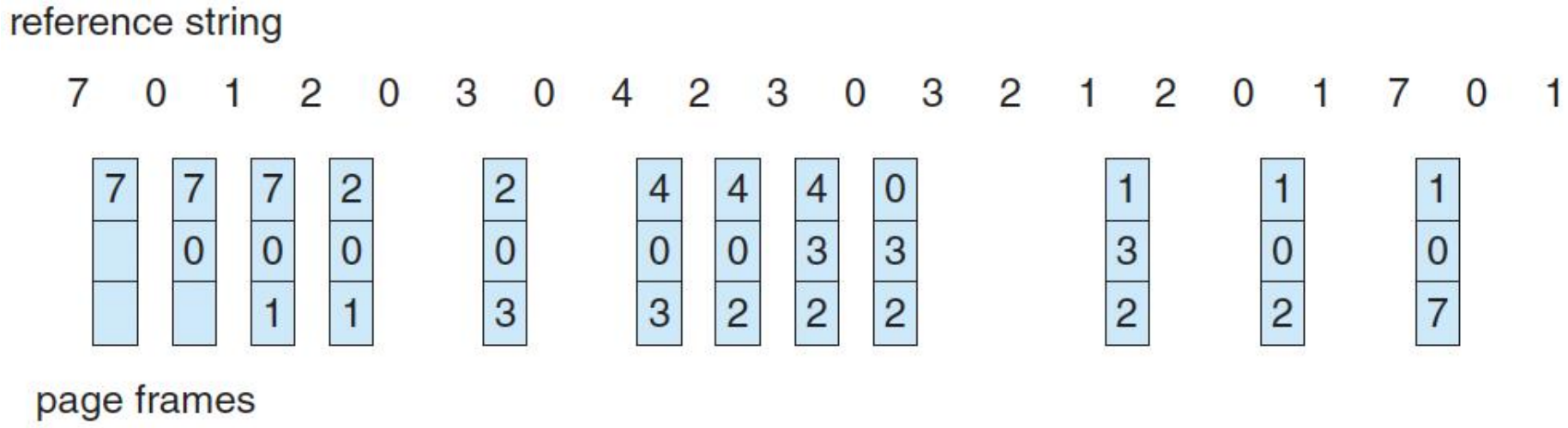
- Replace page that will not be used for longest period of time in the **future**.

- Knowing future is hard!
  - Used as a "best case" for measuring how well your page replacement algorithm performs.



Least Recently Used (LRU)

- Use past knowledge rather than future.
- Replace page that has not been used in the most amount of time (i.e., the one least recently used).
  - Associate time of last use with each page.



- 12 faults → better than FIFO but worse than OPT.
- Generally good algorithm and frequently used in practice.

Implementing LRU

- Implementing using Counters:
  - Every page entry has a counter.
  - Update counter with current clock when the page is referenced.
  - When a page needs to be replaced, look at the counters to find smallest value (among the pages with frames).
    - search through table needed!
- LRU and OPT are examples of algorithms that do not suffer Belady's Anomaly.

Implementing LRU

- LRU needs special hardware.

- Reference bit (1-bit counter)
  - With each page associate a bit, initially = 0.
  - When page is referenced bit set to 1.
  - Replace any page with reference bit = 0 (if one exists).

What if a process does not have enough frames?

Thrashing

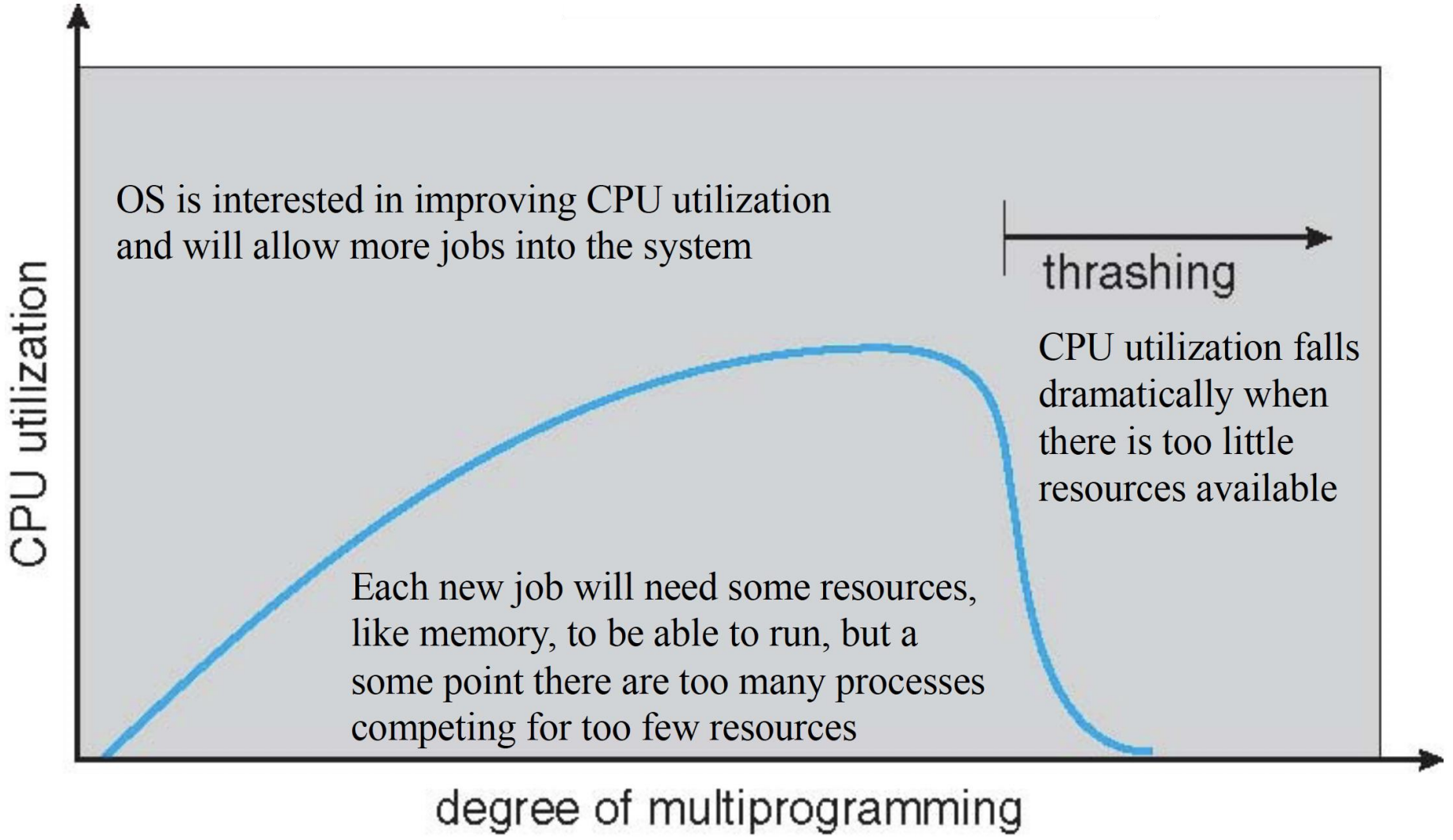
- A process is thrashing if it is spending more time paging than executing.
- Say a process does not have "enough" frames—that is, it does not have the minimum number of frames it needs to support pages in the working set.
  - The process will quickly page-fault.
  - It must replace some page.
  - However, since all its pages are in active use, it must replace a page that will be needed again right away.
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

Thrashing Effects

- Low CPU utilization.

- The operating system thinks that it needs to increase the degree of multiprogramming.
  - Adds another process to the system.
  - More thrashing!
  - Induces thrashing for other processes!

Thrashing Effects



Reducing Thrashing

- Use a local replacement algorithm
  - Local replacement requires that each process select from only its own set of allocated frames.

- If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

- To prevent thrashing, we must provide a process with as many frames as it needs.