Memory Management

Disclaimer: some slides are adopted from book authors' slides with permission

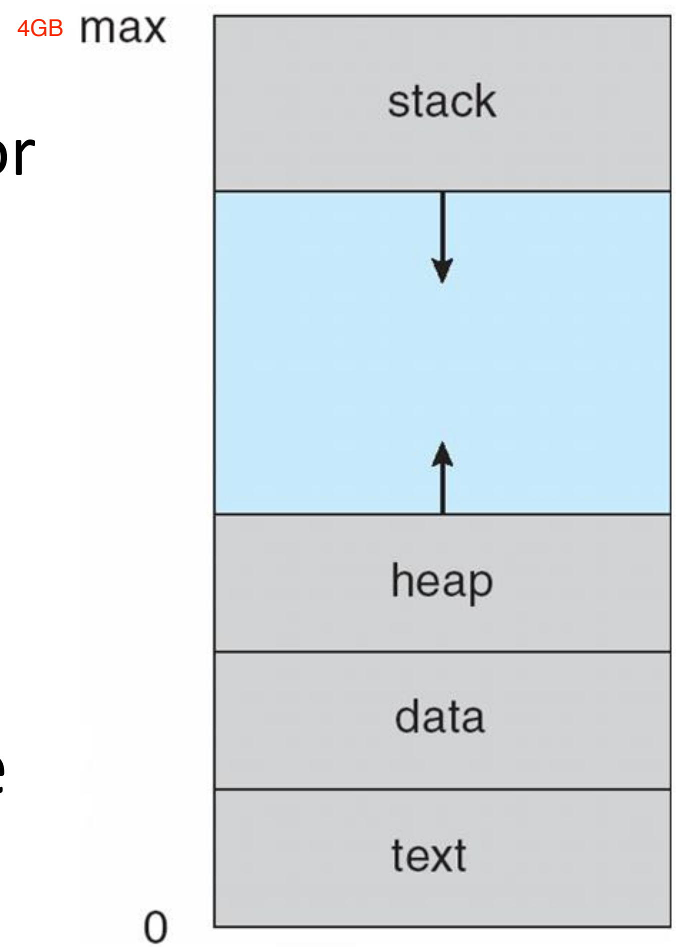
Concepts to Learn

- Demand paging

Virtual Memory (VM)

- Abstraction
  - 4GB linear address space for each process
- Reality
  - 1GB of actual physical memory shared with 20 other processes
- Does each process use the *(1) entire virtual memory* *(2) all the time?*

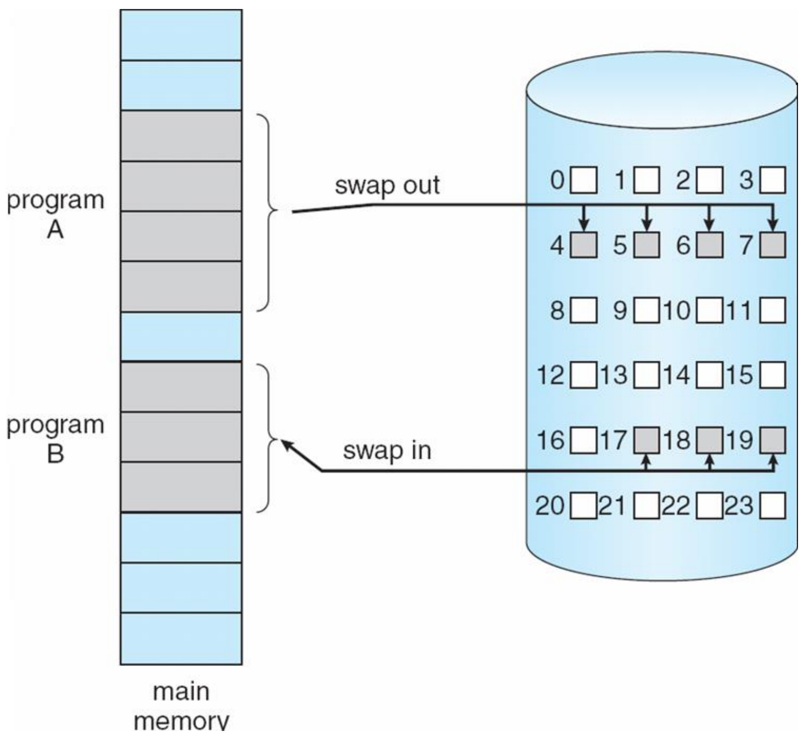
no



Demand Paging

- Idea: instead of keeping the entire memory pages in memory all the time, keep only part of them on a on-demand basis

部分使用内存,不常使用的swap out



Page Table Entry (PTE)

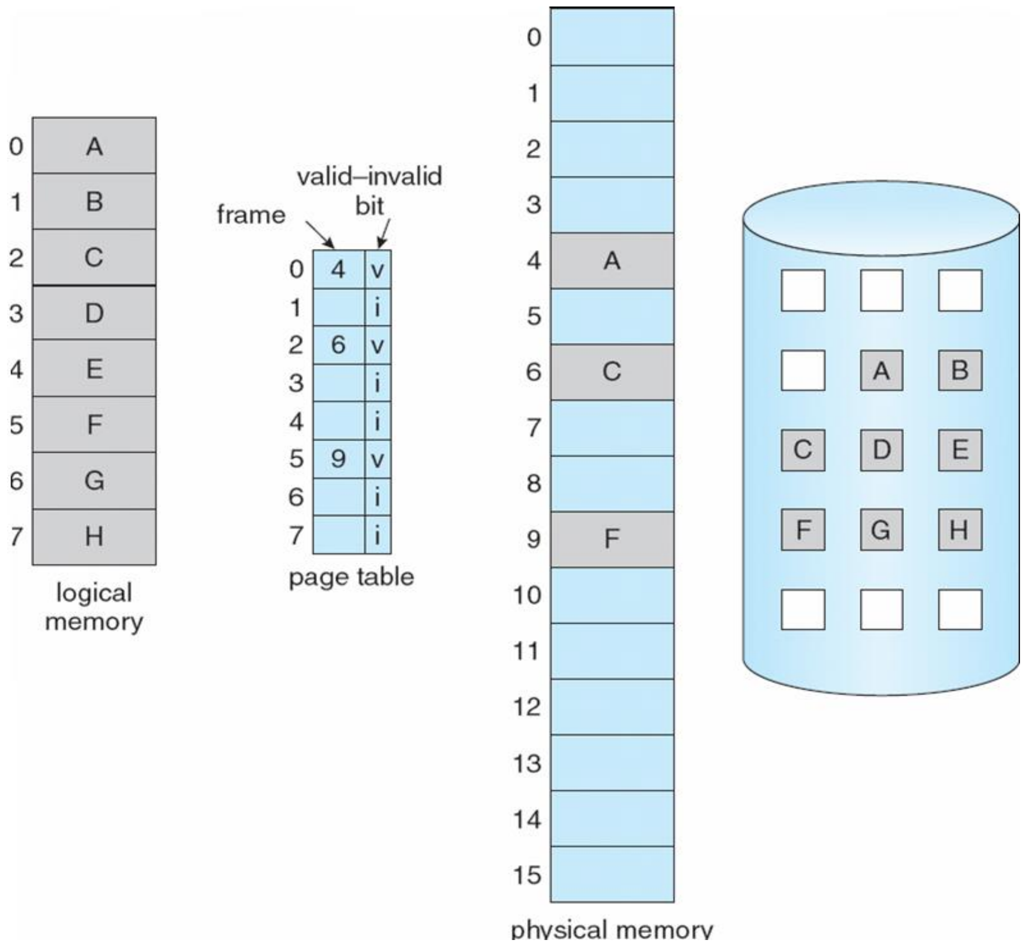
- PTE format (architecture specific)

1	1	1	2	20 bits
V	M	R	P	Page Frame No

  - **Valid bit (V): whether the page is in memory**
  - Modify bit (M): whether the page is modified
  - 被引用过 Reference bit (R): whether the page is accessed
  - Protection bits(P): readable, writable, executable

Partial Memory Mapping

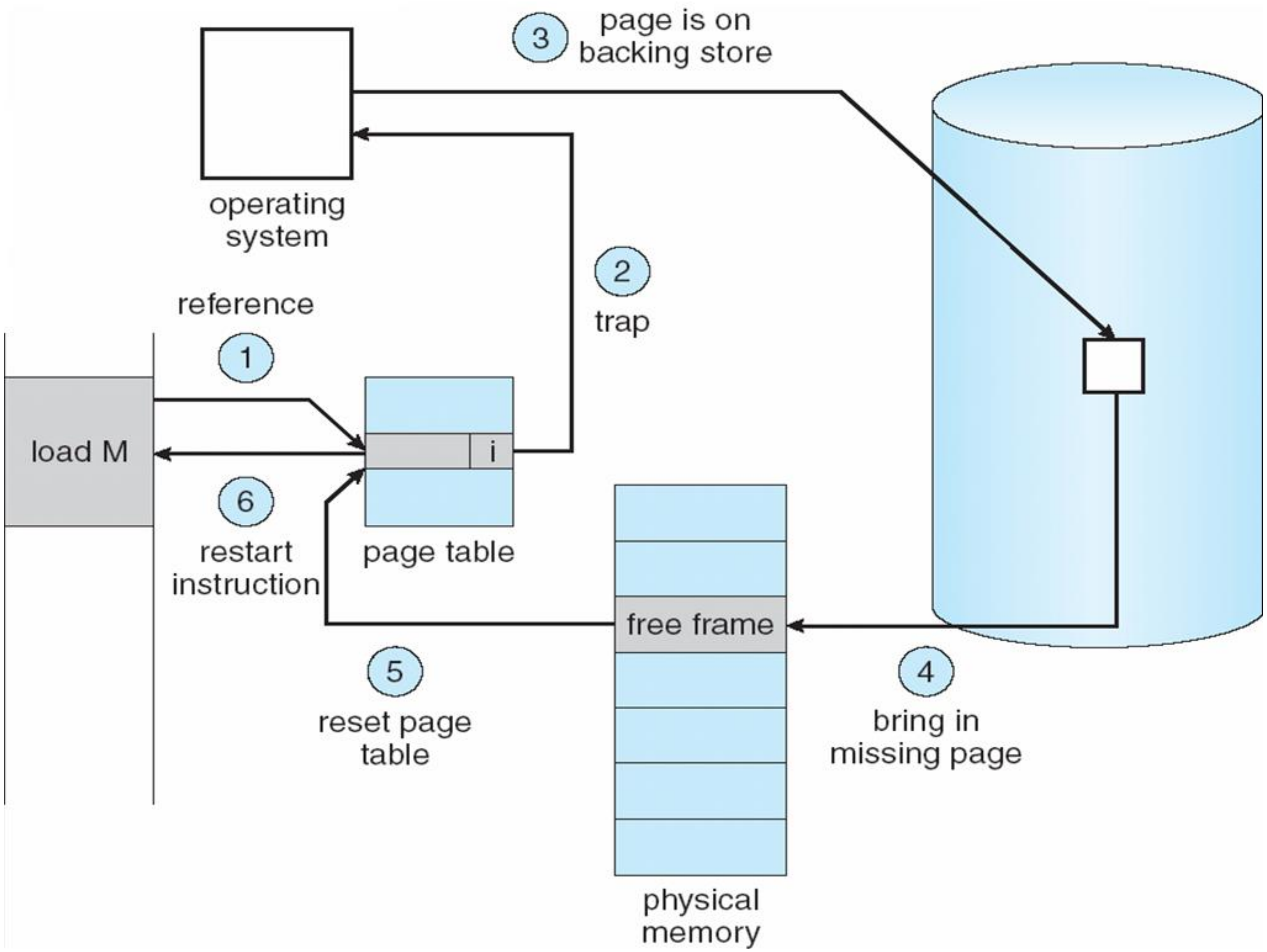
- Not all pages are in memory (i.e., valid=1)



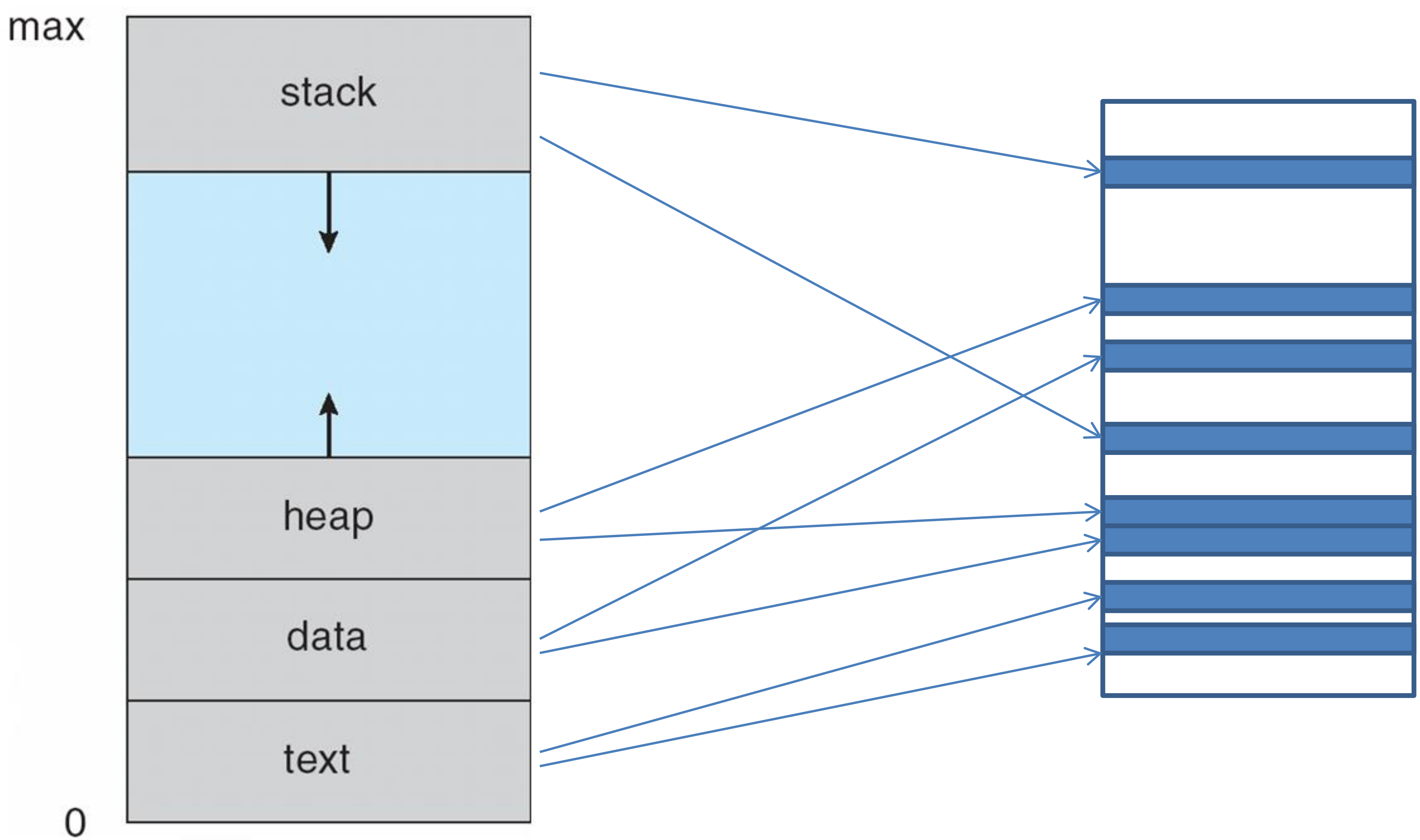
Page Fault

- When a virtual address can not be translated to a physical address, MMU generates a trap to the OS
- Page fault handling procedure
  - Step 1: allocate a free page frame
  - Step 2: bring the stored page on disk (if necessary)
  - Step 3: update the PTE (mapping and valid bit)
  - Step 4: restart the instruction

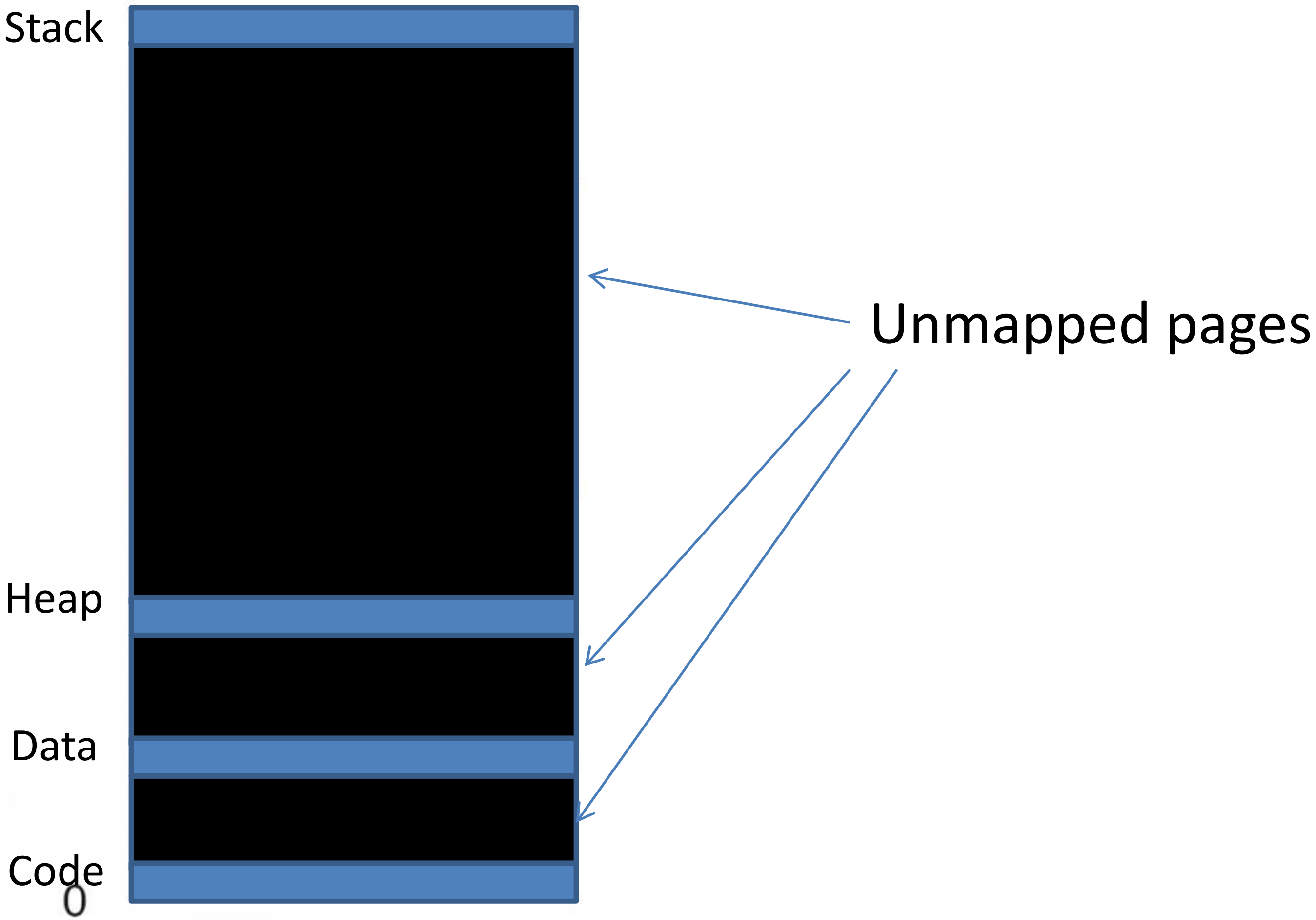
Page Fault Handling



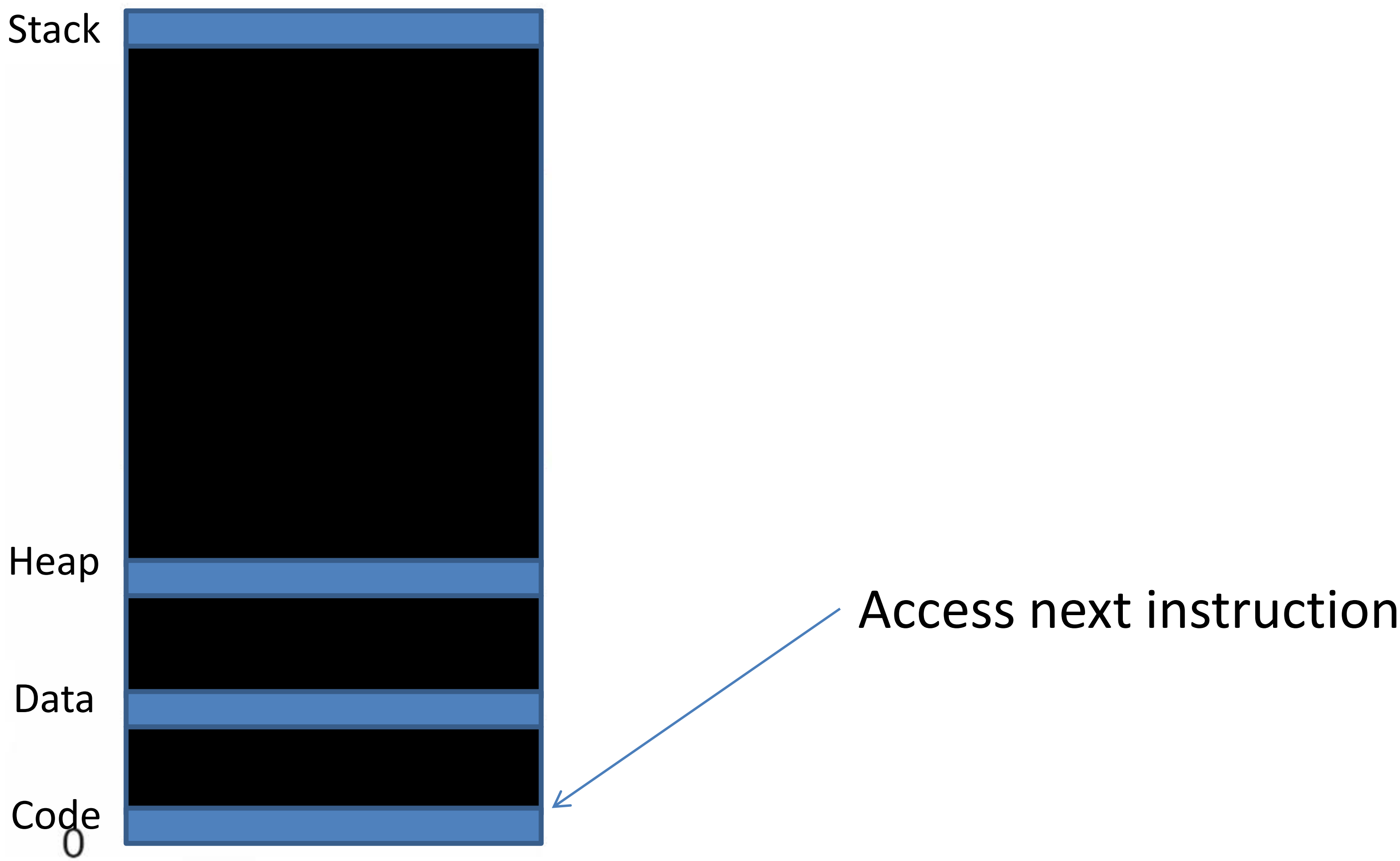
Demand Paging



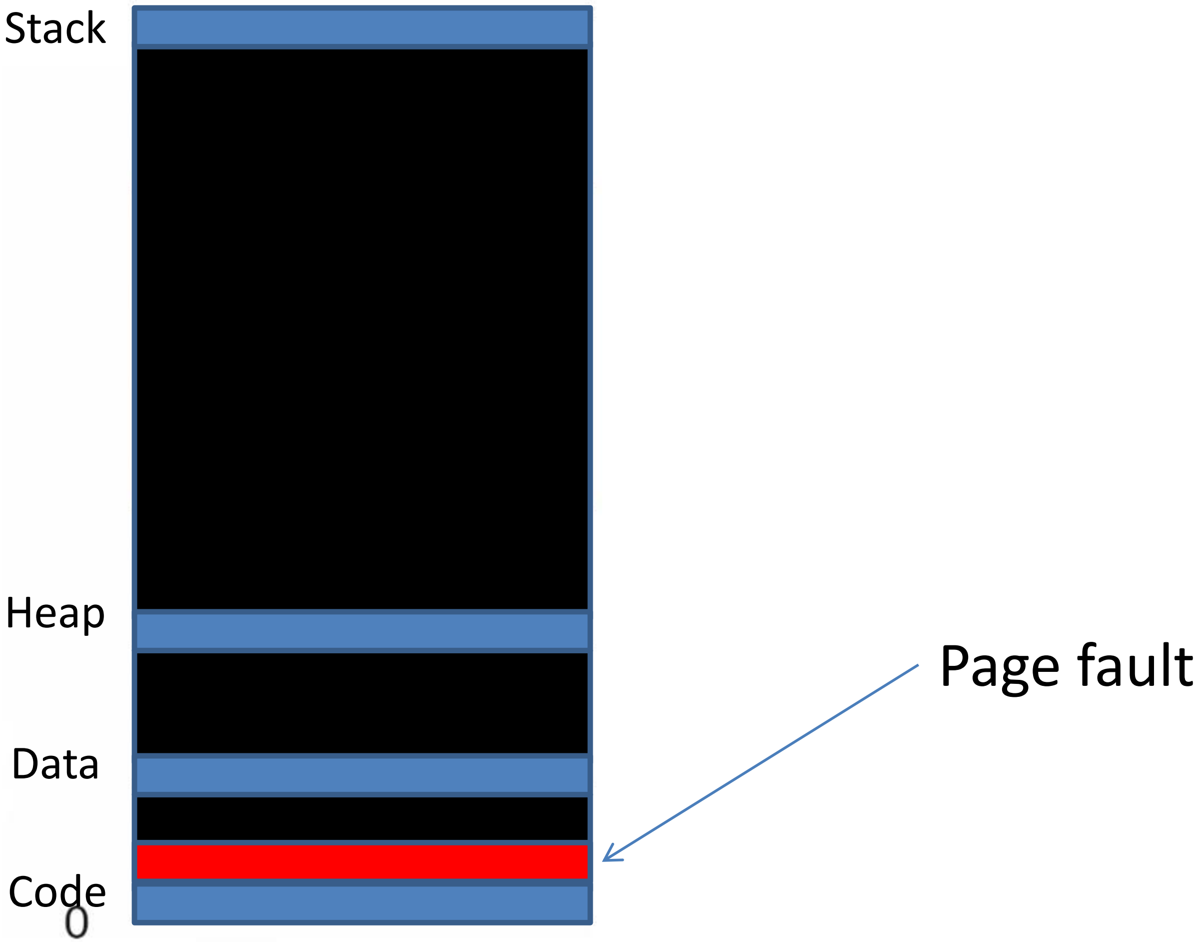
Starting Up a Process



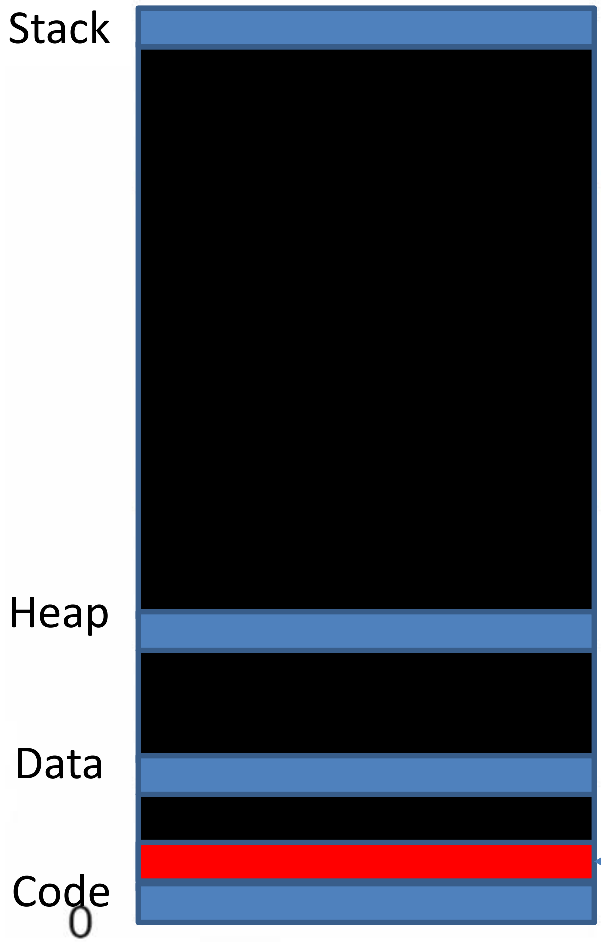
Starting Up a Process



Starting Up a Process



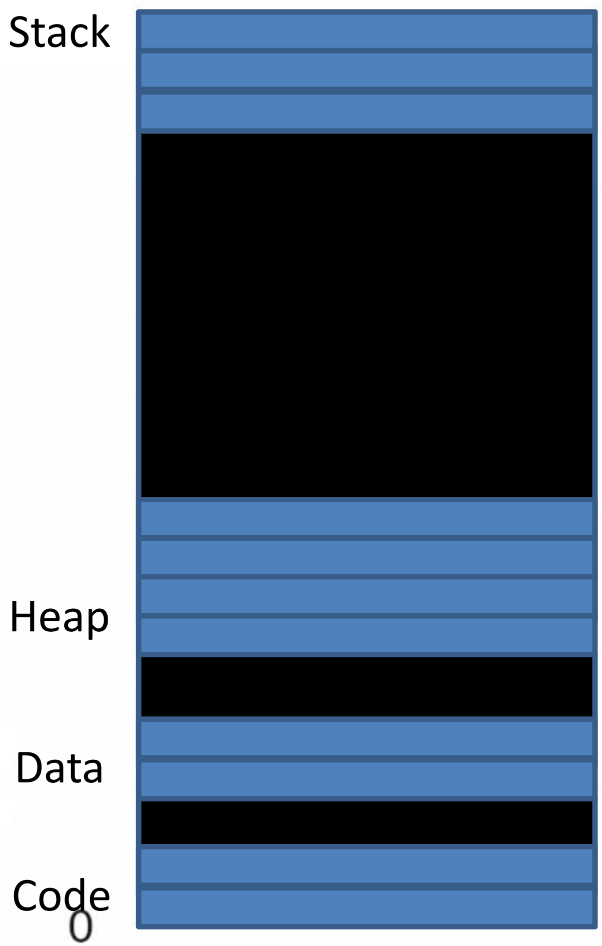
Starting Up a Process



OS

1) allocates free page frame
2) loads the missed page from the disk (exec file)
3) updates the page table entry

Starting Up a Process



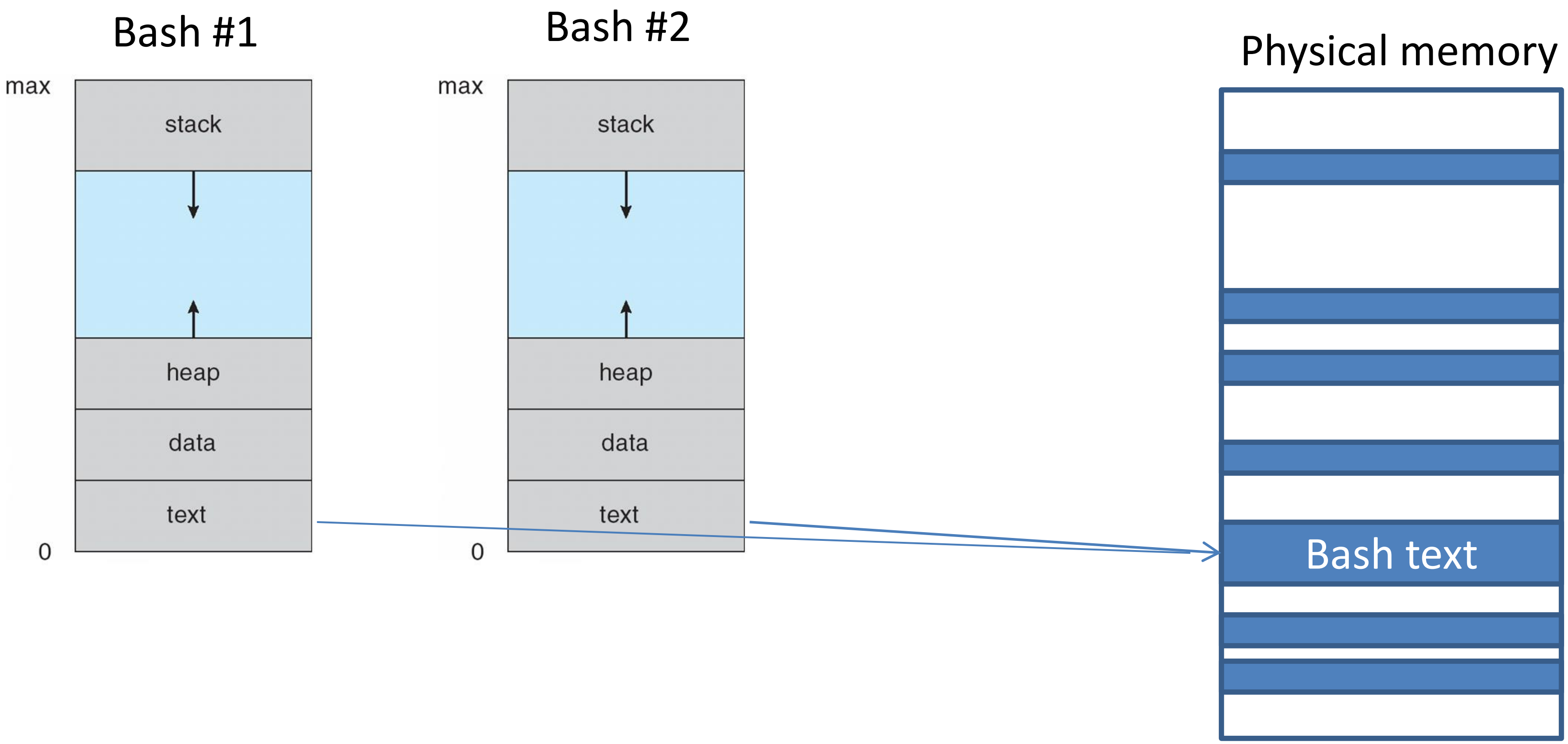
Over time, more pages are mapped as needed

Anonymous Page

- An executable file contains code (binary)
  - So we can read from the executable file

- What about heap?
  - No backing storage (unless it is swapped out later)
  - Simply map a new free page (anonymous page) into the address space

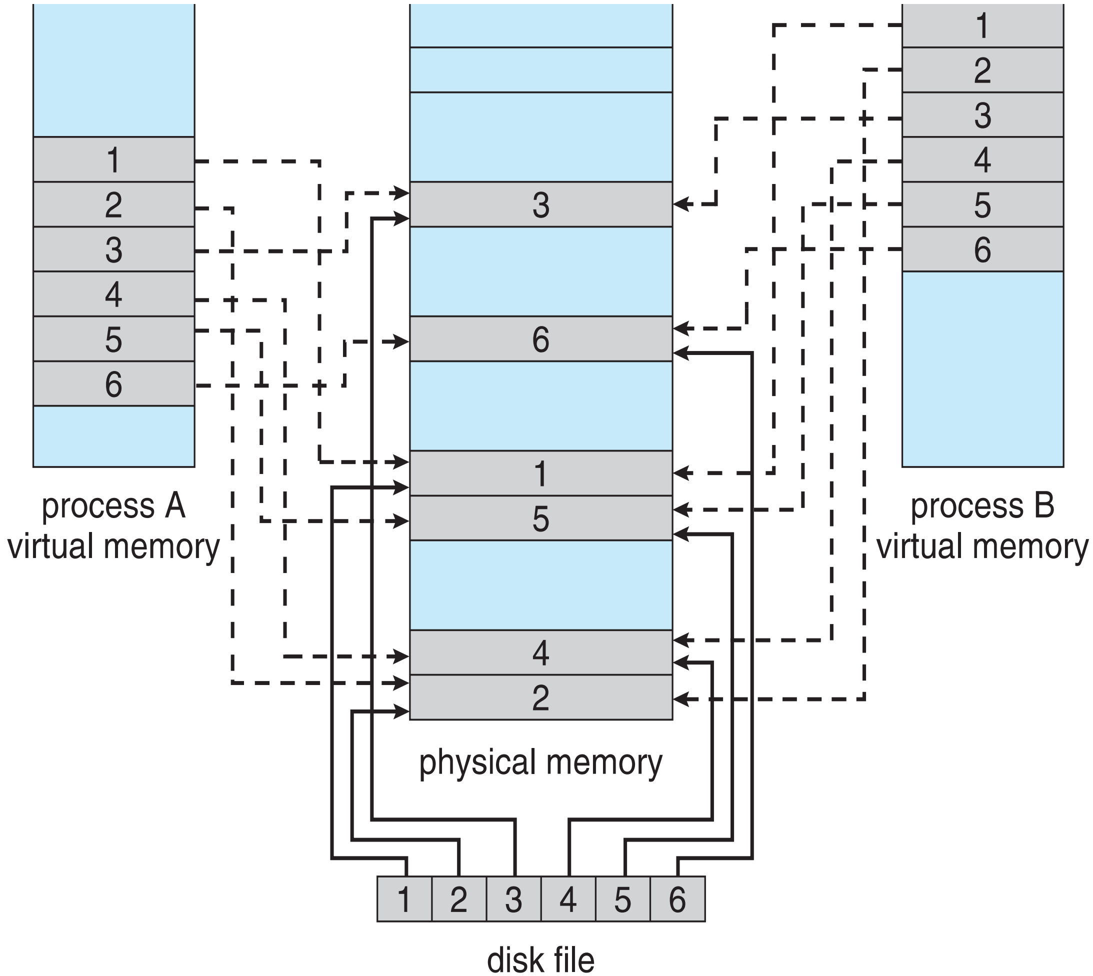
Program Binary Sharing



- Multiple instances of the same program
  - E.g., 10 bash shells

Memory Mapped I/O

- Idea: map a file on disk onto the memory space



Memory Mapped I/O

- Benefits: you don't need to use read()/write() system calls, just directly access data in the file via memory instructions
- How it works?
  - Just like demand paging of an executable file
  - What about writes?
    - Mark the modified (M) bit in the PTE
    - Write back the modified pages back to the original file

Recap

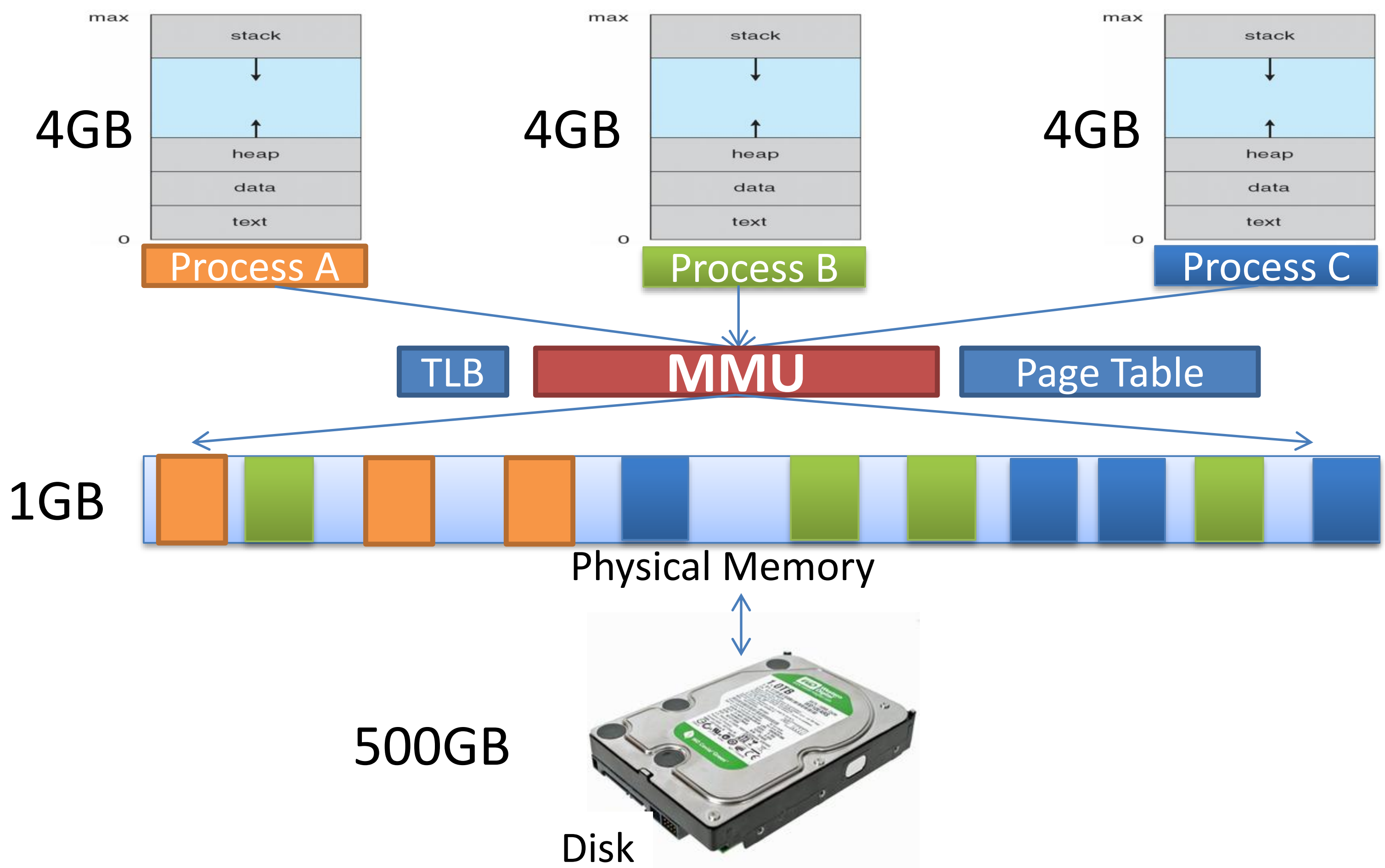
- Multi-level paging
  - Instead of a single big table, many smaller tables
  - Save space
- Demand paging
  - Mapping memory dynamically over time
  - keep necessary pages on-demand basis
- Page fault handling
  - Happens when the CPU tries to access unmapped address.

Concepts to Learn

- Page replacement policy
- Thrashing

Memory Size Limit?

- Demand paging → **illusion of infinite memory**



Illusion of Infinite Memory

- Demanding paging
  - Allows more memory to be allocated than the size of physical memory
  - Uses memory as **cache** of disk

- What to do when memory is full?
  - On a page fault, there's no free page frame
  - Someone (page) must go (be evicted)

Recap: Page Fault

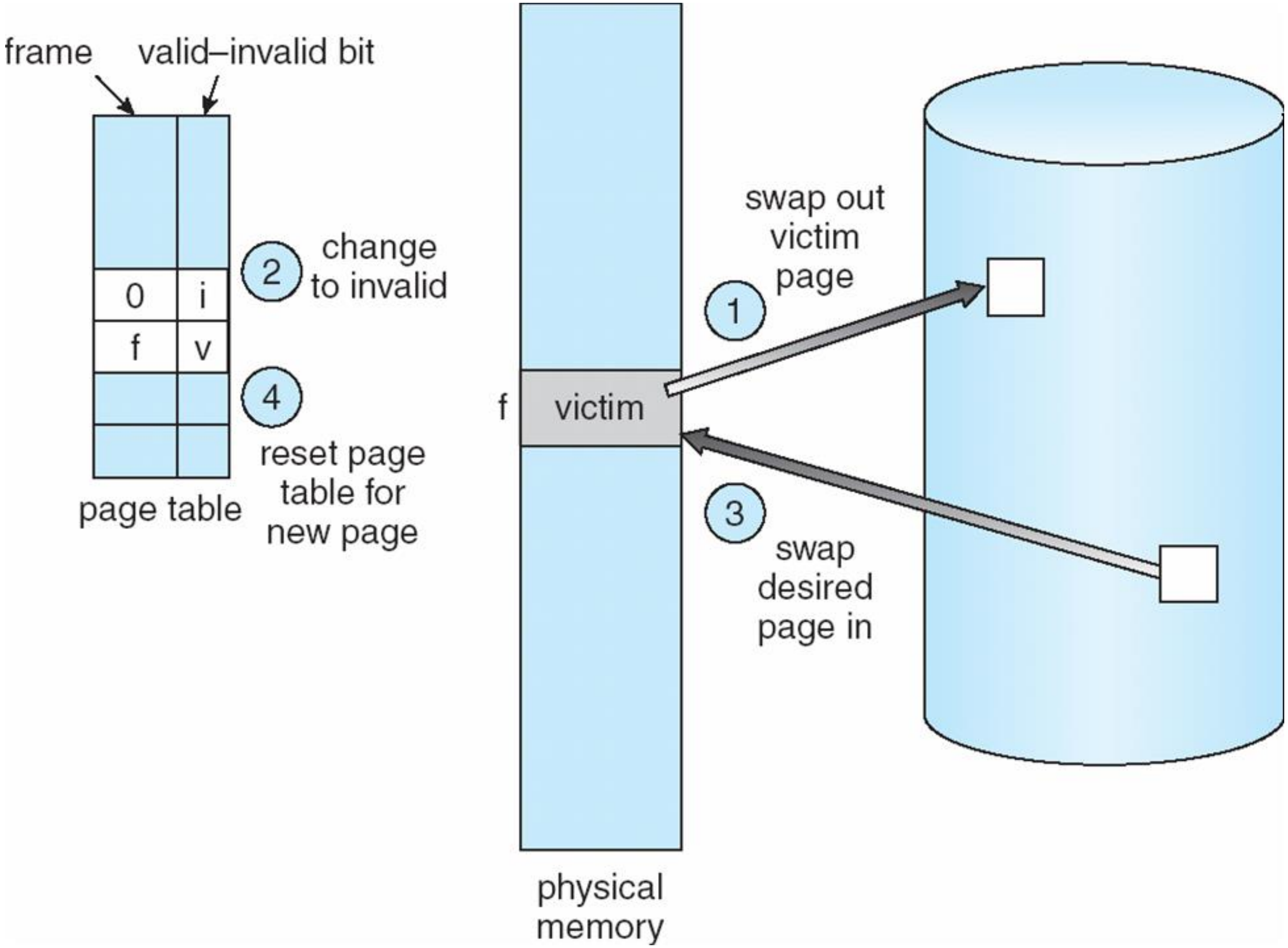
- On a page fault
  - Step 1: allocate a free page frame
  - Step 2: bring the stored page on disk (if necessary)
  - Step 3: update the PTE (mapping and valid bit)
  - Step 4: restart the instruction

KU THE UNIVERSITY OF KANSAS

Page Replacement Procedure

- On a page fault
  - Step 1: allocate a free page frame
    - If there's a free frame, use it
    - If there's no free frame, choose a **victim frame** and evict it to disk (if necessary) → **swap-out**
  - Step 2: bring the stored page on disk (if necessary)
  - Step 3: update the PTE (mapping and valid bit)
  - Step 4: restart the instruction

Page Replacement Procedure



Page Replacement Policy

- Which page (a.k.a. victim page) to go?
  – What if the evicted page is needed soon?
    - A page fault occurs, and the page will be re-loaded
  – Important decision for performance reason
    - The cost of choosing wrong page is very high: disk accesses

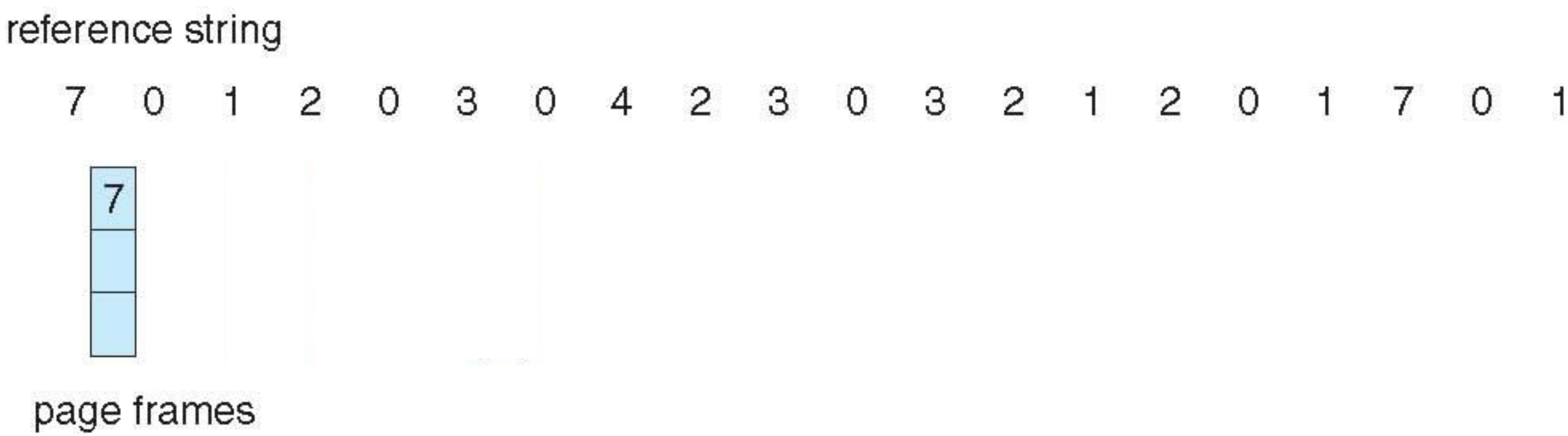
Page Replacement Policies

- FIFO (First In, First Out)
  - Evict the oldest page first.
  - Pros: fair
  - Cons: can throw out frequently used pages
- Optimal
  - Evict the page that will not be used for the longest period
  - Pros: optimal
  - Cons: you need to know the future

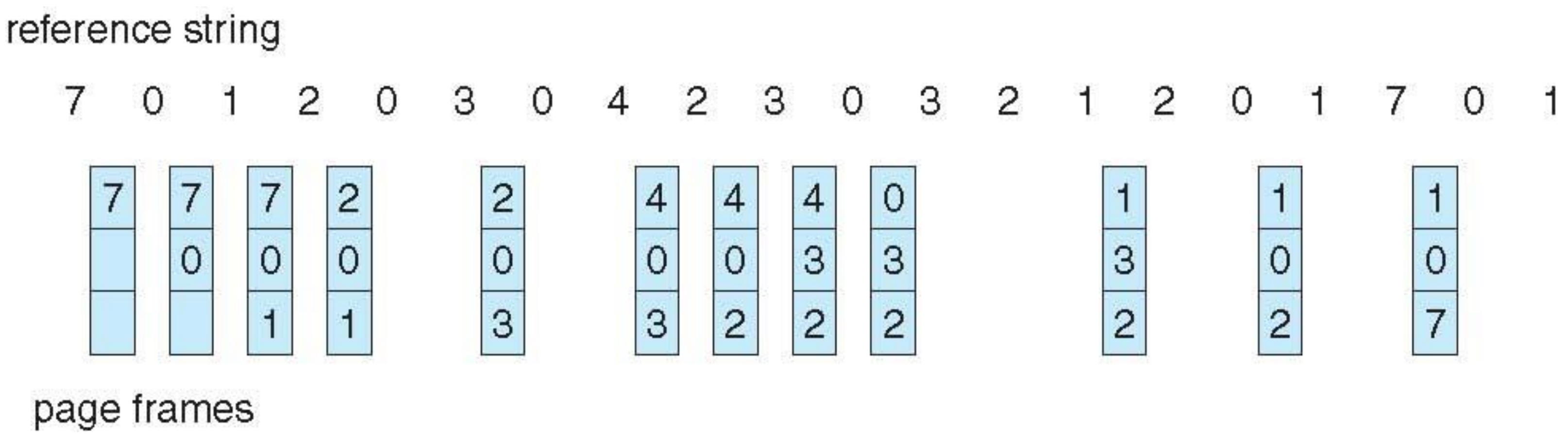
Page Replacement Policies

- Random
  - Randomly choose a page
  - Pros: simple. TLB commonly uses this method
  - Cons: unpredictable
- LRU (Least Recently Used)
  - Look at the past history, choose the one that has not been used for the longest period
  - Pros: good performance
  - Cons: complex, requires h/w support

LRU Example



LRU Example


reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7
7 0
7 0 1
2 0 1
2 0 3
4 0 3
4 0 2
4 3 2
0 3 2
1 3 2
1 0 2
1 0 7
page frames

Example

- Complete the following with the FIFO, Optimal, LRU replacement policies, respectively

Reference	E	D	H	B	D	E	D	A	E	B	E
Page #1	E	E	E								
Page #2		D	D								
Page #3			H								
Mark X for a fault	X	X	X								

FIFO

Reference	E	D	H	B	D	E	D	A	E	B	E
Page #1	E	E	E	B	B	B	B	A	A	A	A
Page #2		D	D	D	*	E	E	E	*	B	B
Page #3			H	H	H	H	D	D	D	D	E
Mark X for a fault	X	X	X	X		X	X	X		X	X

Optimal

Reference	E	D	H	B	D	E	D	A	E	B	E
Page #1	E	E	E	E	E	E	E	E	E	E	E
Page #2		D	D	D	D	D	D	A	A	A	A
Page #3			H	B	B	B	B	B	B	B	B
Mark X for a fault	X	X	X	X				X			

LRU

Reference	E	D	H	B	D	E	D	A	E	B	E
Page #1	E	E	E	B	B	B	B	A	A	A	A
Page #2		D	D	D	D	D	D	D	D	B	B
Page #3			H	H	H	E	E	E	E	E	E
Mark X for a fault	X	X	X	X		X		X		X	

KU THE UNIVERSITY OF KANSAS

Implementing LRU

- Ideal solutions
  - Timestamp
    - Record access time of each page, and pick the page with the oldest timestamp
  - List
    - Keep a list of pages ordered by the time of reference
    - Head: recently used page, tail: least recently used page

  - Problems: very expensive (time & space & cost) to implement

Page Table Entry (PTE)

- PTE format (architecture specific)

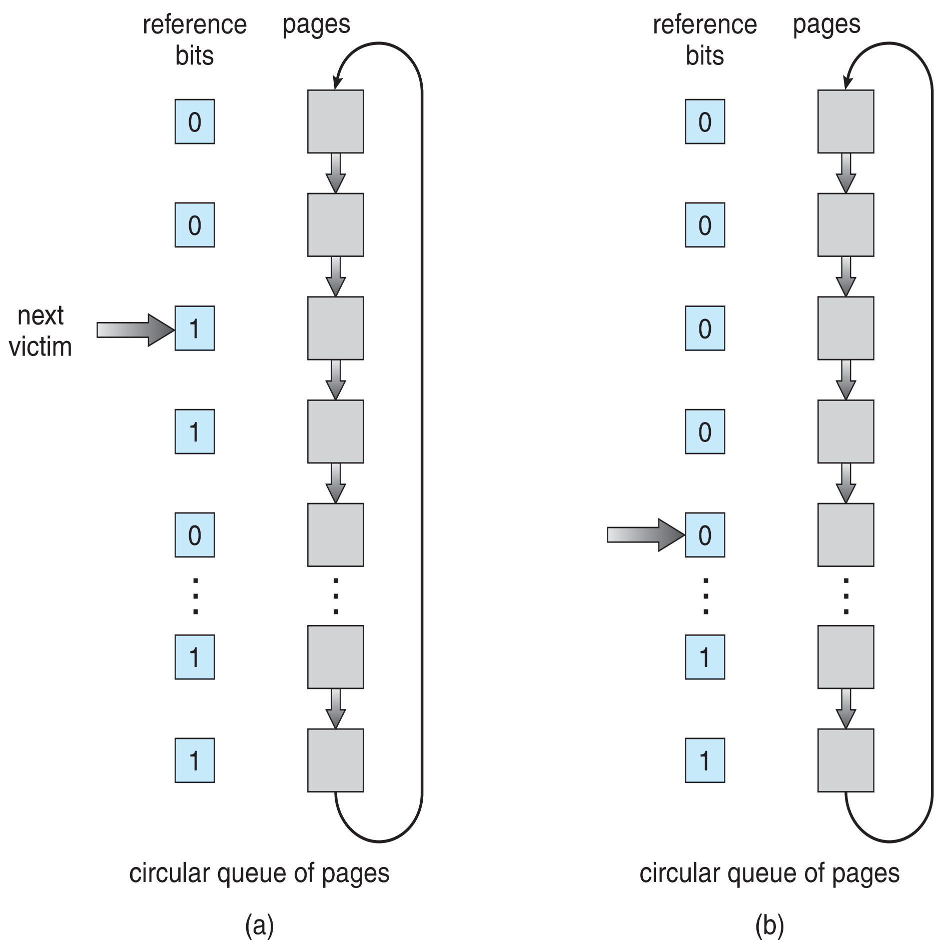
1	1	1	2	20 bits
V	M	R	P	Page Frame No

  - Valid bit (V): whether the page is in memory
  - Modify bit (M): whether the page is modified
  - **Reference bit (R): whether the page is accessed**
  - Protection bits(P): readable, writable, executable

Implementing LRU: Approximation

- Second chance algorithm (or clock algorithm)
  - Replace **an old** page, not **the oldest** page
  - Use 'reference bit' set by the MMU
- Algorithm details
  - Arrange physical page frames in circle with a pointer
  - On each page fault
    - Step 1: advance the pointer by one
    - Step 2: check the reference bit of the page:
      1 → Used recently. Clear the bit and go to Step 1
      0 → Not used recently. Selected victim. End.

Second Chance Algorithm



Implementing LRU: Approximation

- N chance algorithm
  - OS keeps a counter per page
  - On a page fault
    - Step 1: advance the pointer by one
    - Step 2: check the reference bit of the page: check the reference bit
      1 → reference=0; counter=0
      0 → counter++; if counter =N then found victim, otherwise repeat Step 1.
  - Large N → better approximation to LRU, but costly
  - Small N → more efficient but poor LRU approximation

Performance of Demand Paging

- Three major activities
  - Service the interrupt – hundreds of cpu cycles
  - **Read/write the page from/to disk – 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)

$$\begin{aligned} \text{EAT} = (1 - p) \text{ x memory access} \\ + p \text{ (page fault overhead} \\ + \text{swap page out} \\ + \text{swap page in)} \end{aligned}$$

KU THE UNIVERSITY OF KANSAS

Performance of Demand Paging

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- How to calculate EAT? (page fault probability = *p*)
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault (p = 0.001), then

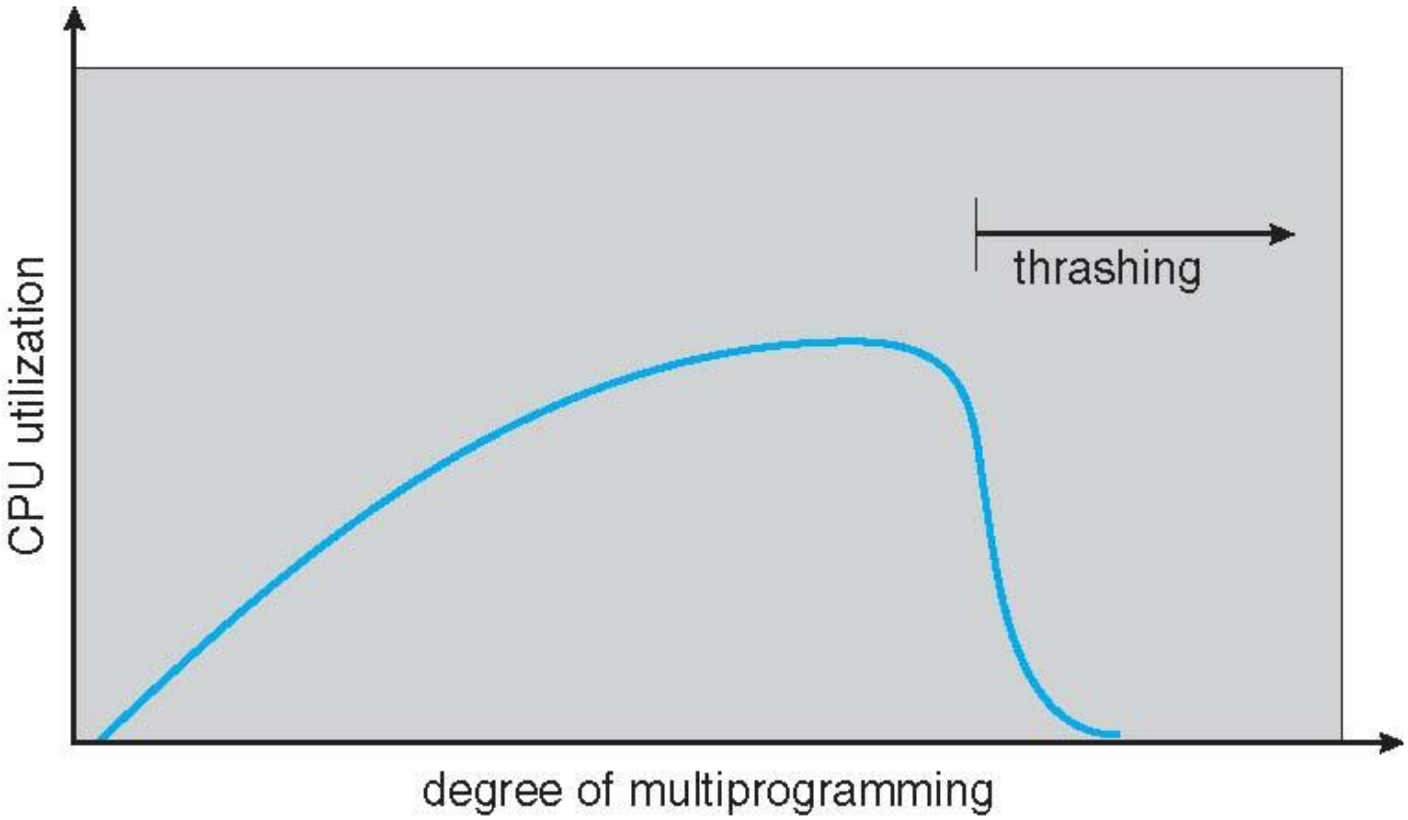
  EAT = 8.2 microseconds. → This is a slowdown by a factor of 40!!
- If you want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p   200*1.1 = 220

    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Thrashing

- A processes is busy swapping pages in and out
  - Don’t make much progress
  - Happens when a process do not have “enough” pages in memory
  - Very high page fault rate
  - Low CPU utilization (why?)
  - CPU utilization based admission control may bring more programs to increase the utilization → more page faults

Thrashing



Concepts to Learn

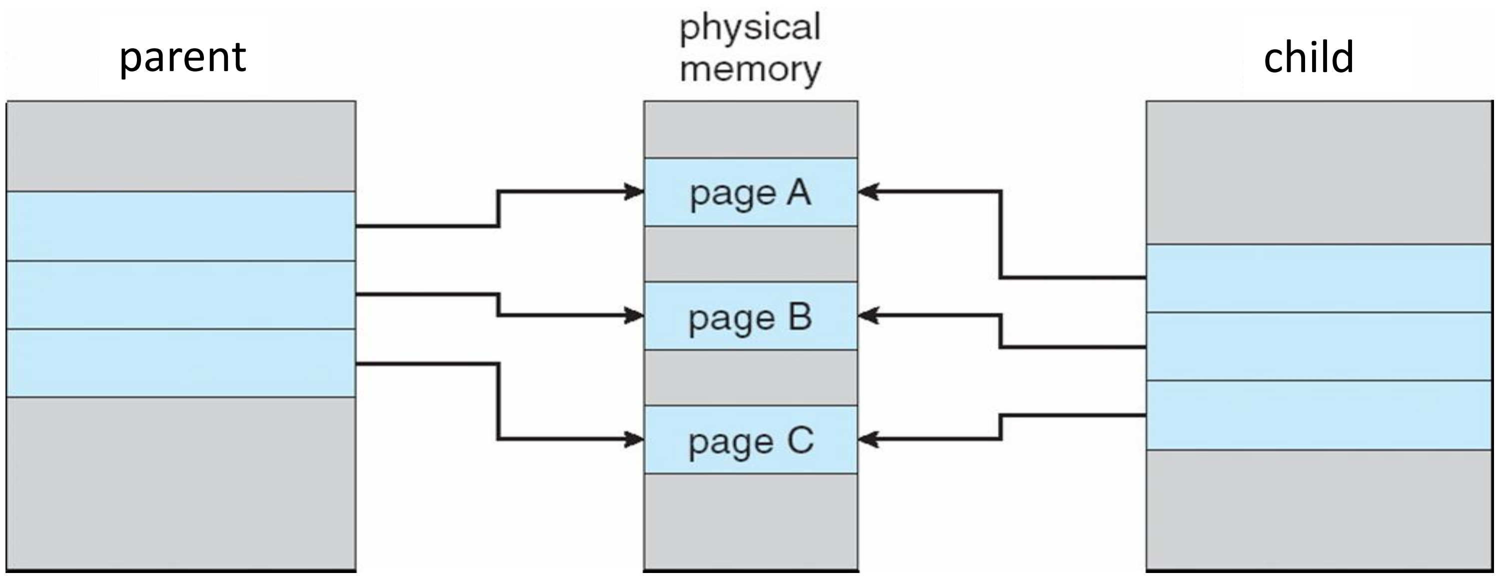
- Copy-on-Write (COW)
- Memory allocator

Copy-on-Write (COW)

- Fork() creates a copy of a parent process
  - Copy the entire pages on new page frames?
    - If the parent uses 1GB memory, then a fork() call would take a while
    - Then, suppose you immediately call exec(). Was it of any use to copy the 1GB of parent process's memory?

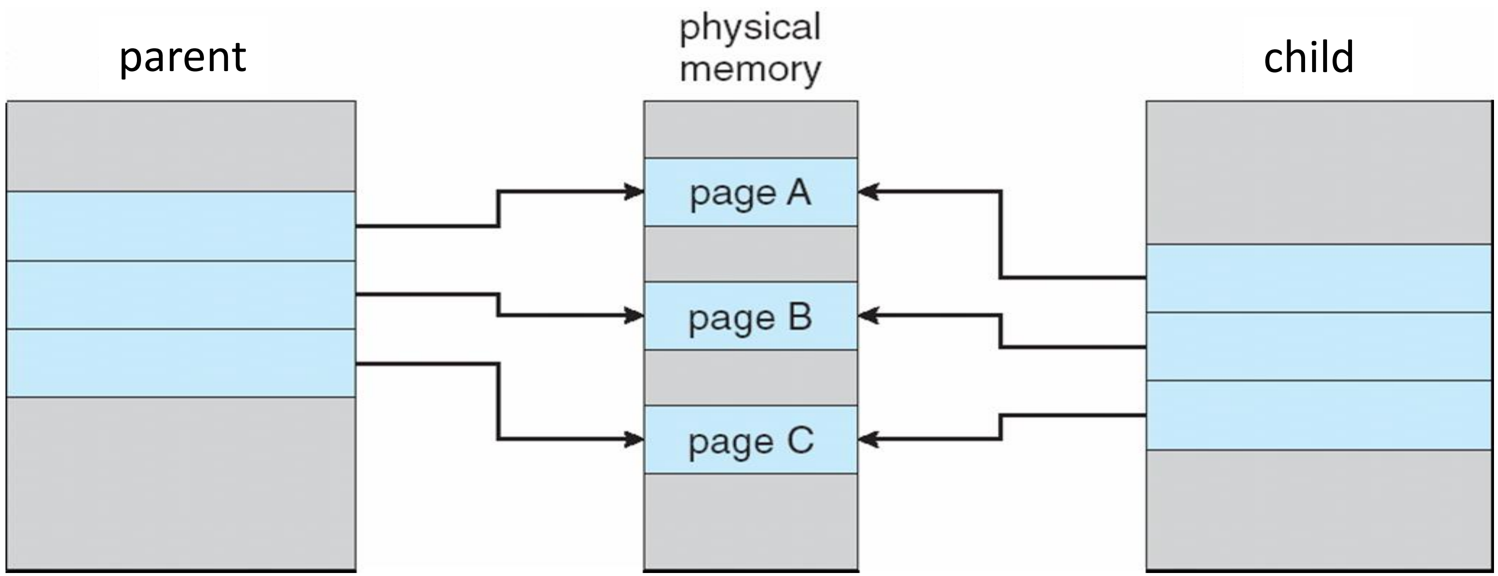
Copy-on-Write

- Better way: copy the page table of the parent
  - Page table is much smaller (so copy is faster)
  - Both parent and child point to the exactly same physical page frames



Copy-on-Write

- What happens when the parent/child reads?
- What happens when the parent/child writes?
  - Trouble!!!



Page Table Entry (PTE)

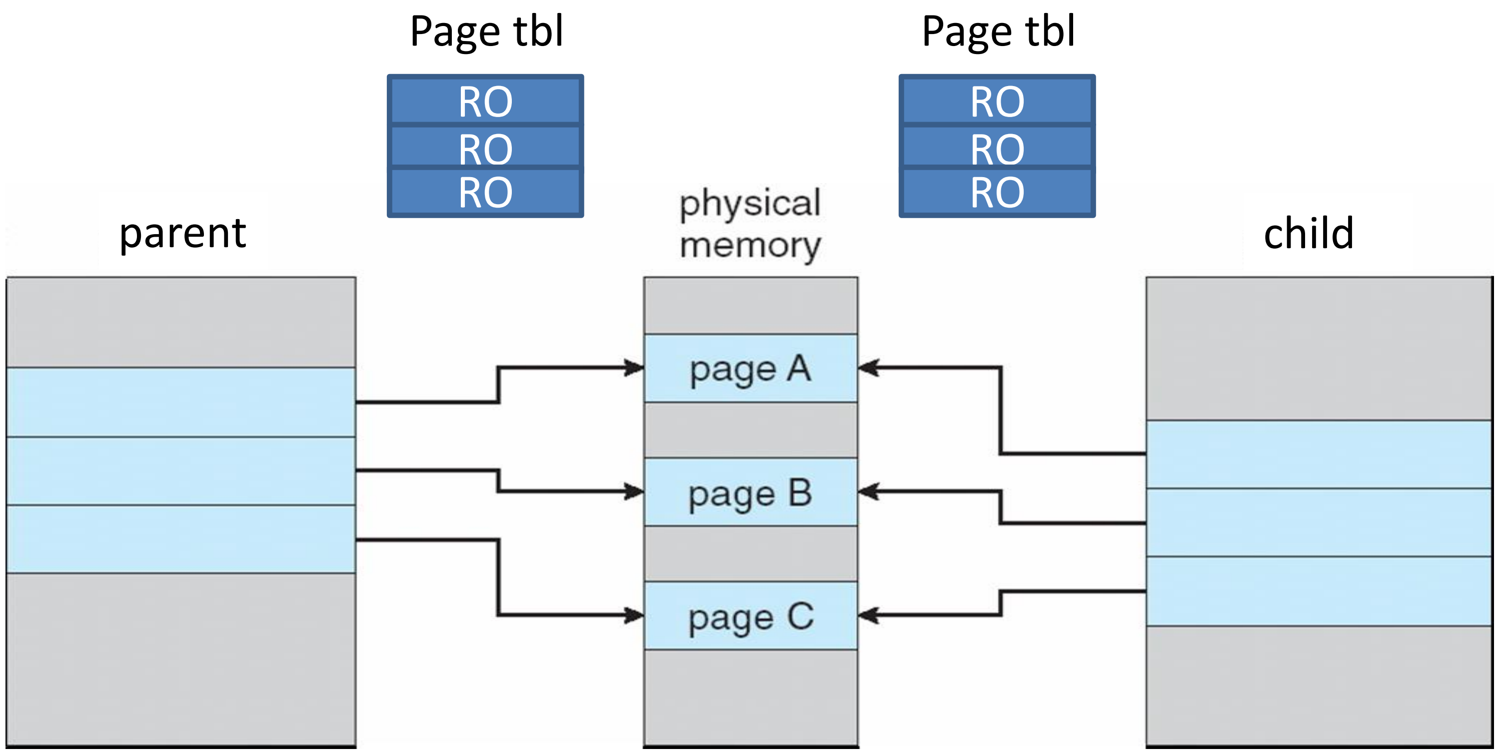
- PTE format (architecture specific)

1	1	1	2	20 bits
V	M	R	P	Page Frame No

  - Valid bit (V): whether the page is in memory
  - Modify bit (M): whether the page is modified
  - Reference bit (R): whether the page is accessed
  - **Protection bits(P): readable, writable, executable**

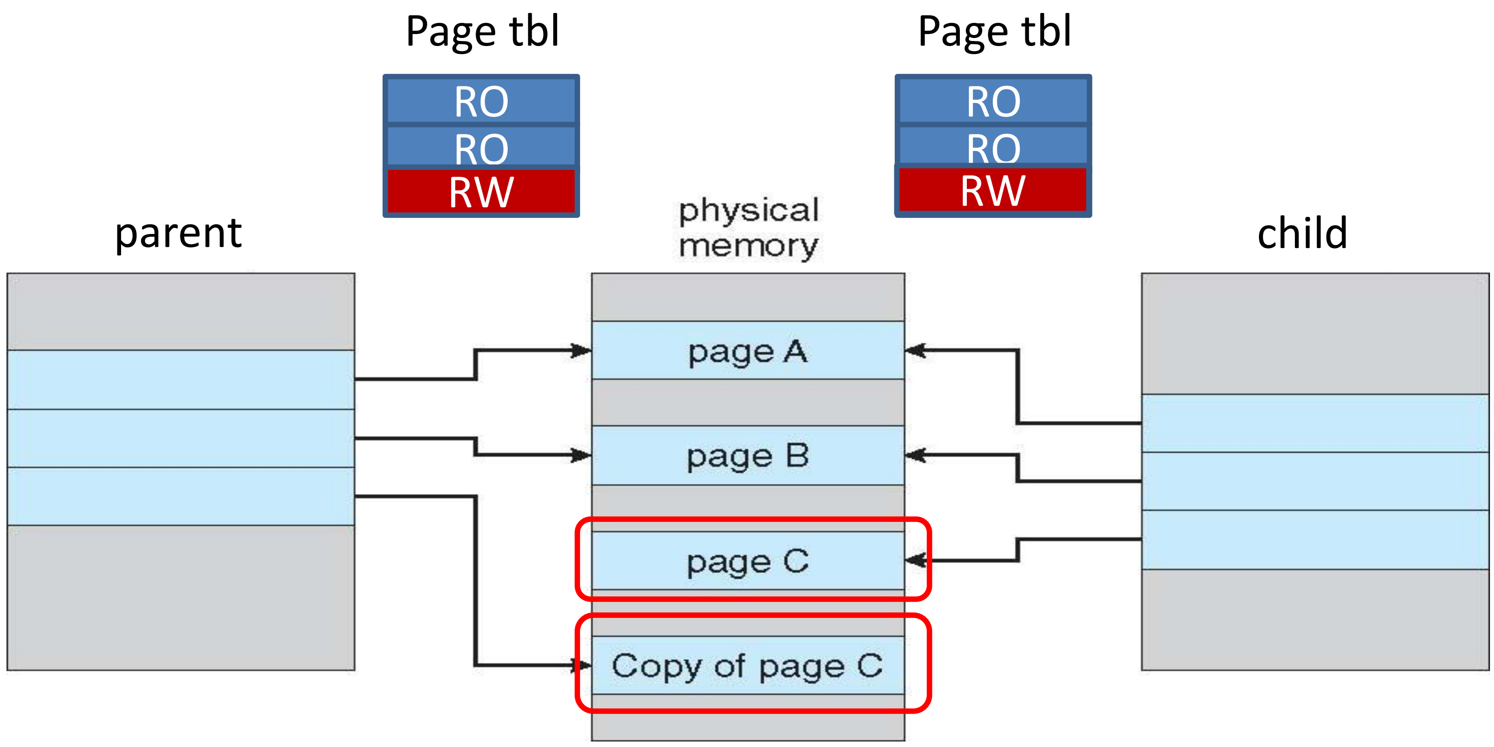
Copy-on-Write

- All pages are marked as read-only



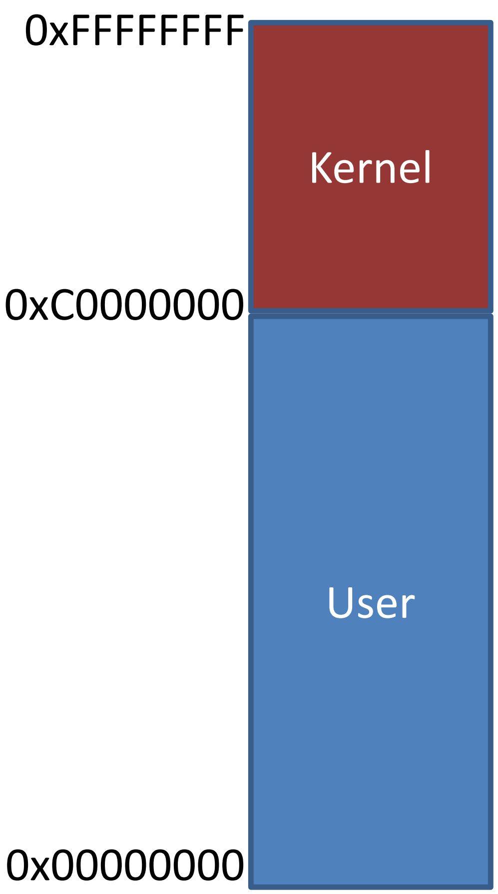
Copy-on-Write

- Up on a write, a page fault occurs and the OS copies the page on a new frame and maps to it with R/W protection setting



Kernel/User Virtual Memory

3GB for user 1GB for Kernel
User 不能影响kernel，User代码出错不会使kernel代码出错，操作系统不会奔溃



- Kernel memory
  - Kernel code, data
  - Identical to all address spaces
  - Fixed 1-1 mapping of physical memory
- User memory
  - Process code, data, heap, stack,...
  - Unique to each address space
  - On-demand mapping (*page fault*)