Virtual Memory

1DV512 – Operating Systems

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 24, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapter 9

Agenda

- Motivation and Introduction
- Demand Paging
- Page Replacement
- Virtual Memory Management Examples
- Summary

Motivation

- Code needs to be in memory to execute, but entire program rarely used
  $\Rightarrow$ e.g., error handling code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded programs
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running $\Rightarrow$ more programs run at the same time
  - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory $\Rightarrow$ each user program runs faster

Virtual Memory

- *Virtual memory* $\Rightarrow$ separation of logical memory from physical memory
- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation and for more programs running concurrently
- Less I/O needed to load or swap processes

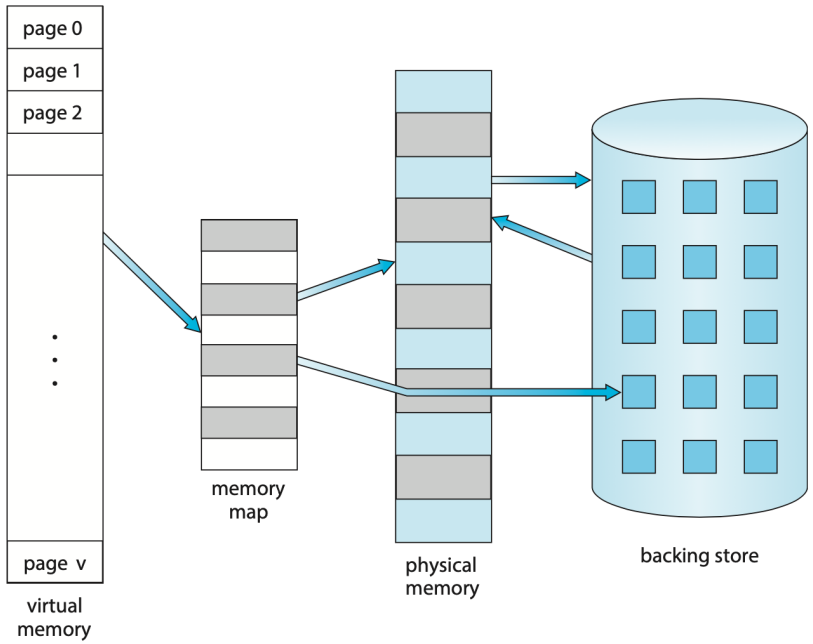

Fig. 10.1 in OSC book

Virtual Memory Implementation

- *Virtual address space* ⇒ logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until the end of space
  - Usually designed with stack starting at max logical address and grow "down" while heap grows "up" ⇒ no physical memory needed until heap or stack grows to a given new page!
  - Enables *sparse* address spaces with holes left for growth, dynamically linked libraries, etc.
- Meanwhile, physical memory organized in page frames
- MMU must map logical to physical addresses

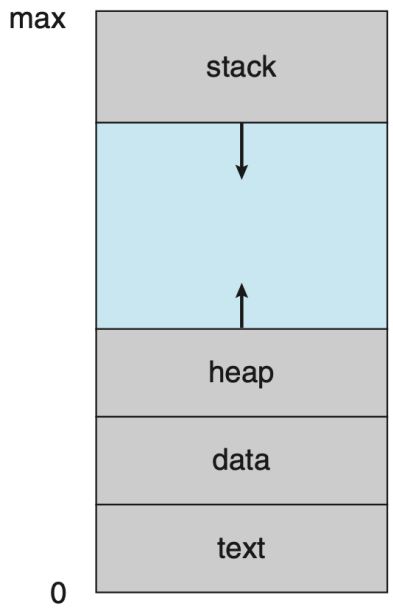

Fig. 10.2 in OSC book

Virtual Memory Implementation (cont.)

- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()` ⇒ speeding up process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation ⇒ separate or combined with paging

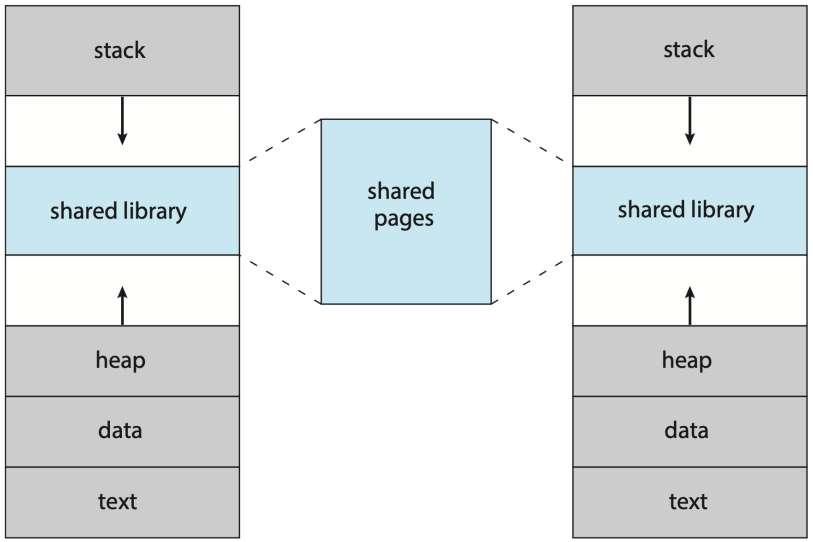

Fig. 10.3 in OSC book

Agenda

- Motivation and Introduction
- Demand Paging
- Page Replacement
- Virtual Memory Management Examples
- Summary

Demand Paging

- Could bring entire process into memory at load time...
- ...Or bring a page into memory only when it is needed ⇒ *demand paging*
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
- Similar to paging system with swapping
- *Valid-invalid bit* for each page table entry: v ⇒ in-memory (*memory resident*), i ⇒ not in memory (or invalid)

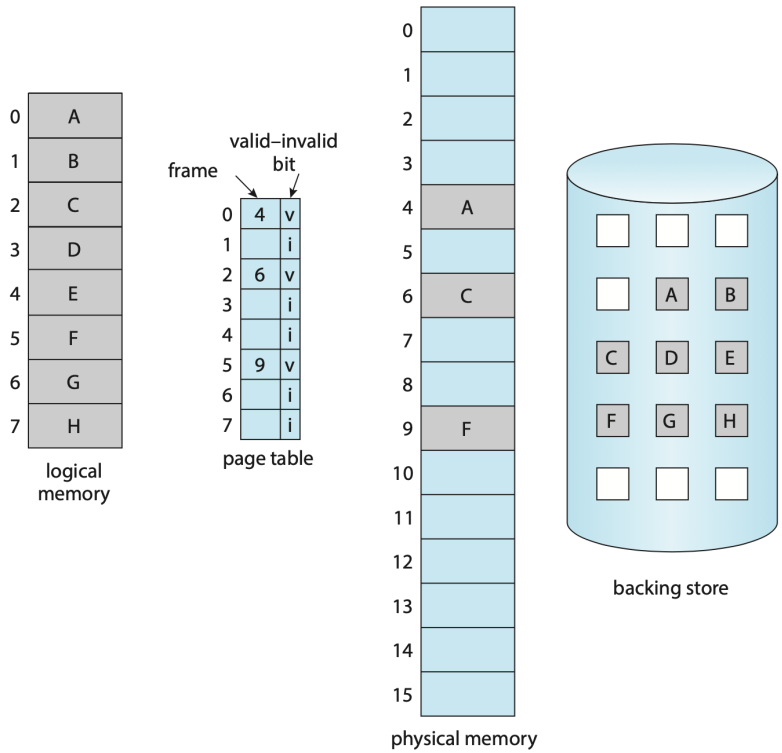

Fig. 10.4 in OSC book

Page Faults

What happens if the process tries to access a page not brought into memory?

1. During MMU address translation, if valid-invalid bit in page table entry is $i \Rightarrow$ *page fault*
2. Paging hardware will notify the OS
3. OS checks the page table (e.g., in process control block):
   - invalid address $\Rightarrow$ abort
   - not in memory $\Rightarrow$ proceed
4. Find free frame and swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory $\Rightarrow$ set validation bit $= v$
6. Restart the instruction that caused the page fault

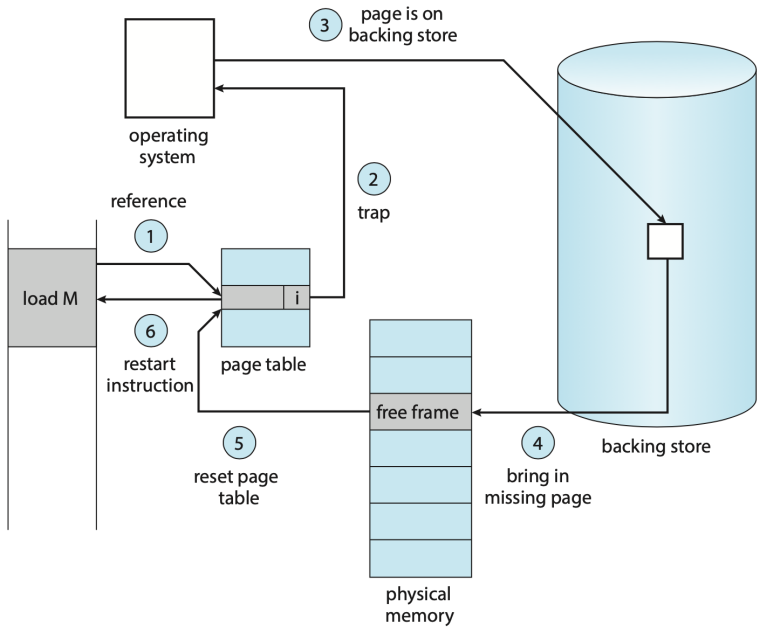

Fig. 10.5 in OSC book

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* ⇒ never bring a page into memory until it is required!
- A given instruction might 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 ⇒ same as for paging and swapping!
  - Page table with valid-invalid bits
  - Secondary memory (swap device with swap space)
  - Instruction restart ⇒ might require non-trivial architectural solutions (eager memory access, restoring memory state. . .)

Aspects of Demand Paging (cont.)

- When a page fault occurs, the operating system must bring the 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
  - *Zero-fill-on-demand* ⇒ the content of frames zeroed-out before allocated
  - On system start ⇒ all available memory is placed on free-frame list

Fig. 10.6 in OSC book

- Demand paging can degrade the performance in the worst case, if many page faults occur ⇒ processes can be preempted by OS while waiting for the page
- *Effective access time* is directly proportional to the *page-fault rate*, e.g.:
  - Let hardware memory access time = 200 ns
  - Average page-fault service time = 8 ms
  - If one access out of 1000 causes a page fault ⇒ EAT = 8.2 µs
    ⇒ slowdown by a factor of 40 !
  - For performance degradation $< 10\%$ ⇒ less than one page fault in every 400,000 memory accesses!
- Utilizing swap space I/O improves the demand paging performance

Agenda

- Motivation and Introduction
- Demand Paging
- Page Replacement
- Virtual Memory Management Examples
- Summary

Running out of Free Memory

- What happens if there is *no* free memory frame, when it's needed?
  - Used by user process pages $\Rightarrow$ *over-allocated memory*
  - Also the demand from kernel, I/O buffers, etc.
- Several design choices:
  - Terminate the process $\Rightarrow$ not a very user-friendly solution!
  - Standard swapping $\Rightarrow$ high overhead, typically not used any longer
  - Combining page swapping with *page replacement*

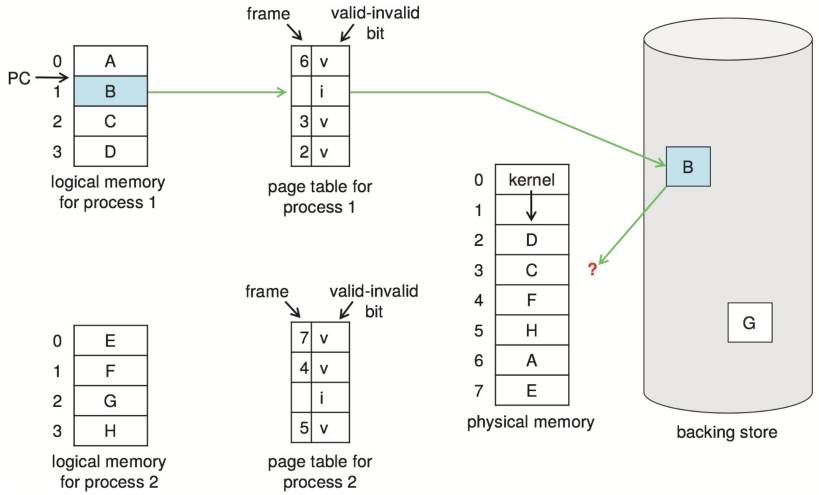

Fig. 10.9 in OSC book

Basic Page Replacement

Page replacement ⇒ find a suitable page in memory and page it out

1. Find the location of the desired page on disk
2. Find a free frame or select a *victim frame* and write it to disk if necessary ⇒ use the *modify bit* (*dirty bit*) to reduce overhead of page transfers for non-modified pages
3. Read the desired page into the frame, and update the page and frame tables
4. Continue the process from the faulted instruction

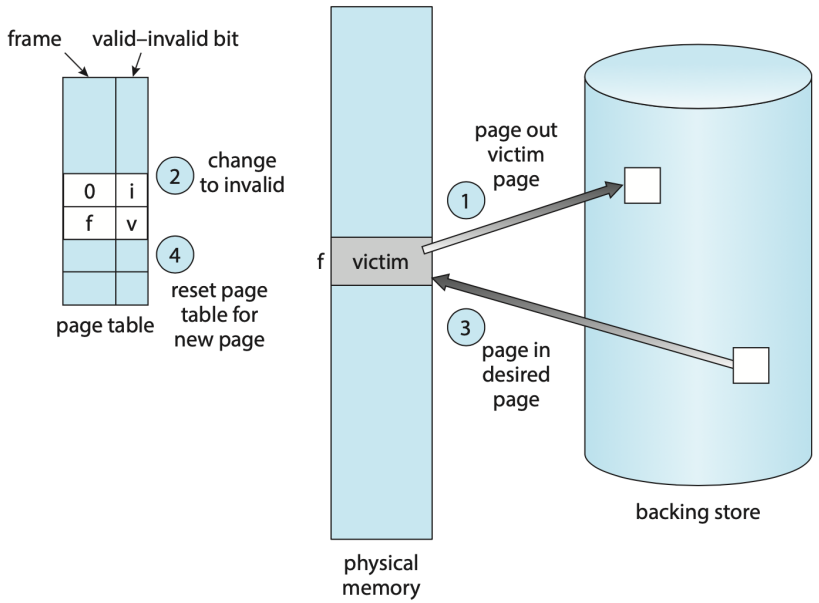

Fig. 10.10 in OSC book

Page replacement is basic for demand paging — and it completes the separation between logical memory and physical memory

Page Replacement Algorithms

- The choice of particular *page-replacement algorithm* ⇒ aiming for the lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references *(reference string)* and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

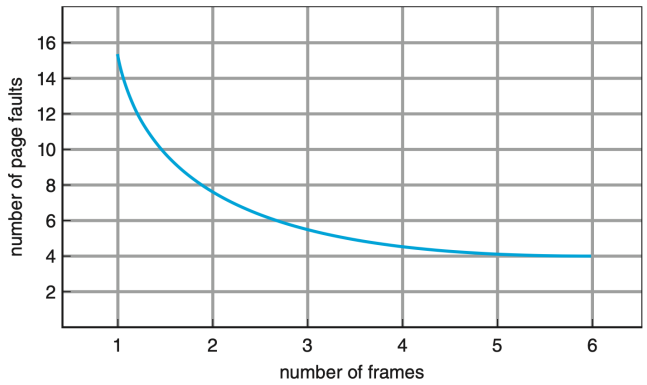

Fig. 10.11 in OSC book

The reference string for the examples below:
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

FIFO Page Replacement

- *First-in, first-out* ⇒ the oldest page is chosen for replacement
- How to track ages of pages? ⇒ Just use a FIFO queue!
- For our example reference string, our three frames are initially empty
- In total: 15 page faults
- *Belady's anomaly* ⇒ adding more frames can cause more page faults! (e.g., the chart on the right for another ref. string)

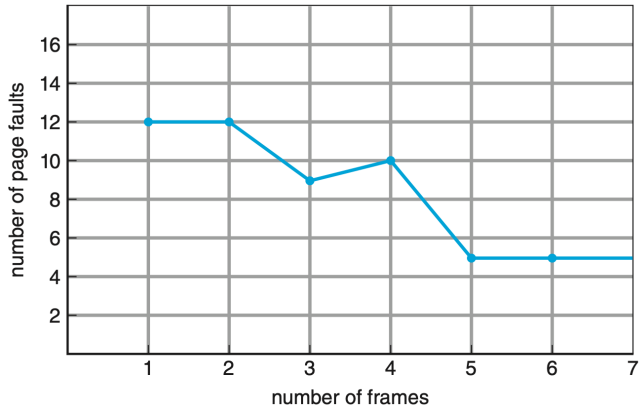

Fig. 10.13 in OSC book

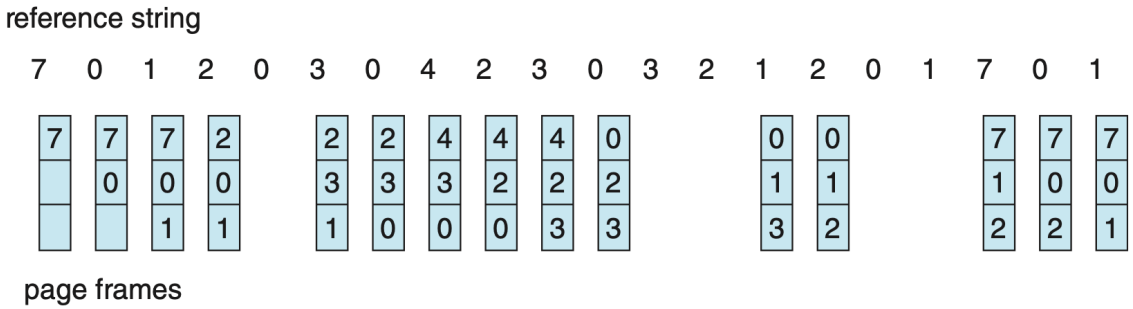

Fig. 10.12 in OSC book

Optimal Page Replacement

- *Optimal page replacement* ⇒ replace page that will not be used for longest period of time
- For our example reference string, 9 page faults in total is minimal
- How do you know this? ⇒ Can't read the future! (cf. SJF scheduling algorithm...)
- Not used in practice, but rather as a benchmark

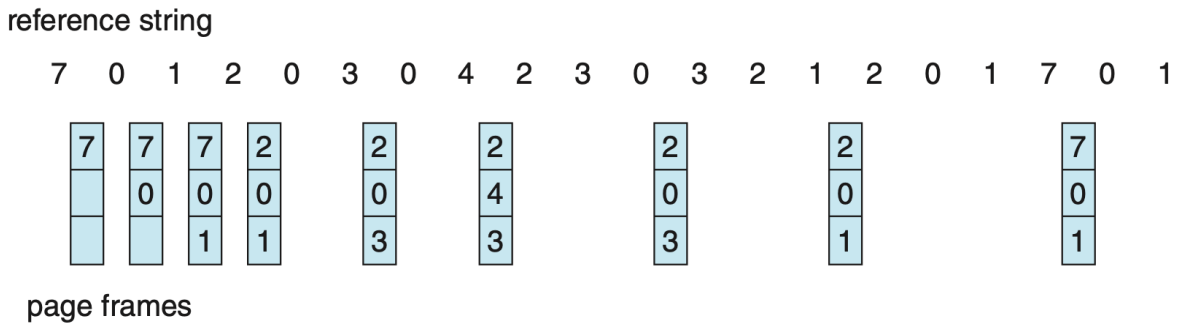

Fig. 10.14 in OSC book

LRU Page Replacement

- *Least Recently Used (LRU) page replacement* ⇒ use past knowledge rather than future
- Replace page that has not been used in the most amount of time ⇒ associate time of last use with each page
- For our example reference string, 12 faults ⇒ better than FIFO, but worse than OPT
- Generally good algorithm and frequently used
- Implementation approaches:
  - *Counter* implementation ⇒ every page entry has an associated counter or logical clock value
  - *Stack* implementation ⇒ a stack of page numbers in a double linked form

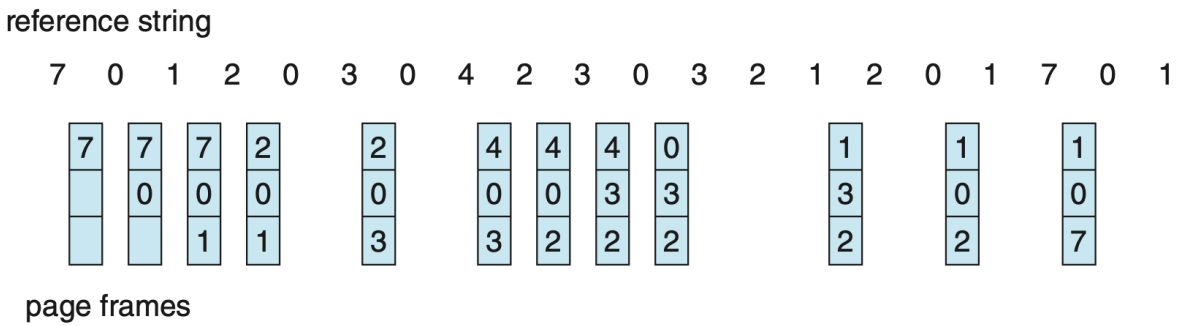

Fig. 10.15 in OSC book

Agenda

- Motivation and Introduction
- Demand Paging
- Page Replacement
- Virtual Memory Management Examples
- Summary

Virtual Memory in Popular OSs

- Linux
  - Demand paging ⇒ allocating pages from a list of free frames
  - A page-replacement policy similar to the LRU approximation
  - Each page has an accessed bit that is set whenever the page is referenced
  - Two types of page lists: an *active list* and an *inactive list*
  - If free memory falls below a certain threshold, the kernel starts reclaiming pages on the inactive list
- Windows
  - Demand paging with *clustering* ⇒ clustering brings in pages surrounding the faulting page
  - A page-replacement policy similar to the LRU approximation
  - Processes are assigned *working set minimum* (minimum number of guaranteed pages) and *working set maximum*

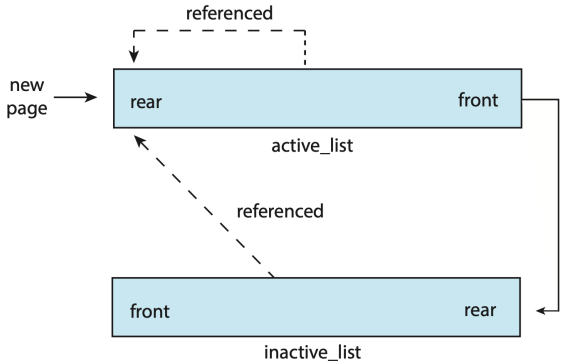

Fig. 10.29 in OSC book

Copy-on-Write

- When starting processes with the `fork()` approach ⇒ demand paging for the initial page might be avoided
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages ⇒ *if* either process modifies a shared page, only *then* it is copied!
- Pages that cannot be modified (pages containing executable code) can be shared by the parent and child
- Copy-on-write is a common technique used by multiple OSs, incl. Linux, Windows, and macOS
- Several OS support additional `vfork()` system call ⇒ the parent process is suspended, and the child uses its address space (until invoking `exec()` typically) ⇒ no copying of pages even takes place!

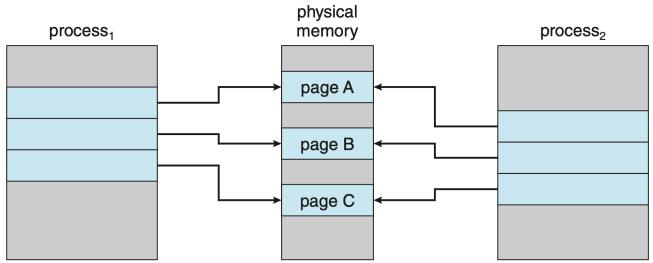

Fig. 10.7 in OSC book

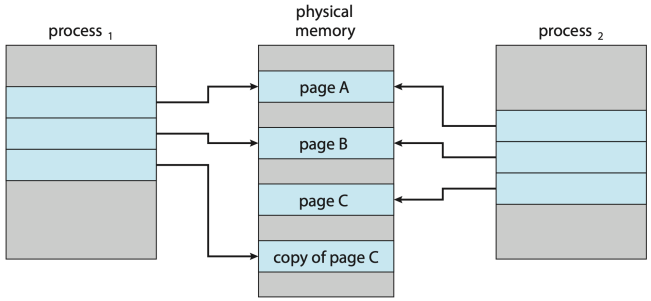

Fig. 10.8 in OSC book

Agenda

- Motivation and Introduction
- Demand Paging
- Page Replacement
- Virtual Memory Management Examples
- Summary

Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage ⇒ multiple benefits, but careful implementation is necessary!
- A page fault occurs when a page that is currently not in memory is accessed ⇒ must be brought from the backing store into a memory frame
- When available memory runs low, a page-replacement algorithm selects an existing page to replace ⇒ algorithms include FIFO, optimal, and LRU
- *Thrashing* ⇒ the system spends more time paging than executing
- *Kernel memory* is allocated differently than user-mode processes ⇒ memory allocated in contiguous chunks according to a strategy such as buddy allocation or slab allocation
- Many modern OSs (Linux, Windows, Solaris, . . .) handle virtual memory in a similar way