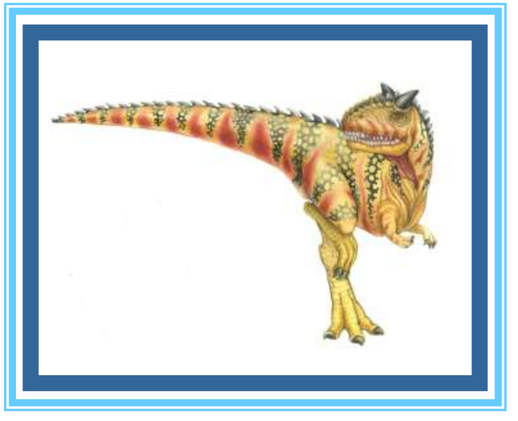
Virtual Memory



Slides borrowed from Galvin, with many of our modifications

Recap: What a programmer wants from memory

- Memory addresses will start from 0
- Memory will be contiguous
- I should be able to dynamically increase the memory
- I should have as much memory as I want (**an illusion**)

First 3 requirements satisfied with paging

I may even want more memory than the physical mem size. I do not want to be limited to the physical memory size on a particular system.

Virtual memory – next few classes

Recap: What a programmer wants from memory

- Memory addresses will start from 0
- Memory will be contiguous

First 3 requirements satisfied with paging

- I should be able to dynamically increase the memory
- I should have as much memory as I want (**an illusion**)

I may even want more memory than the physical mem size. I do not want to be limited to the physical memory size on a particular system.

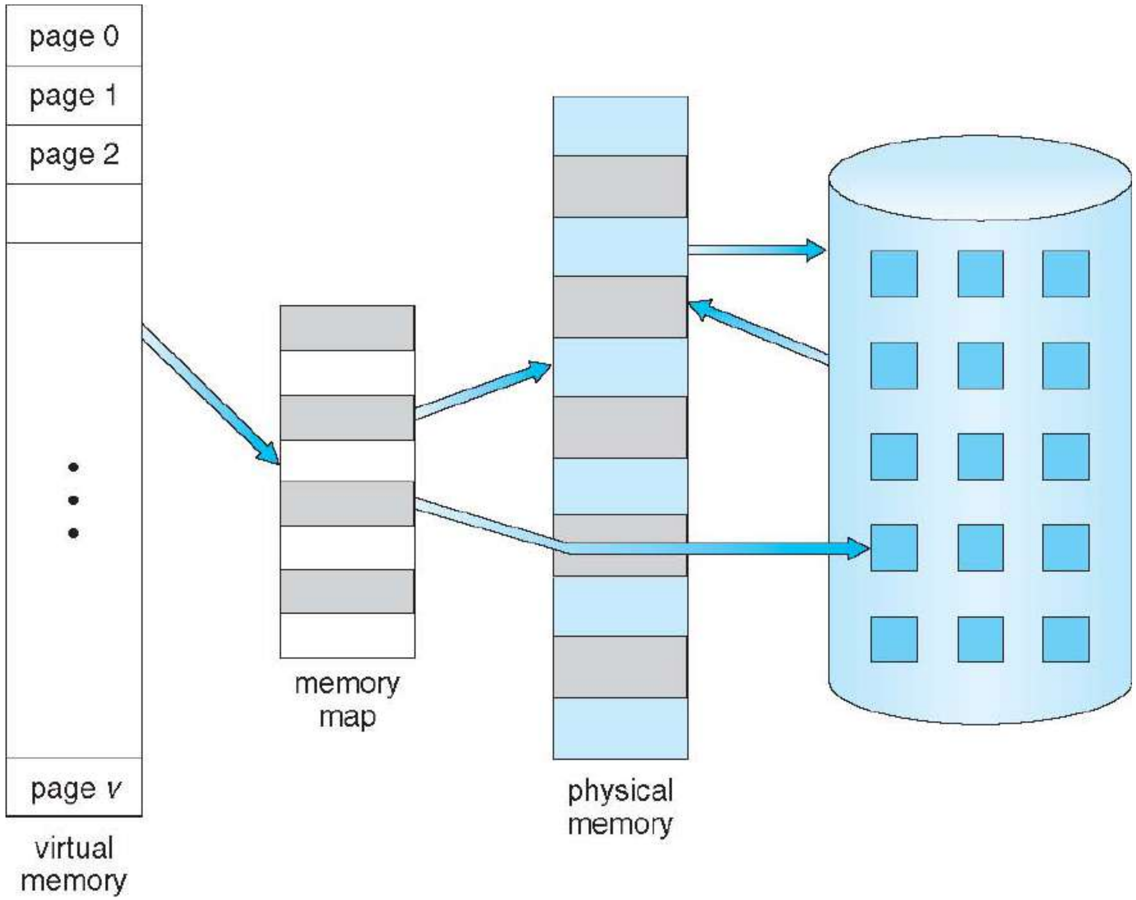
Virtual memory – next few classes

Basic idea behind creating this illusion

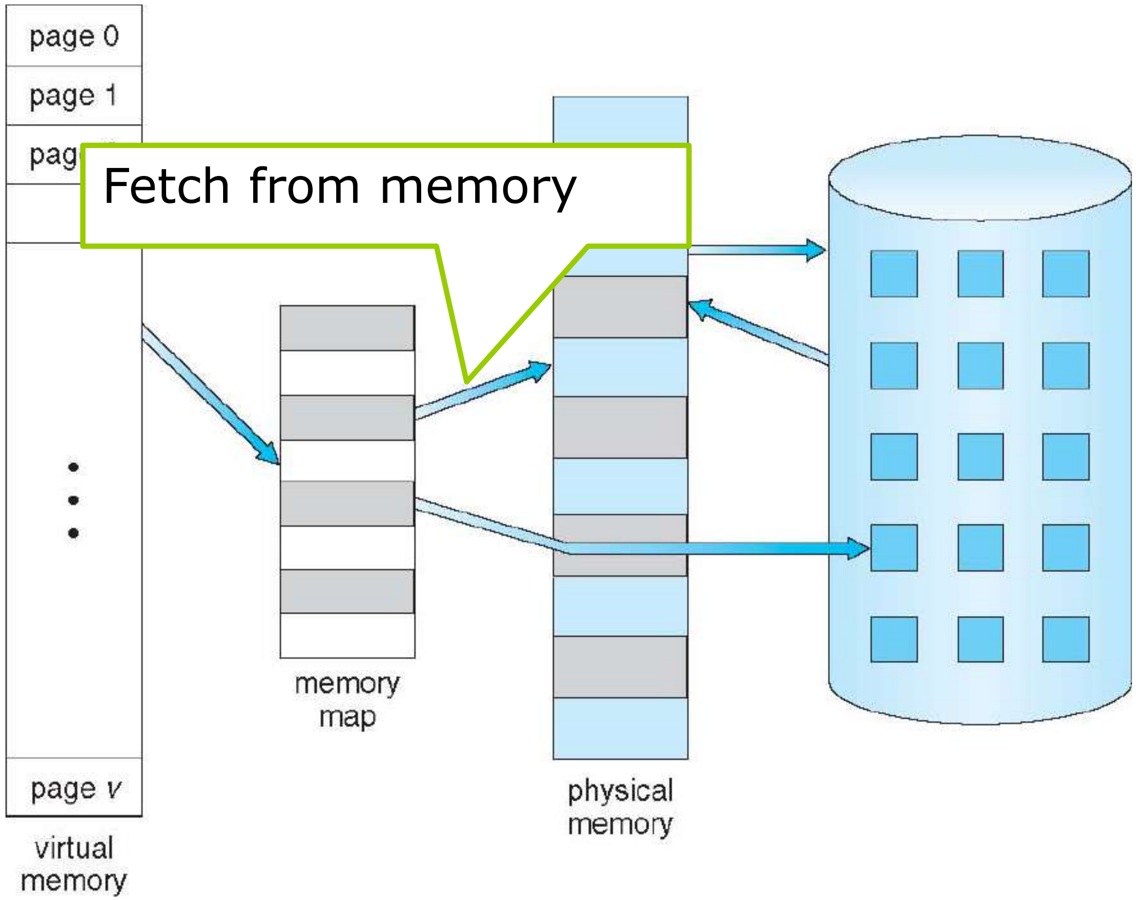
- OS will stash away pages which are not used right now and resume them when needed by processes
  - Need a large space which can store those extra pages
  - Natural choice: Your hard disk (or SSD)!

- Rationale: Code needs to be in memory to execute
  - BUT entire program is rarely used
  - All pages not needed at same time
  - OS should evict pages (from main memory), store and retrieve (to/from hard disk) pages as needed

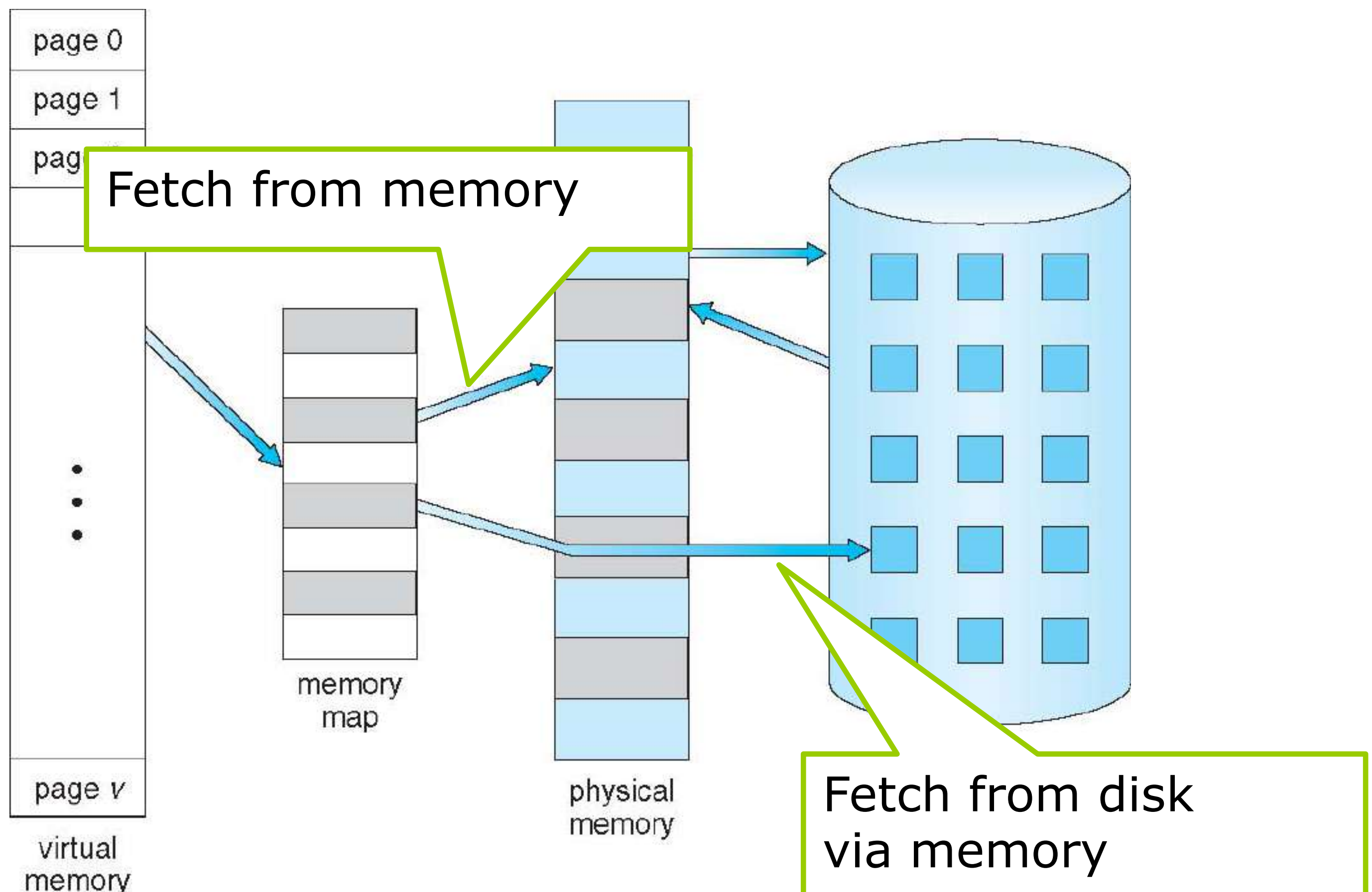
Virtual Memory That is Larger Than Physical Memory



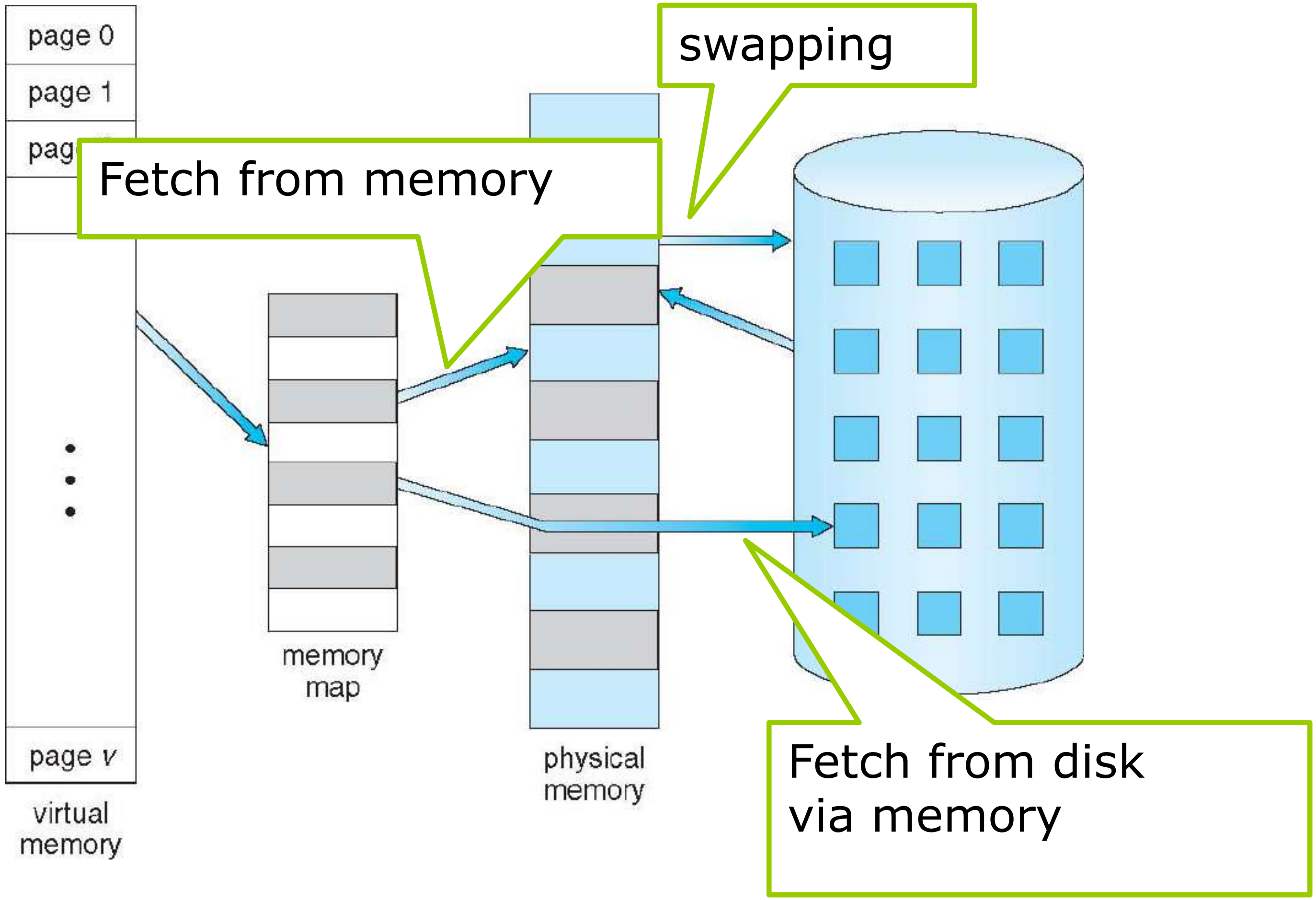
Virtual Memory That is Larger Than Physical Memory



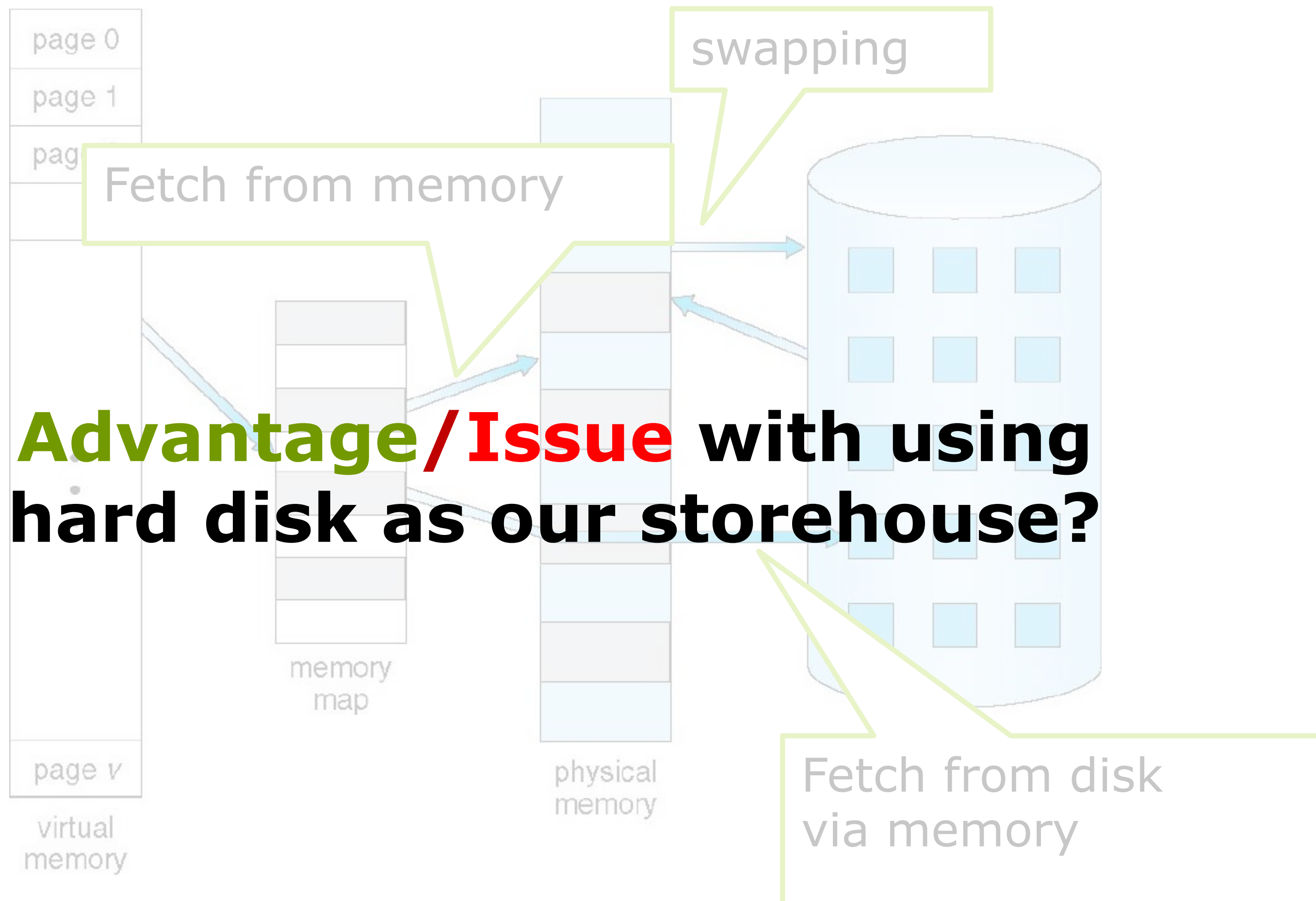
Virtual Memory That is Larger Than Physical Memory



Virtual Memory That is Larger Than Physical Memory



Virtual Memory That is Larger Than Physical Memory



Advantage/Issue with using hard disk as our storehouse?

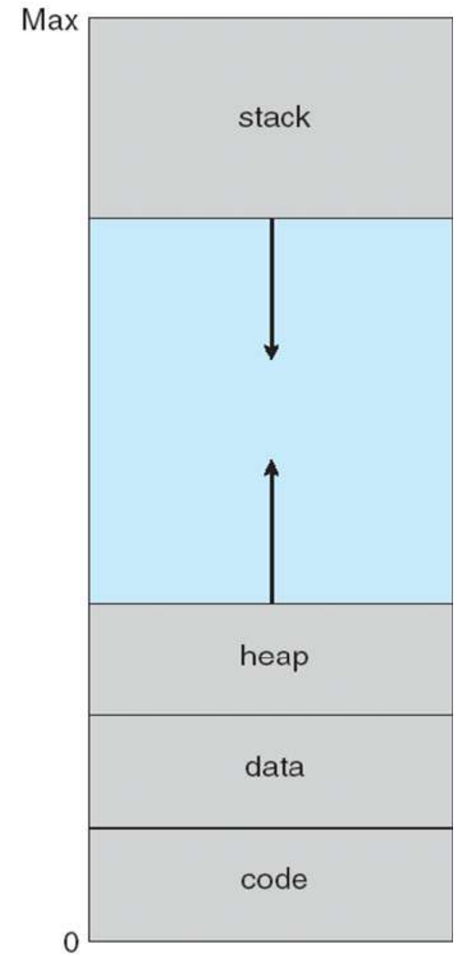
Trade off: Advantage

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames

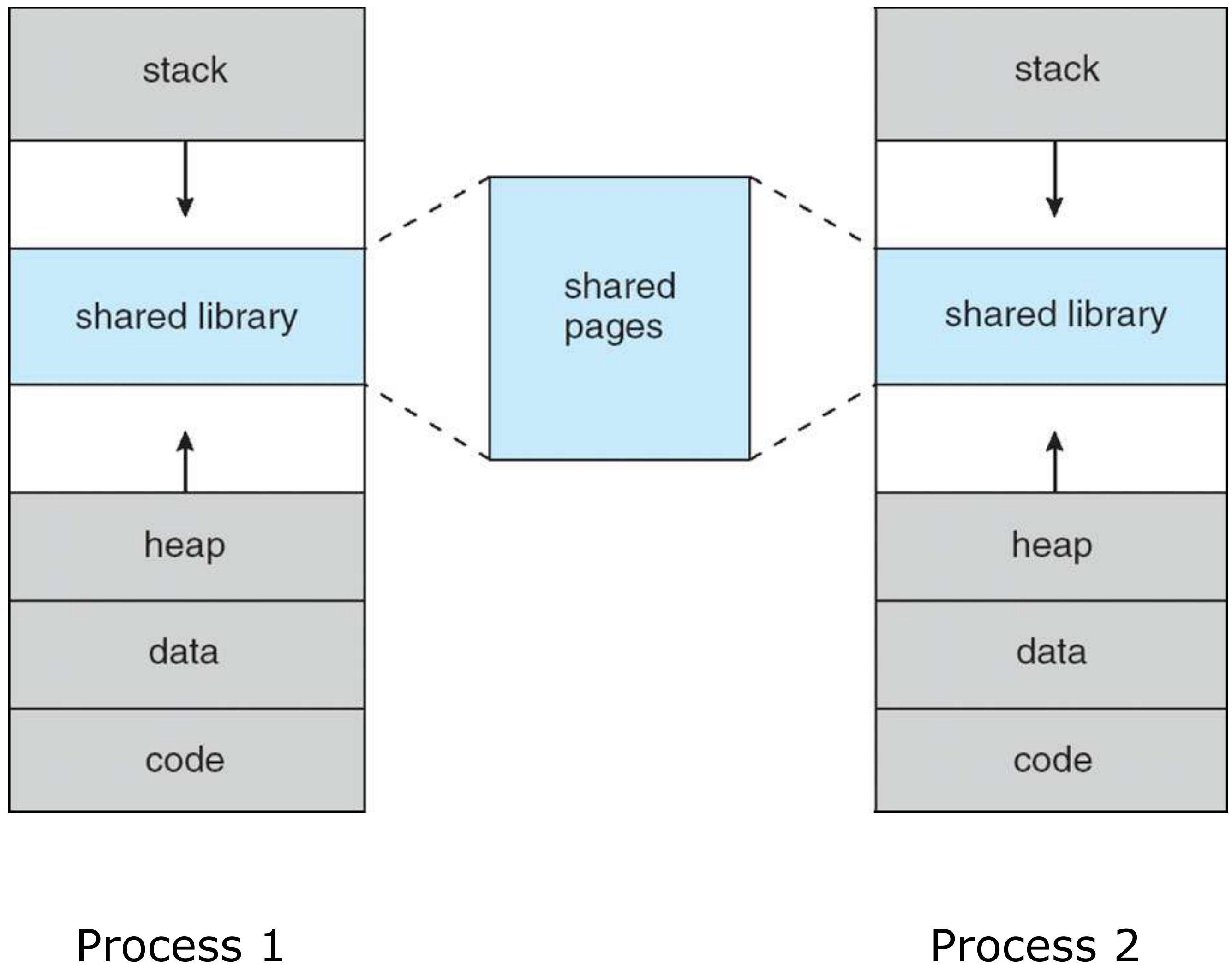
Virtual address space size = physical memory size + space allocated in you HDD (swap space)

Virtual-address Space: properties

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- 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 process creation

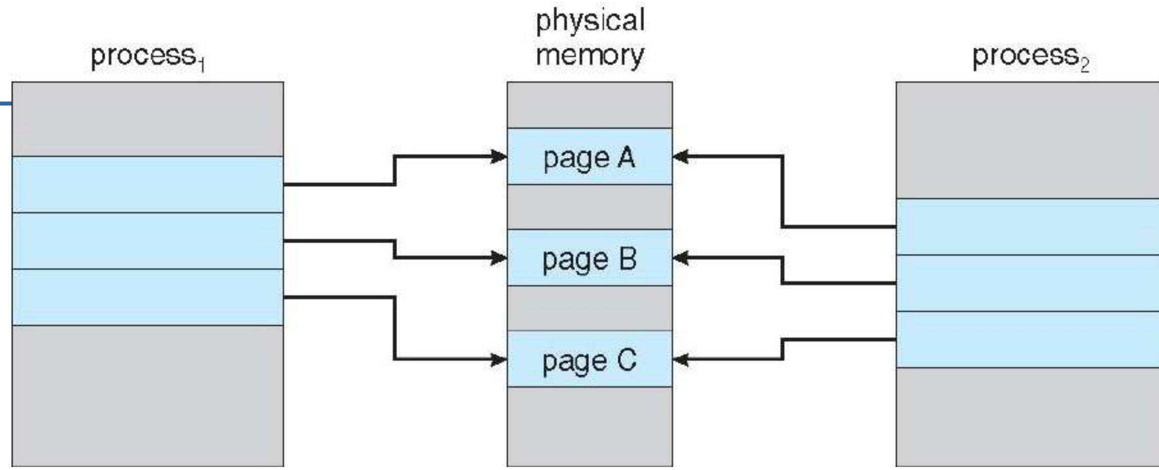


Shared Library Using Virtual Memory

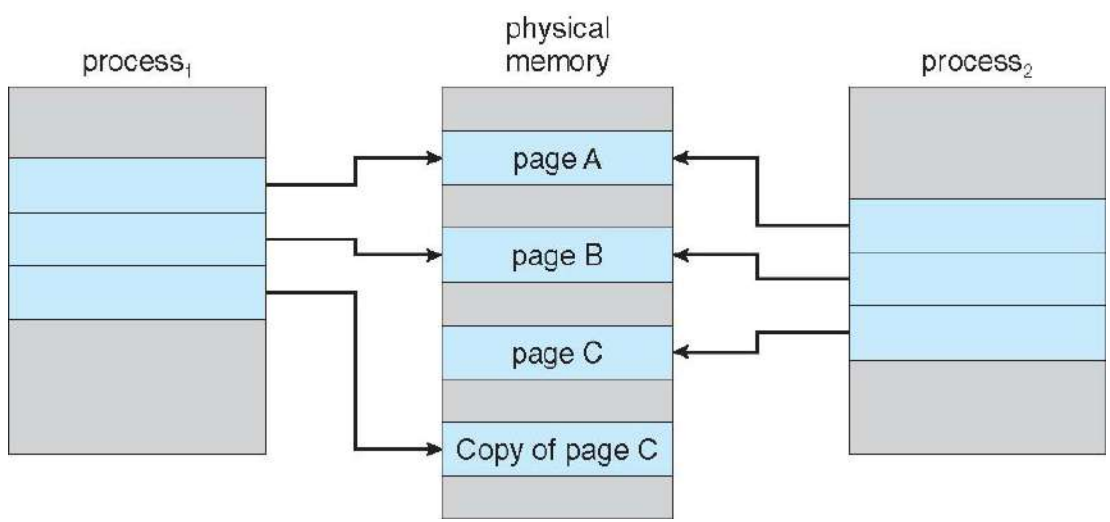


Optimization: Copy-on-Write

- **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
  - Allows more efficient process creation as only modified pages are copied
- **`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

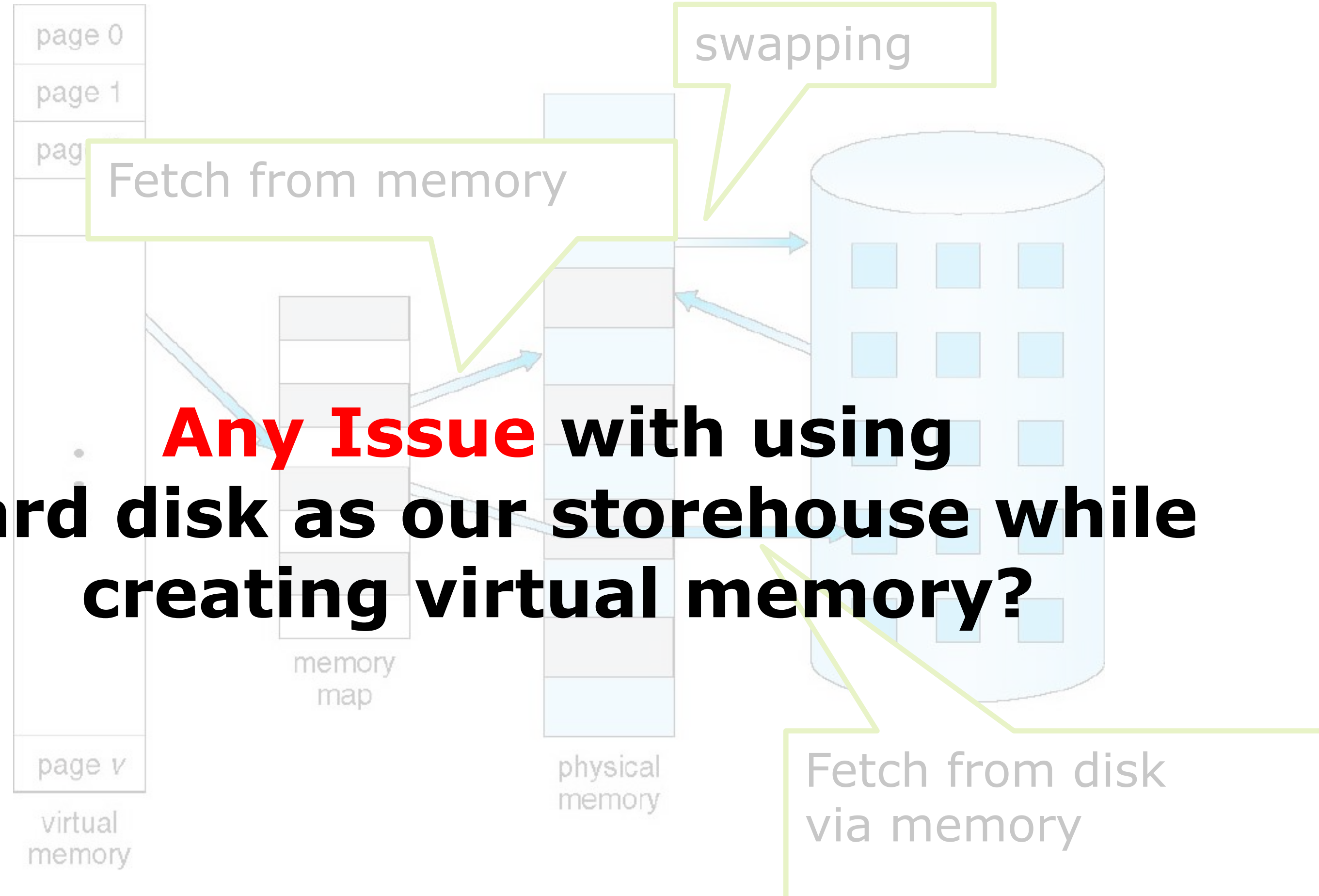


Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Virtual Memory That is Larger Than Physical Memory



Any Issue with using hard disk as our storehouse while creating virtual memory?

Trade off: Issue

Hard disk read/write is extremely slow compared to main memory (50 – 200 times slower)

OS challenge

- How would OS use this larger, slower hard disk device to transparently provide user the illusion of a large virtual address space

OS challenge

- How would OS use this larger, slower hard disk device to transparently provide user the illusion of a large virtual address space
  - Requirement 1: Incorporate swapping into logical to physical address space mapping process
  - Requirement 2: Determine which pages to swap and when
  - Requirement 3: Determine how to allocate free frames to processes (user processes and kernel)

Demand Paging

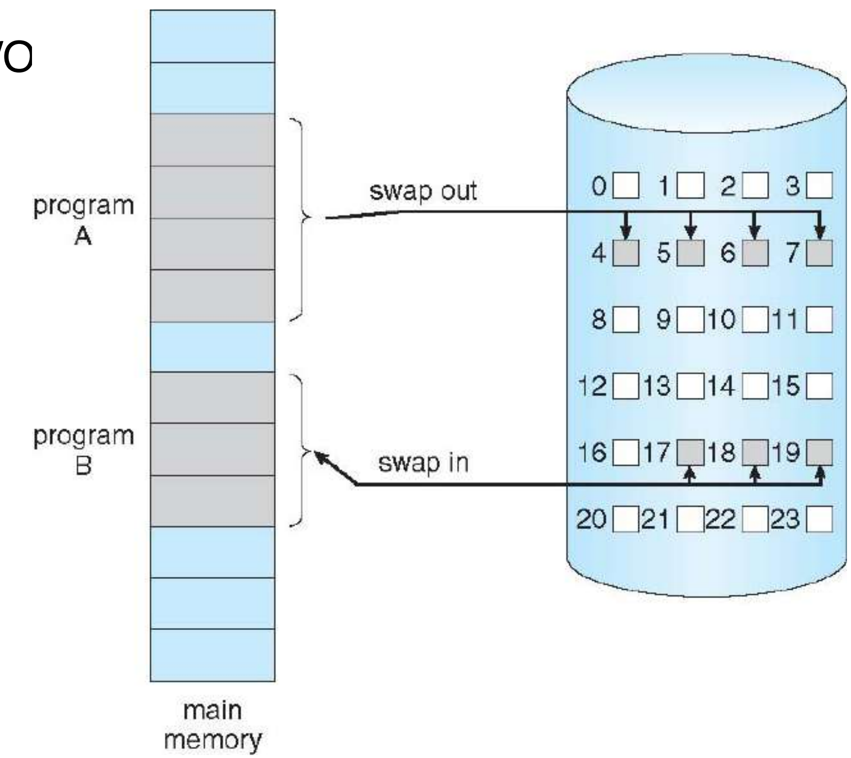
- Could bring entire process into memory at load time
- OR bring a page into memory only when it is needed: demand paging
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

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
  - More users
- We will swap pages

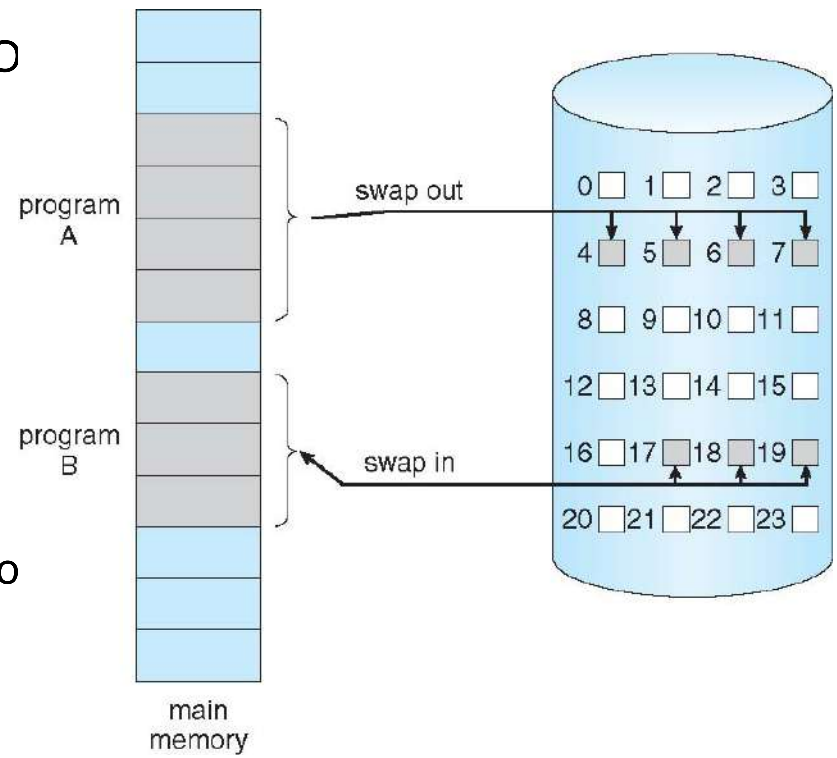
Demand Paging

- 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
  - More users
- We will swap pages



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
  - More users
- We will swap pages
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is **pager**



Sketch of how pager should work in demand paging

- With swapping, pager guesses which pages will be used before swapping out again
- Instead of whole process, pager brings in only those pages into memory
- How to determine that set of pages?
  - **That is our requirement 2 (later)**
- If pages needed are already **memory resident**
  - No difference from non demand-paging scenario
- 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

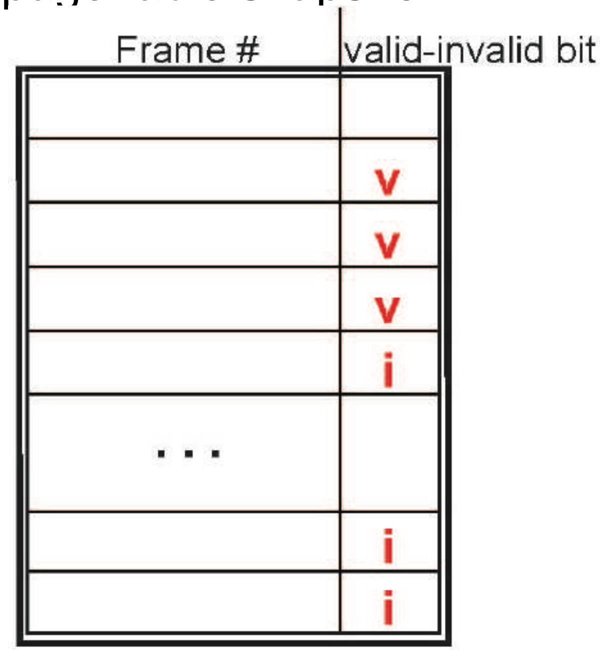
Sketch of how pager should work in demand paging

- With swapping, pager guesses which pages will be used before swapping out again
- Instead of whole process, pager brings in only those pages into memory
- How to determine that set of pages?
  - **That is our requirement 2 (later)**
- If pages needed are already **memory resident**
  - No difference from non demand-paging scenario
- 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

Need two new machineries
- Valid/Invalid bit
- Page fault handler

Valid-Invalid Bit

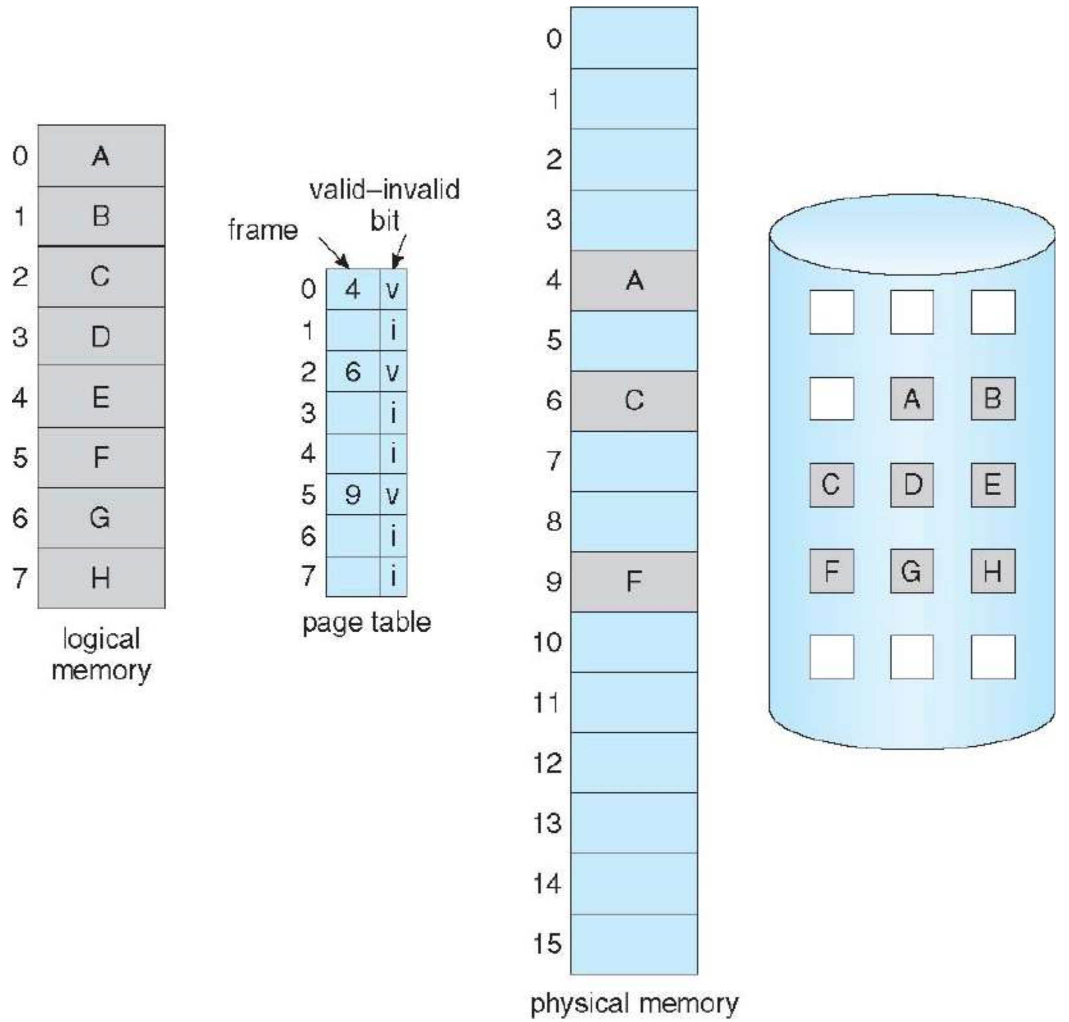
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



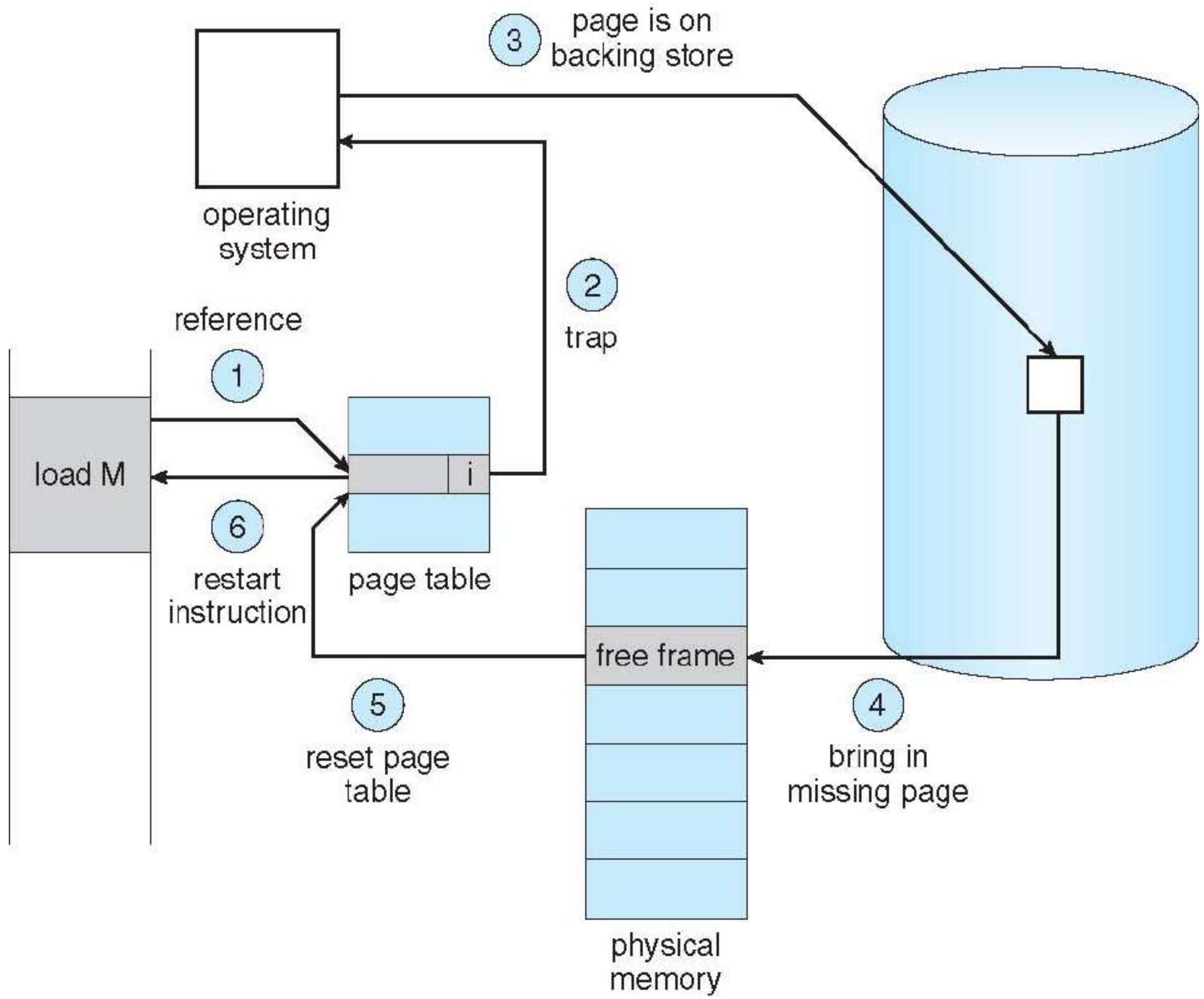
Page Fault

- If there is a reference to a page which is not in main memory

page fault

1. Operating system decides:
   - Invalid reference ⇒ abort
   - Just not in memory (using valid/invalid bit)
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



Locating disk address

- How would the page fault handler know which disk address to access in case of swapping in a page from disk to main memory?

Locating disk address

- How would the page fault handler know which disk address to access in case of swapping in a page from disk to main memory?

"*When a page is swapped out, Linux uses the corresponding Page table Entry (PTE) to store enough information to locate the page on disk again. Obviously a PTE is not large enough in itself to store precisely where on disk the page is located, but it is more than enough to store an index into the swap_info array and an offset within the swap_map and this is precisely what Linux does*." -- https://www.kernel.org/doc/gorman/html/understand/understand014.html

How would "Demand Paging" work?

- **Pure Demand Paging** – start process with *no* pages in memory
  - OS sets PC to first instruction of process
    - Page not in memory → ***page fault***
  - Similarly, for every other pages on ***first access*** *(cause they are not in memory when references for first time)*
- A given instruction can access multiple pages → ***multiple page faults***

```
LOAD    R2,100(R5)
STORE   R5,50(R10)
```

- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (**swap space**)
  - Instruction restart
    - Re-execute instruction that resulted in page fault

Complete steps of Demand Paging

Stages in Demand Paging (worst case)

1. **Page Fault** → 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 disk
5. Issue a read from the disk to a free frame
6. While waiting, allocate the CPU to some other process
7. Receive an interrupt from the disk I/O subsystem (***I/O completed***)
8. Save the registers and process state for the other process
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

- 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 $0 \le p \le 1$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access time

  + p (page fault overhead + ***swap page out*** + swap page in)

Only if the replaced page is dirty

Demand Paging :: Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 nsec + p (8 msec)

  = (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, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 %

  220 > 200 + 7,999,800 x p

  or, 20 > 7,999,800 x p

  or, p < .0000025

  < one page fault in every 400,000 memory accesses

OS challenge

- How would OS use this larger, slower hard disk device to transparently provide user the illusion of a large virtual address space
  - Requirement 1: Incorporate swapping into logical to physical address space mapping process

    Demand paging with
    valid-invalid bit / page fault handler

  - Requirement 2: Determine which pages to swap and when
  - Requirement 3: Determine how to allocate free frames to processes (user processes and kernel)

Need to swap if there is no Free Frame

- All pages are used up by the processes
- May also be used by the kernel, I/O buffers, etc.
- Page replacement – find some page in memory, but not ***really*** in use, page it out
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

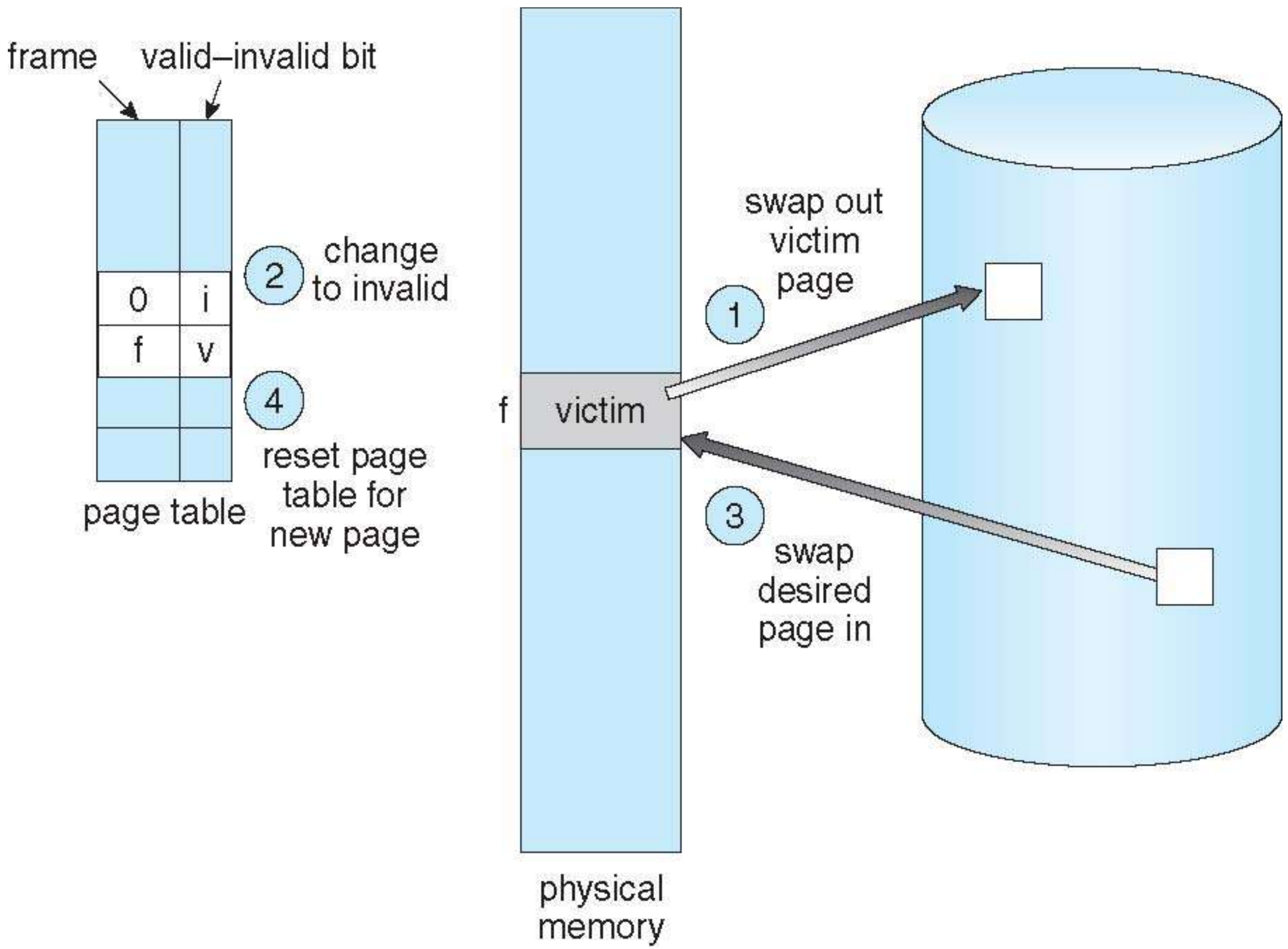
- Prevent ***over-allocation*** of memory by modifying page-fault service routine to include page replacement
  - Too many pages should not be allocated to a process
- Use ***modify (dirty) bit*** to reduce overhead of page transfers
  - When a page is modified, it is marked as dirty
  - only modified pages are written to disk (stored in swap space)
  - *If not dirty, why is it not stored?*
- Page replacement completes separation between logical memory and physical memory
  - Large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

1. Find the location of the desired page on disk
2. 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
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



A Simple Calculation

- Let
  - p : probability of page fault
  - δ : probability that victim page is dirty
  - t_{MM} : access time of main memory
  - t_{swap} : time required to swap-in or swap-out a page

- Effective memory access time

$$\mathbf{EAT = (1 - p)\,(t_{MM} + t_{MM}) + p\,[\{(1 - \delta)\,t_{swap} + \delta\,(2\,t_{swap})\} + t_{MM}]}$$

The above expression does not consider the speed-up due to TLB.

Homework:

Modify the expression considering that TLB is also present.

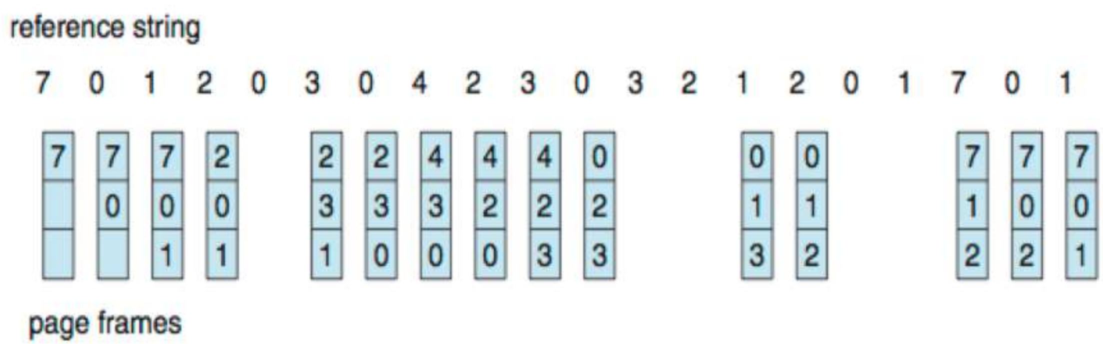
Page Replacement Algorithms: Set up

- **Page-replacement algorithm**
  - Want 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
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

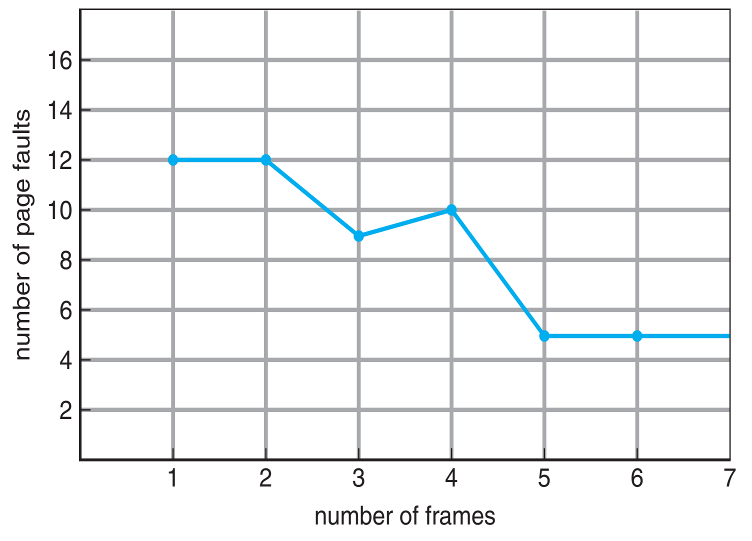


15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

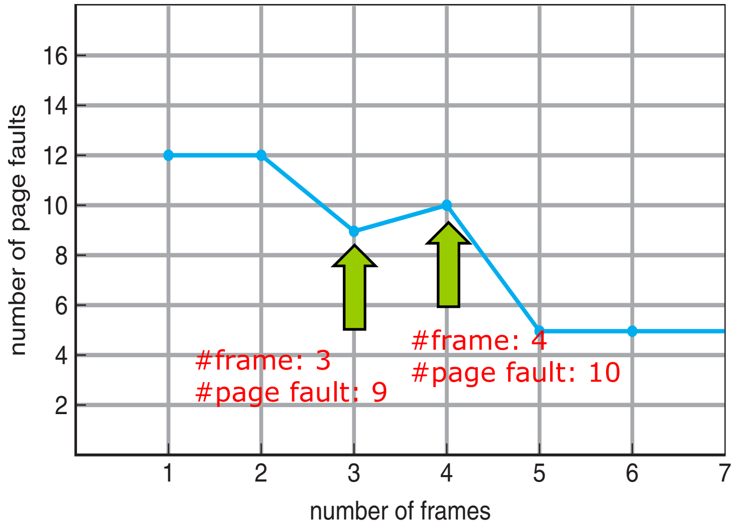
Problem with FIFO: Belady's Anomoly

- Consider reference string: **1,2,3,4,1,2,5,1,2,3,4,**



Problem with FIFO: Belady's Anomoly

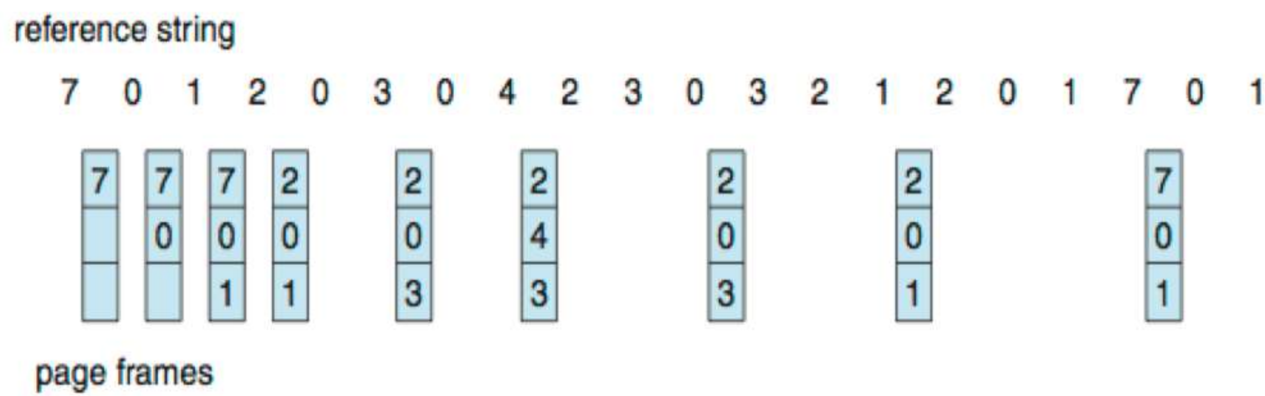
- Consider reference string: **1,2,3,4,1,2,5,1,2,3,4,**



- Adding more frames can cause more page faults!
  - **Belady's Anomaly**

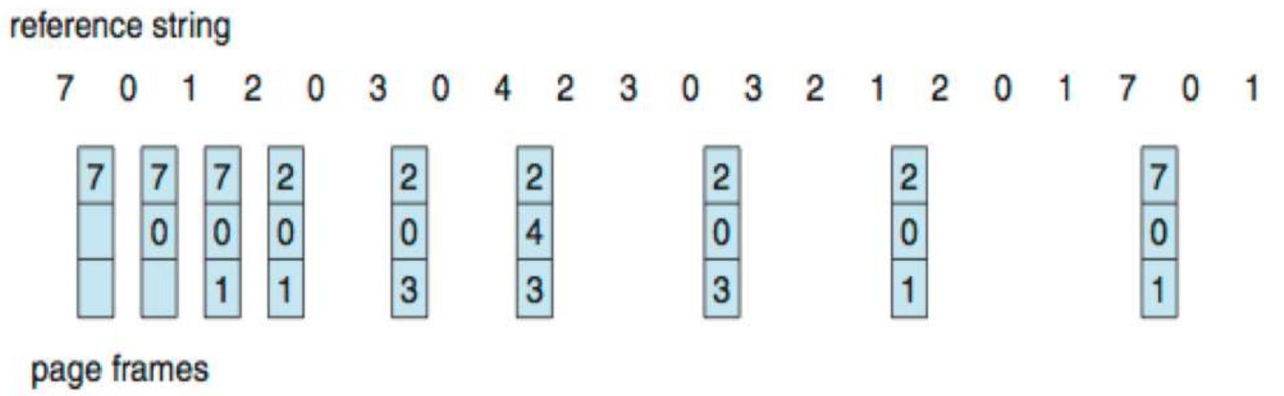
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example



Optimal Algorithm

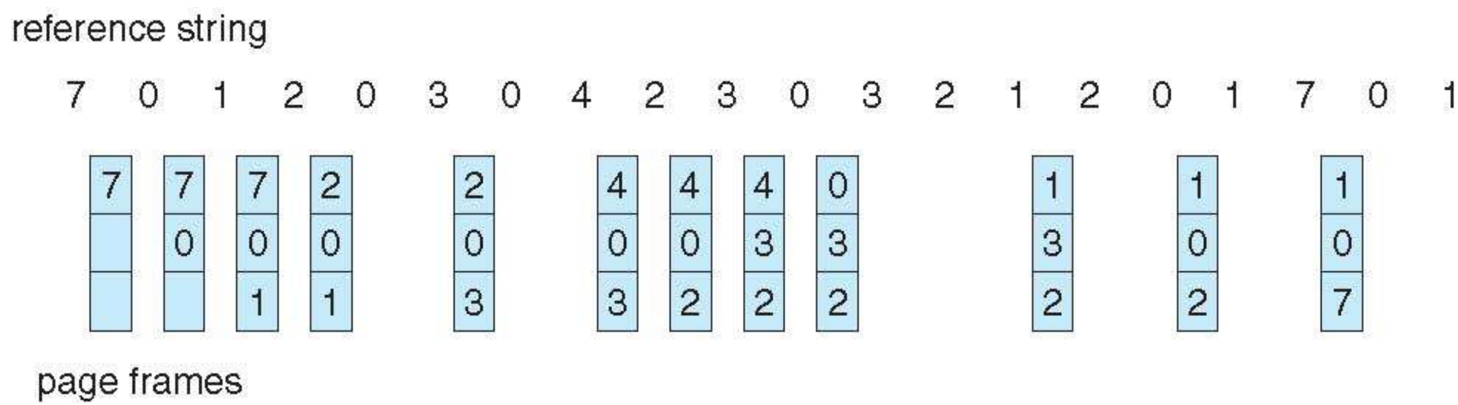
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example



- Problem: You don't know the future memory references
- Still useful: Measuring how well another algorithm performs compare to optimal

Least Recently Used (LRU) Algorithm

- 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

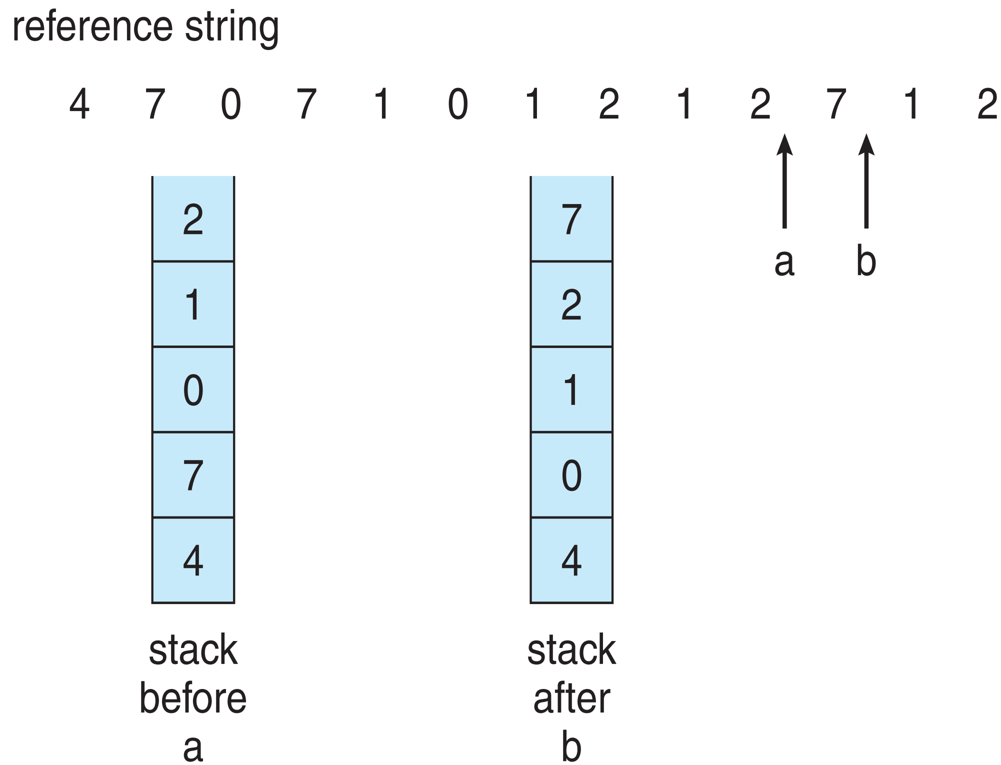


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

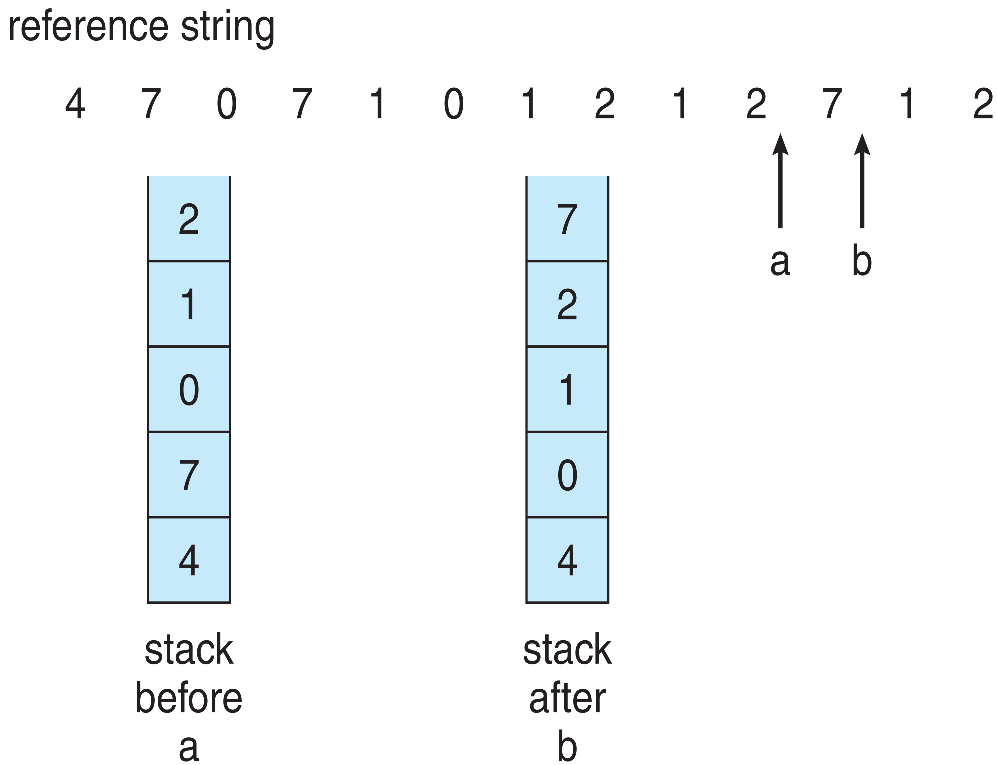
LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed (why?)
  - Bad: each update is expensive
  - Good: No search for replacement (see bottom of stack)
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record Most Recent Page References



Use Of A Stack to Record Most Recent Page References



- **stack algorithms:** the set of pages in a memory with n frames is always a subset of the pages that would be in a memory with n+1 frames

Approximating LRU

- LRU needs special hardware and still slow
- We can approximate it
- Introduce **Reference bit (R)**
  - With each page associate a bit R, initially R = 0
  - When page is read/written do R = 1

- A simple clock algorithm (also called second chance algorithm)
  - Start with *any* page P
    - Step 1: if R = 0, replace page P and break
    - Step 2: if R = 1, recently used, not a good candidate, do R = 0
    - Step 3: P -> (P + 1) % num_pages, repeat from step 1
- An improvement: in Step 1 also check if dirty bit = 0 in first scan. Why?

Approximating LRU

- LRU needs special hardware and still slow
- We can approximate it
- Introduce **Reference bit (R)**
  - With each page associate a bit R, initially R = 0
  - When page is read/written do R = 1

- A simple clock algorithm (also called second chance algorithm)
  - Start with *any* page P
    - Step 1: if R = 0, replace page P and break
    - Step 2: if R = 1, recently used, not a good candidate, do R = 0
    - Step 3: P -> (P + 1) % num_pages, repeat from step 1
- An improvement: in Step 1 also check if dirty bit = 0 in first scan. Why?
  - evict in this order of (ref, dirty) → (0, 0), (0, 1), (1, 0), (1, 1)

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used + largest count has fair shared of usage
- Not common due to complexity

OS challenge

- How would OS use this larger, slower hard disk device to transparently provide user the illusion of a large virtual address space
  - Requirement 1: Incorporate swapping into logical to physical address space mapping process

    Demand paging with
    valid-invalid bit / page fault handler

  - Requirement 2: Determine which pages to swap and when

    When no free frame;
    FIFO, Optimal, LRU

  - Requirement 3: Determine how to allocate free frames to processes (user processes and kernel)

Frame Allocation Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 103 frames (after allocating frames for the OS) and 2 processes, give each process 51 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

– $s_i =$ size of process p_i

– $S = \sum s_i$

– $m =$ total number of frames

– $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

m = 103
s1 = 10
s2 = 127
*a1 = (10/137)*103 = 7*
*a2 = (127/127)*103 = 95*

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

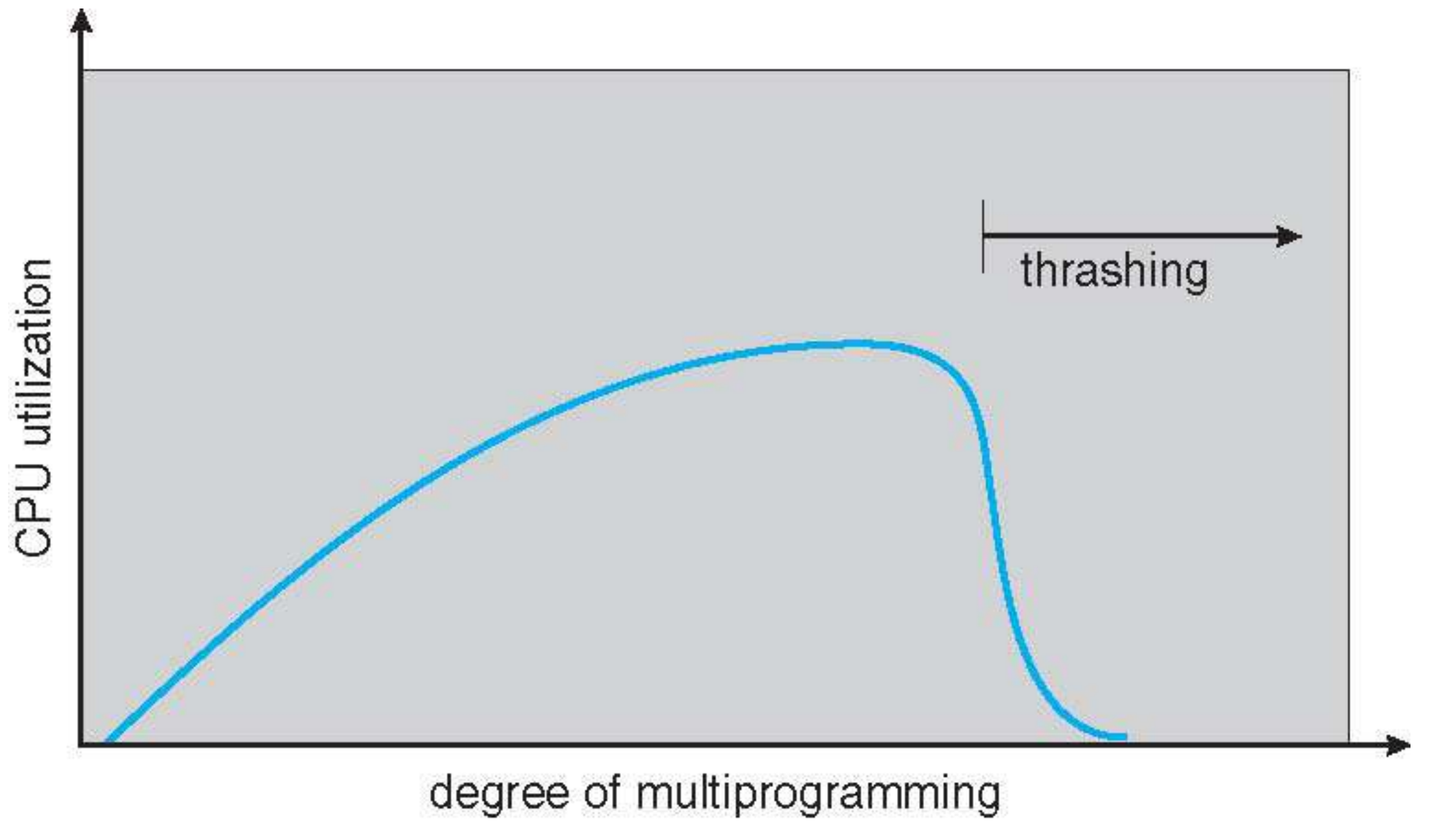
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Process with inadequate frames: Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** $\equiv$ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality (e.g., function) to another
  - Localities may overlap
  - We need to allocate enough frames to a process to accommodate its current locality

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality (e.g., function) to another
  - Localities may overlap
  - We need to allocate enough frames to a process to accommodate its current locality

```
for i = 1 to 128:
    for j = 1 for 128:
            A[i][j]++
```

```
for j = 1 to 128:
    for i = 1 for 128:
            A[i][j]++
```

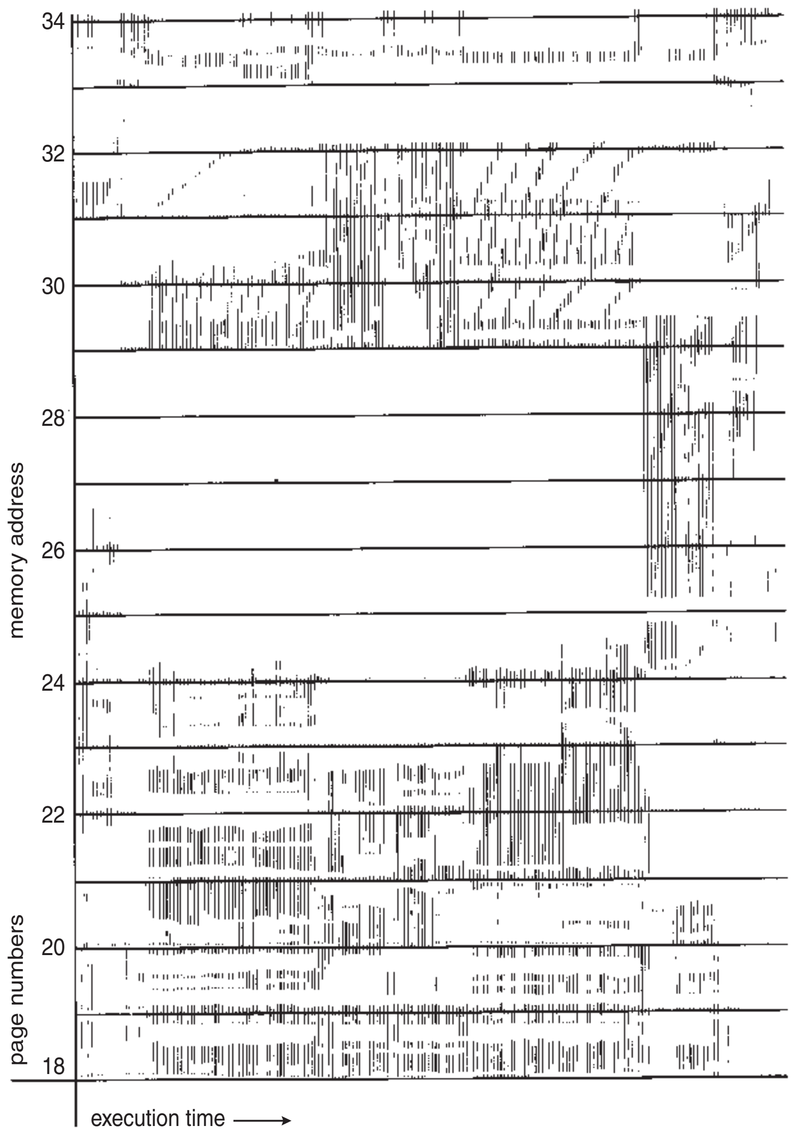
Consider 512 byte pages and 4 bytes array A element

Each row of the matrix A is stored in one page

Demand Paging and Thrashing

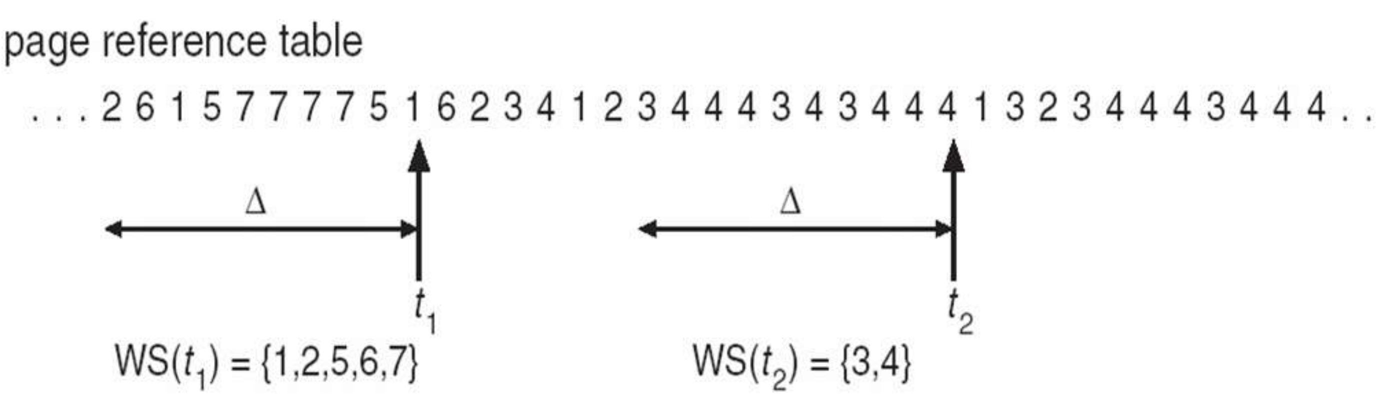
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



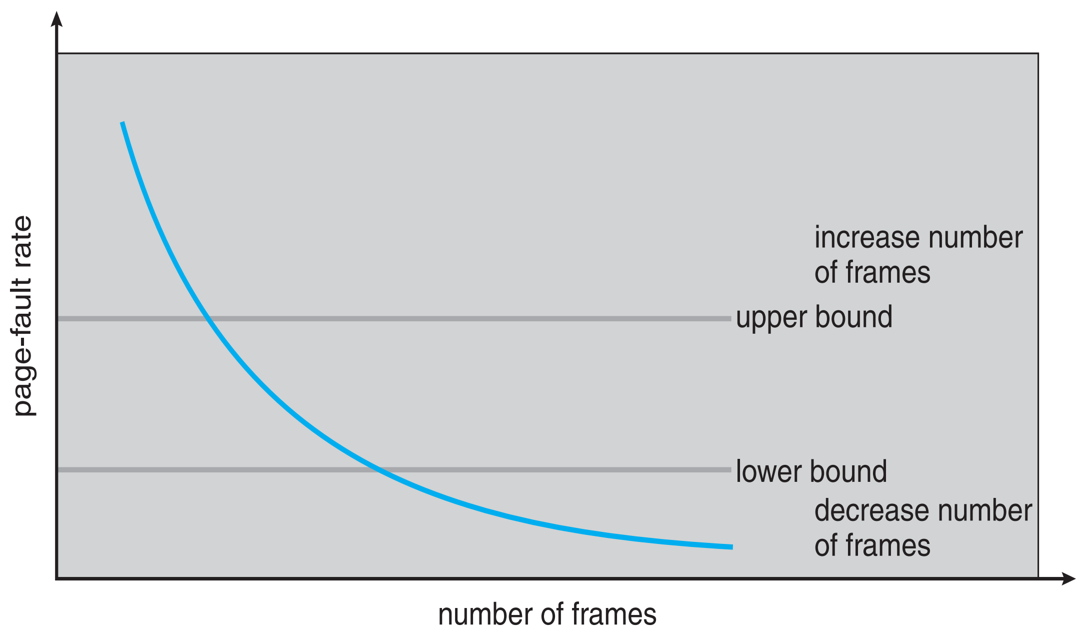
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes



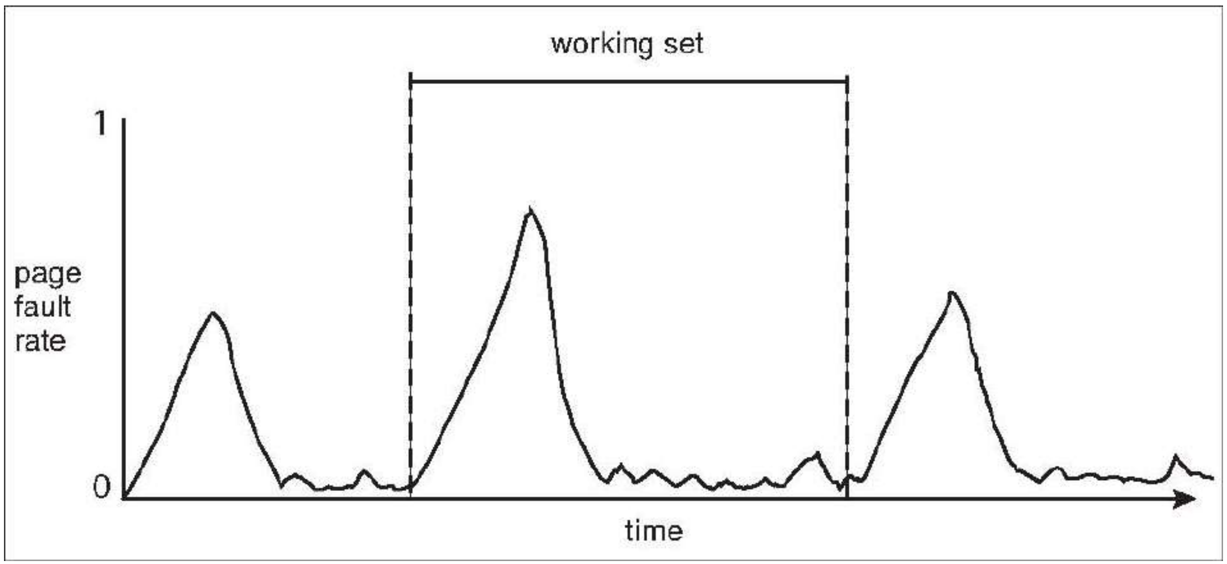
Another approach: Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If PFF rate too low, process loses frame
  - If PFF rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Yet another solution (some Linux versions)

- Use a kernel process called OOM killer (Out-of-Memory killler)
  - Step 1: Pick a process which is using *too much memory*
  - Step 2: Kill the process
- How to know which process is using too much memory?
- More in this blog: https://mukul-mehta.in/til/oom/
- Problems?

Allocating Kernel Memory

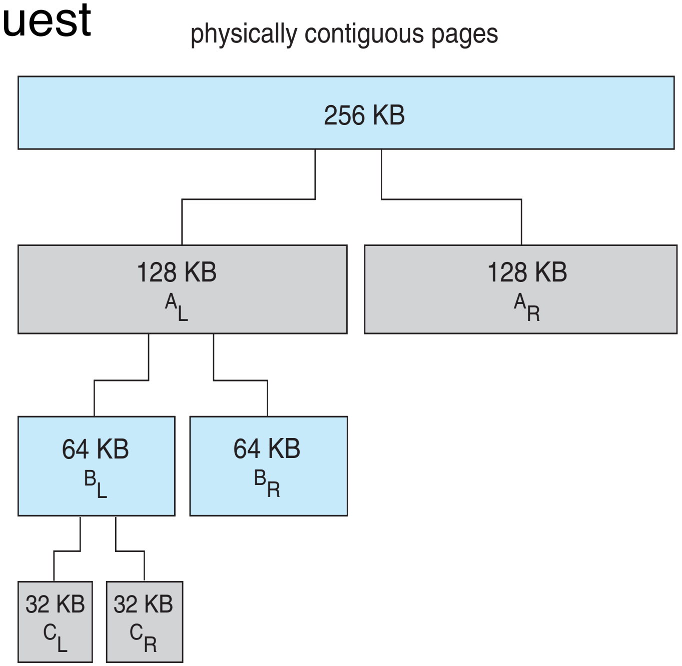
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System Allocator

- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request

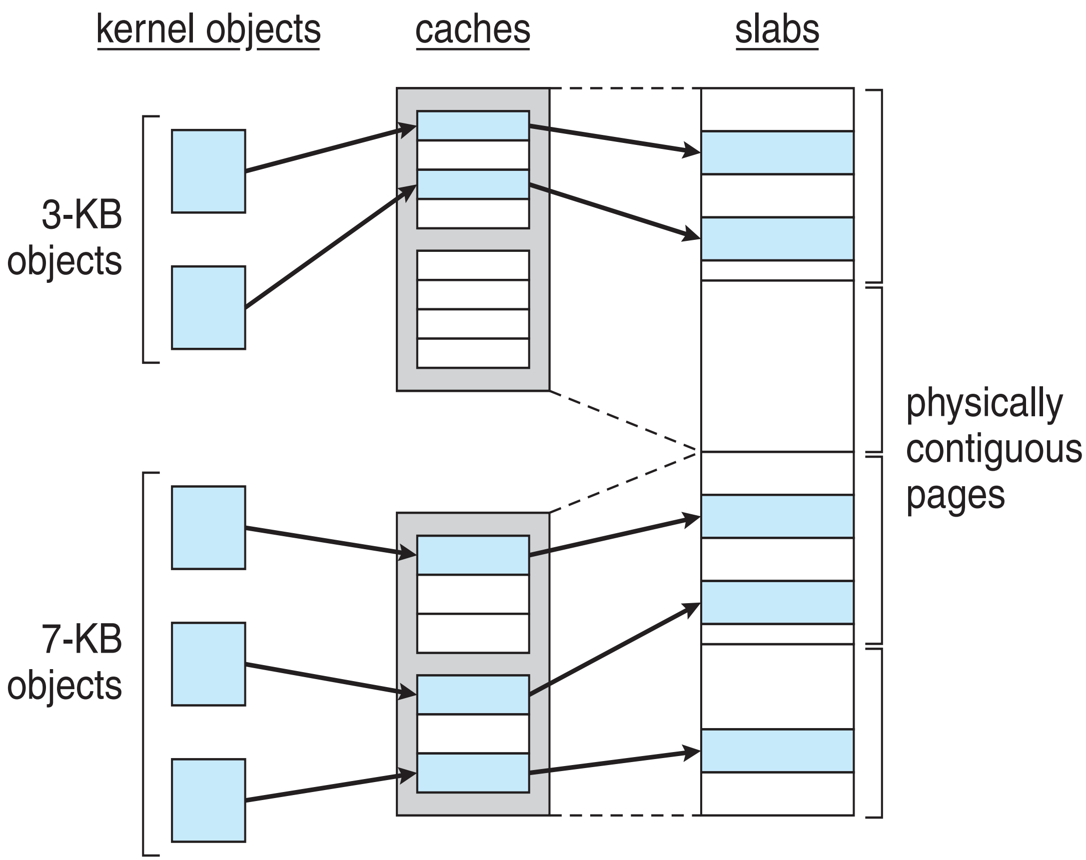


- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator

- Alternate strategy
- **Idea:** kernel data structures are well known and only a few types
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of kernel data structures
- When cache created, filled with objects marked as `free`
- When structures stored, objects marked as `used`
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Other Issues – Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume *s* pages are prepaged and *a* of the pages is used
    - Is cost of $\boldsymbol{s * a}$ save pages faults > or < than the cost of prepaging $\boldsymbol{s * (1- a)}$ unnecessary pages*?*
    - $\boldsymbol{a}$ near zero $\Rightarrow$ prepaging loses

Summary

- Virtual memory = physical memory + swap space
  - Primary procedure: swap in and out pages b/w main memory & disk
  - Problem: Hard disk is much slower than main memory
- How to incorporate swapping in virtual to physical address mapping
  - Demand paging – Only bring in pages when required
  - Need machineries: valid/invalid bit to detect if a page in memory or disk, page fault handler to actually swap in/out pages
- Which pages to swap/replace?
  - FIFO, Optimal, LRU algorithm, Approximating LRU with Reference bit
- How to allocate free frames to process
  - User process: Fixed allocation (equal share to all), Priority allocation
  - Kernel: Buddy system allocator, Slab allocator
- Very common problem: Thrashing (processes only swapping pages)
  - Use locality of reference via working set model (set of pages referenced in last Δ time)