

# Virtual Memory



Operating Systems  
Wenbo Shen

# Review

---

- Logical vs physical address
- Memory allocation
  - Contiguous allocation: first-, best-, worst-fit
  - Fragmentation: external vs internal
- paging: page number + page offset
  - Hierarchical page table, hashed page table, inverted page table
  - 1-level vs 2-level, why save memory, page table walk
- Swapping
- MMU
  - TLB

# Background

---

- Code needs to be in memory to execute, but entire program **rarely** needed or used at the same time
  - **unused code:** error handling code, unusual routines
  - **unused data:** large data structures
- Consider ability to execute **partially-loaded program**
  - program no longer constrained by limits of physical memory
  - programs could be larger than physical memory

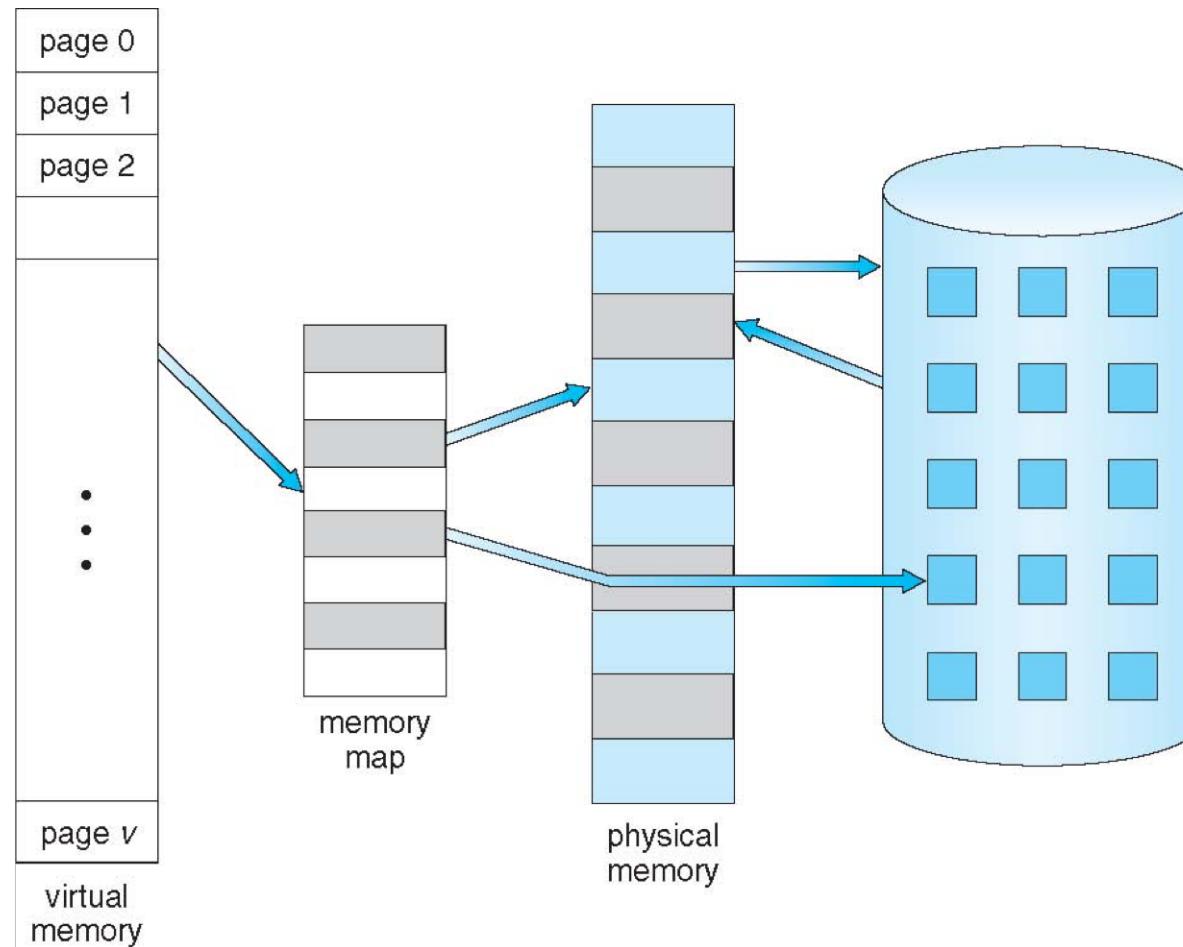
# Background

---

- Virtual memory: separation of **logical memory from physical memory**
  - only part of the program needs to be in memory for execution
    - logical address space can be much larger than physical address space
    - more programs can run concurrently
    - less I/O needed to load or swap processes (part of it)
  - allows memory (e.g., shared library) to be shared by several processes: better IPC performance
  - allows for more efficient process forking (**copy-on-write**)
- Virtual memory can be implemented via:
  - **Paging**

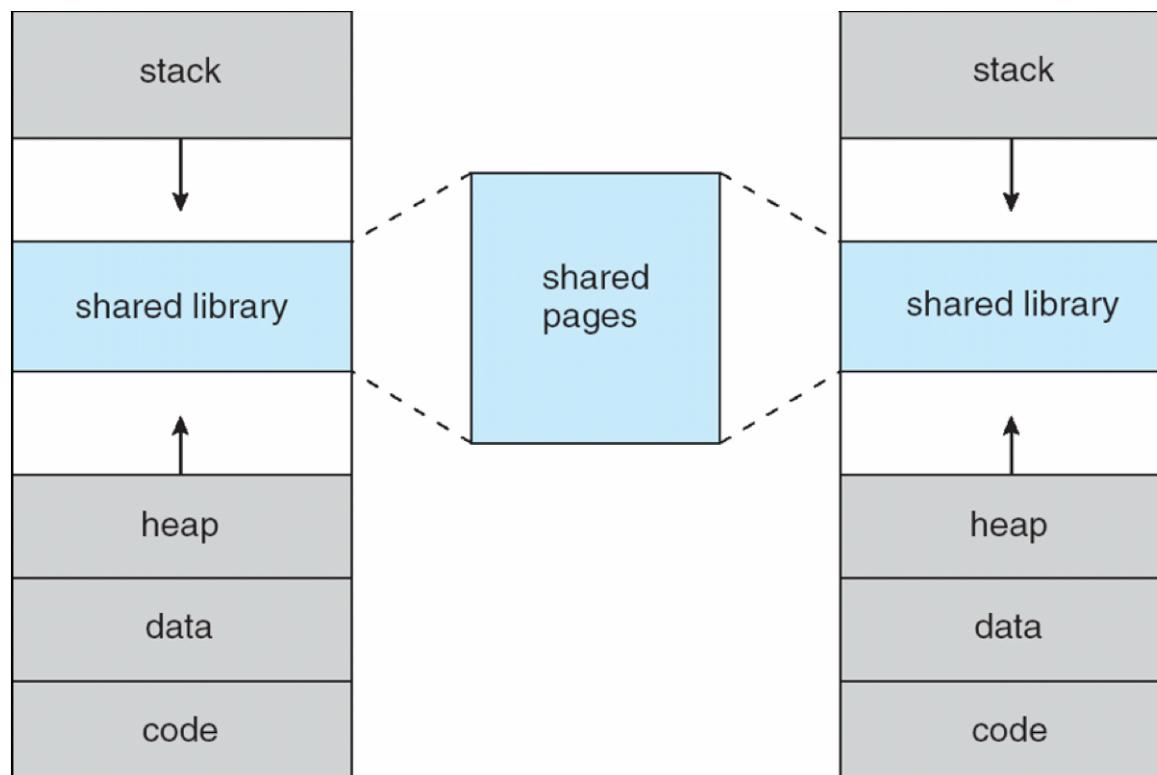
# Virtual Memory Larger Than Physical Memory

- Virtual Memory Larger Than Physical Memory



# Shared Library Using Virtual Memory

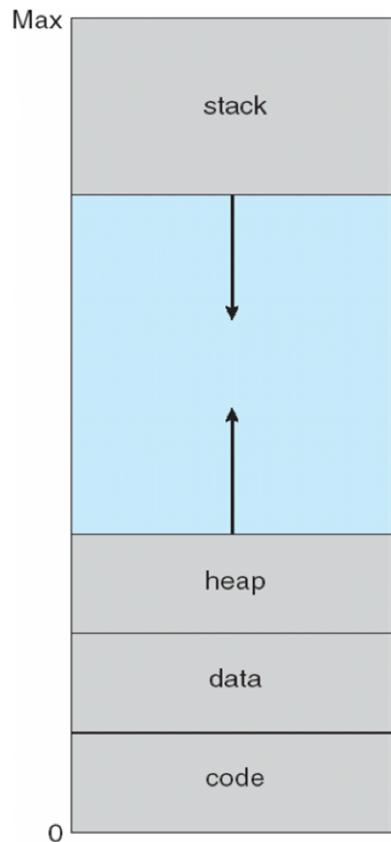
---



# Virtual-address Space

---

- 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
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Pages can be shared during fork(), speeding process creation: COW



# Demand Paging

---

- Demand paging brings a **page** into memory only when it is **demanded**
  - demand means access (read/write)
  - if page is invalid (error) ➡ abort the operation
  - if page is valid but not in memory ➡ bring it to memory
    - Memory here means **physical** memory
    - This is called **page fault**
    - via swapping for swapped pages
    - via mapping for new page
    - no unnecessary I/O, less memory needed, slower response, more apps

# Valid-Invalid Bit

---

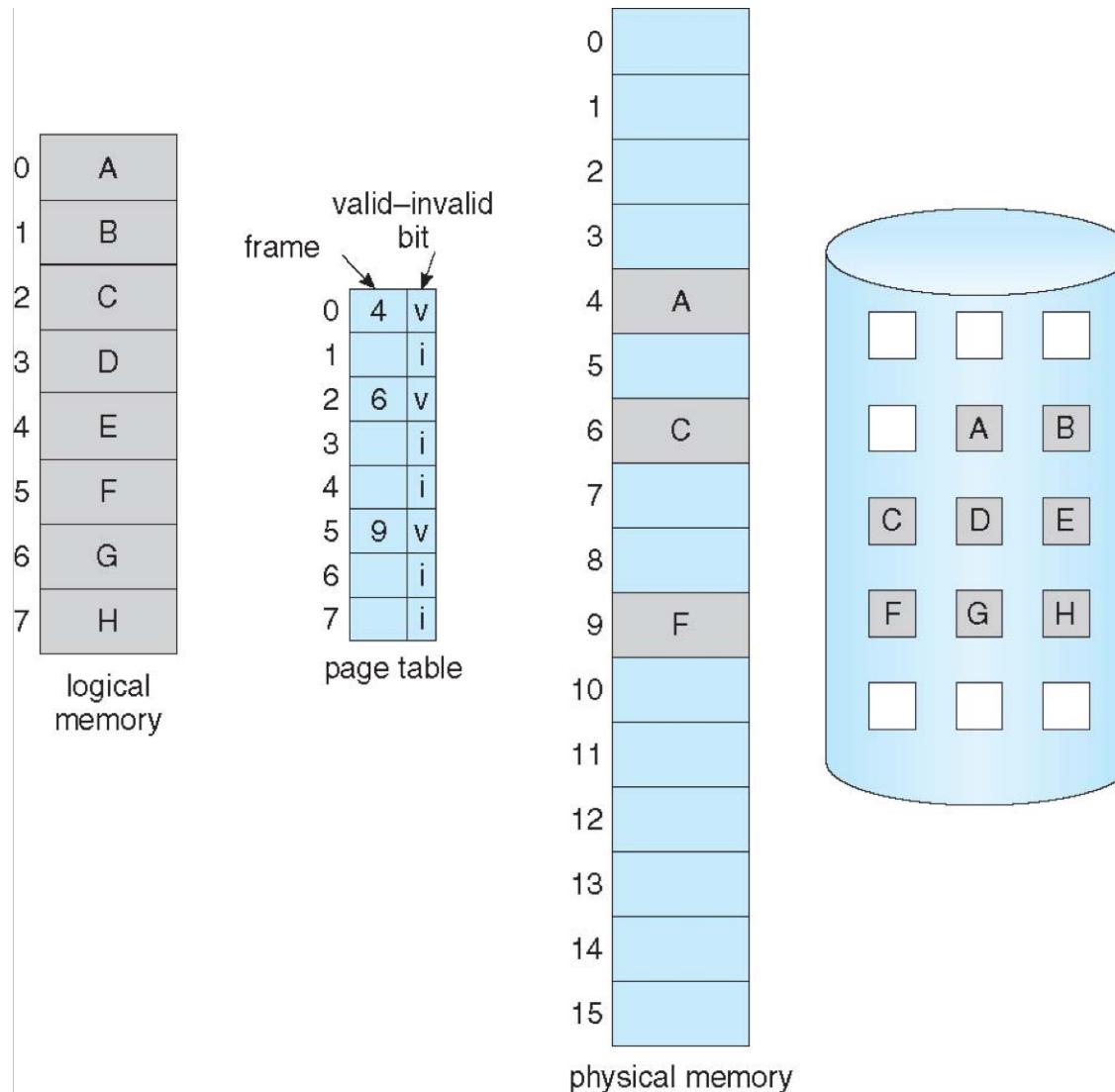
- How does MMU know the physical frame is not mapped?
- Each page table entry has a valid-invalid (present) bit
  - V → in memory (memory is resident), I → not-in-memory
  - initially, valid-invalid bit is set to *i* on all entries
  - during address translation, if the entry is invalid, it will trigger a **page fault**
- Example of a page table snapshot:

| Frame # | v/i bit  |
|---------|----------|
|         | <b>v</b> |
|         | <b>v</b> |
|         | <b>v</b> |
|         | <b>v</b> |
|         | <b>i</b> |
| ....    |          |
|         | <b>i</b> |
|         | <b>i</b> |

page table  
9

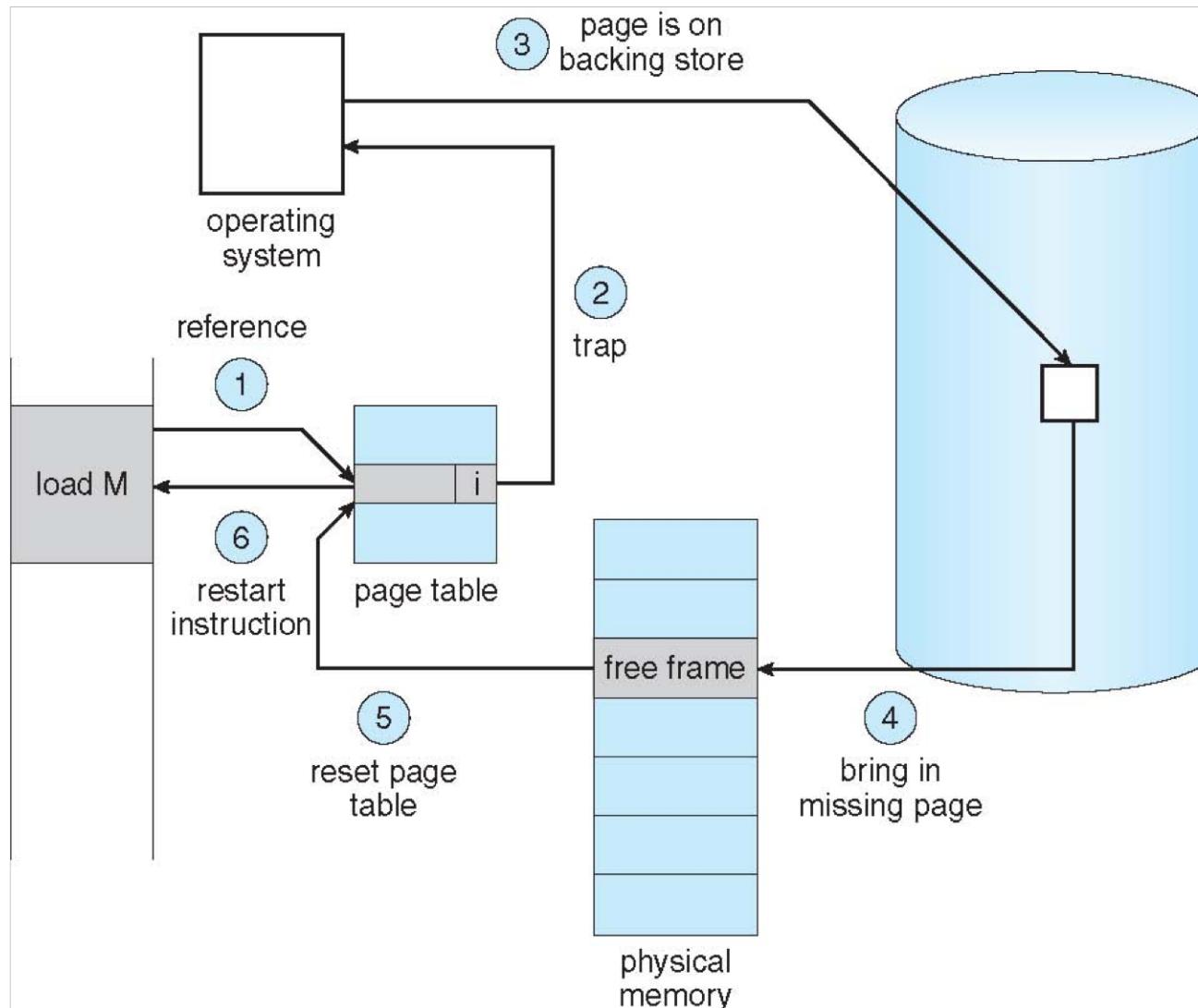
# Page Table (Mem Pages Are Not in Memory)

- Executable file



# Page Fault Handling

- Page fault for code



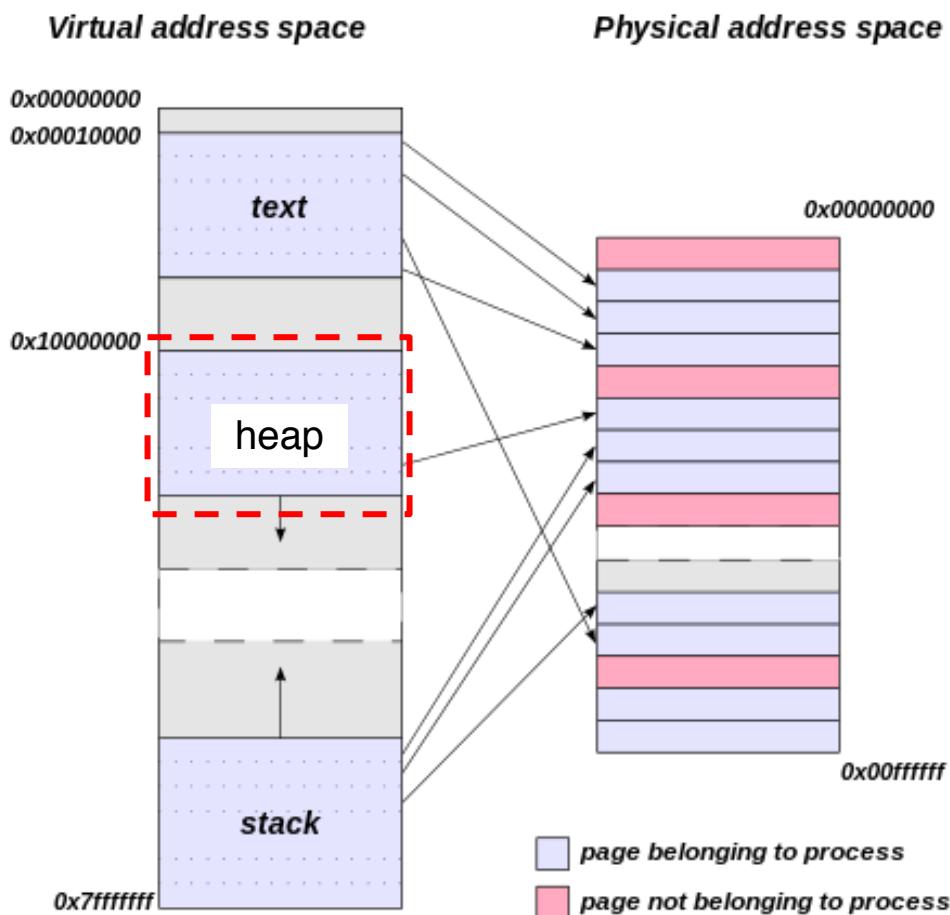
# Page Fault

---

- First reference to a non-present page will trap to kernel:  
**page fault**
- Operating system looks at memory mapping to decide:
  - **invalid reference** ➡ deliver an exception to the process
  - **valid but not in memory** ➡ swap in
    - get an empty physical frame
    - swap page into frame via disk operation
    - set page table entry to indicate the page is now in memory
    - restart the instruction that caused the page fault

# Demand Paging

- Page fault on data

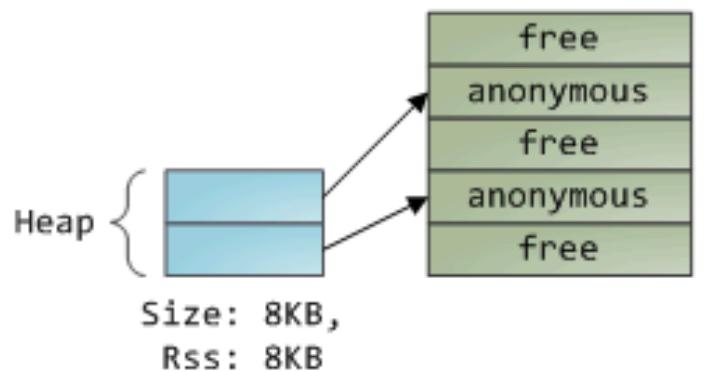


# Demand Paging

- How The Kernel Manages Your Memory

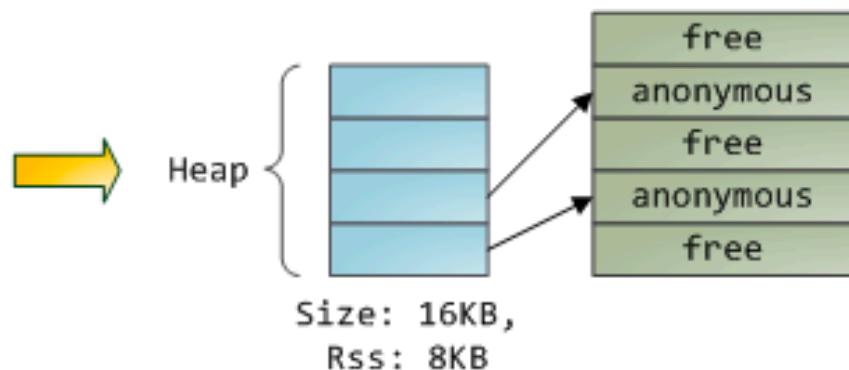
- <https://manybutfinite.com/post/how-the-kernel-manages-your-memory/>

1. Program calls brk() to grow its heap

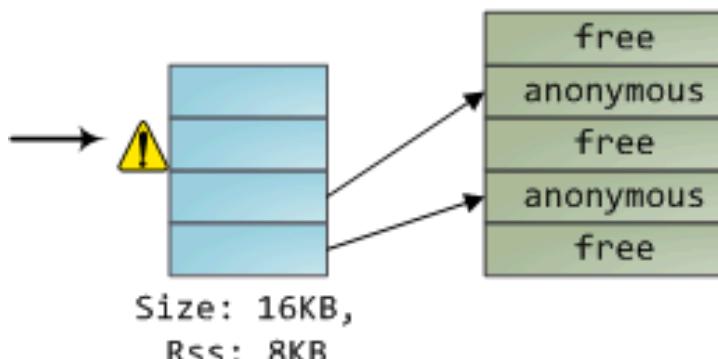


2. brk() enlarges heap VMA.

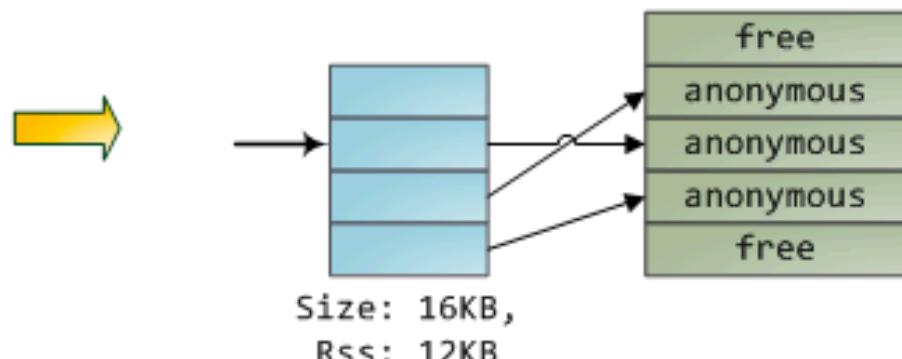
New pages are **not** mapped onto physical memory.



3. Program tries to access new memory.  
Processor page faults.



4. Kernel assigns page frame to process,  
creates PTE, resumes execution. Program is  
unaware anything happened.



# Demand Paging

---

- **Lazy swapper:** never swaps a page in memory unless it will be needed
  - the swapper that deals with pages is also called a pager
- **Pre-Paging:** pre-page all or some of pages a process will need, before they are referenced
  - it can reduce the number of page faults during execution
  - if pre-paged pages are unused, I/O and memory was wasted
    - although it reduces page faults, total I/O# likely is higher

# Demand Paging

---

- Extreme case: start process with no pages in memory (aka. **pure demand paging**)
  - OS sets instruction pointer to first instruction of process
    - invalid page  $\Rightarrow$  page fault
  - every page is paged in on first access
    - **program locality** reduces the overhead
  - an instruction could access multiple pages  $\Rightarrow$  multiple page faults
    - e.g., instruction, data, and page table entries for them
- Demand paging needs hardware support
  - page table entries with **valid / invalid bit**
  - **backing storage** (usually disks)
  - **instruction restart**

# Free-Frame List

---

- 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.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

# 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:
  - 5.1 Wait in a queue for this device until the read request is serviced
  - 5.2 Wait for the device seek and/or latency time
  - 5.3 Begin the transfer of the page to a free frame

# Stages in 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

# Demand Paging: EAT

---

- 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):  
$$(1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{instruction restart overhead})$$

# Demand Paging Example

---

- Assume memory access time: 200 nanoseconds, average page-fault service time: 8 milliseconds
  - $EAT = (1 - p) \times 200 + p \times (8 \text{ milliseconds})$   
 $= (1 - p) \times 200 + p \times 8,000,000$   
 $= 200 + p \times 7,999,800$
  - if one out of 1,000 causes a page fault, then  $EAT = 8.2$  microseconds
    - a slowdown by a factor of 4100%
      - $8.2 \text{ us} / 0.2 \text{ us} = 41$
    - if want < 10 percent, less than one page fault in every 400,000 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 Paging Optimizations

---

- Demand page in from program binary on disk, but **discard** rather than paging out when freeing frame (and reload from disk next time)
  - 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)

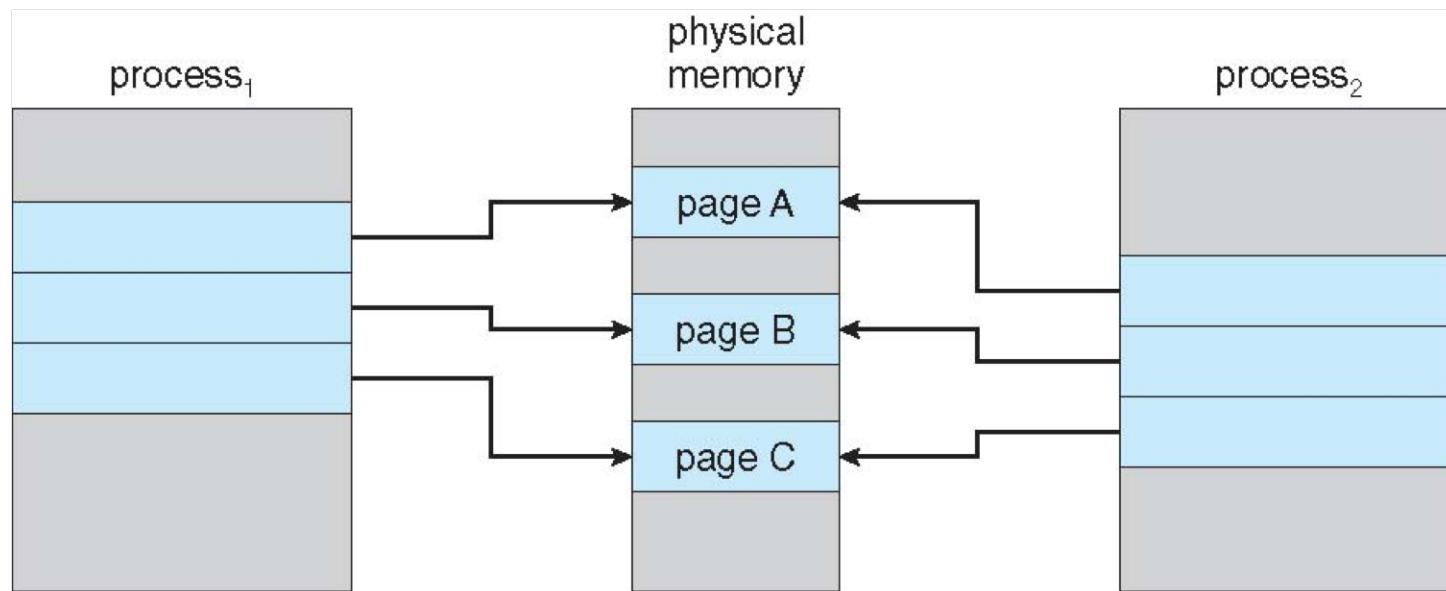
# Copy-on-Write

---

- **Copy-on-write (COW)** allows parent and child processes to initially share the same pages in memory
  - the page is shared as long as no process modifies it
  - if either process modifies a shared page, only then is the page copied
- **COW allows more efficient process creation**
  - no need to copy the parent memory during fork
  - only changed memory will be copied later
- **vfork syscall optimizes the case that child calls exec immediately after fork**
  - parent is suspend until child exits or calls exec
  - child shares the parent resource, including the heap and the stack
    - child cannot return from the function or call exit
  - **vfork could be fragile, it is invented when COW has not been implemented**

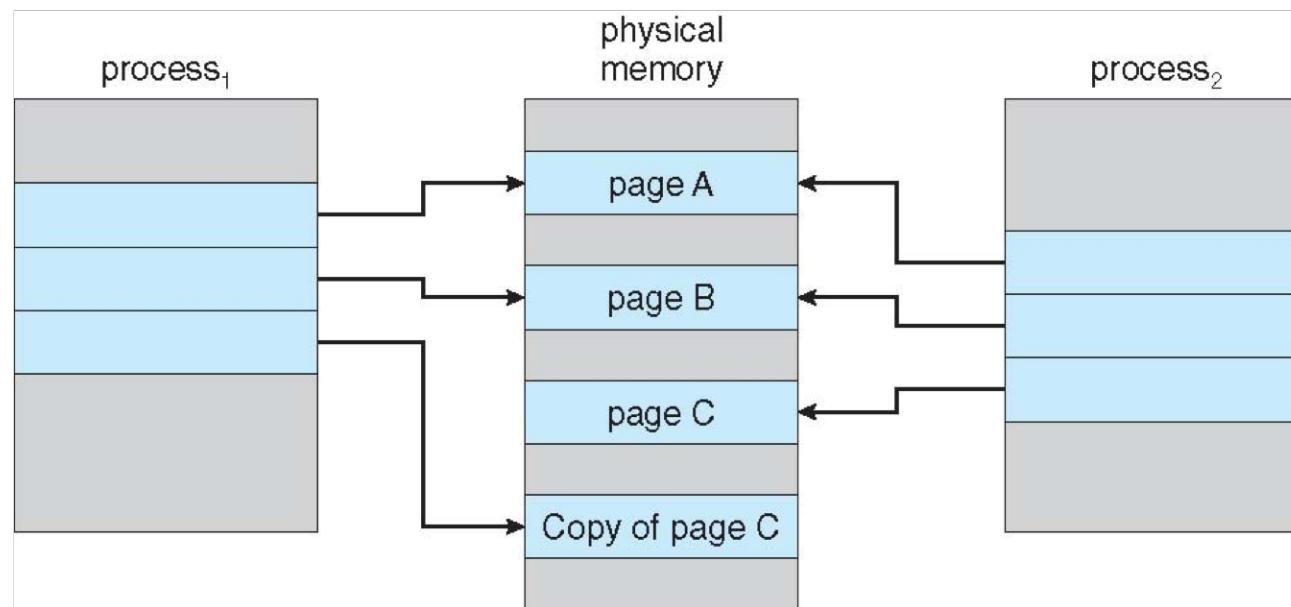
# Before Process 1 Modifies Page C

---



# After Process 1 Modifies Page C

---



# What Happens if There is no Free Frame?

---

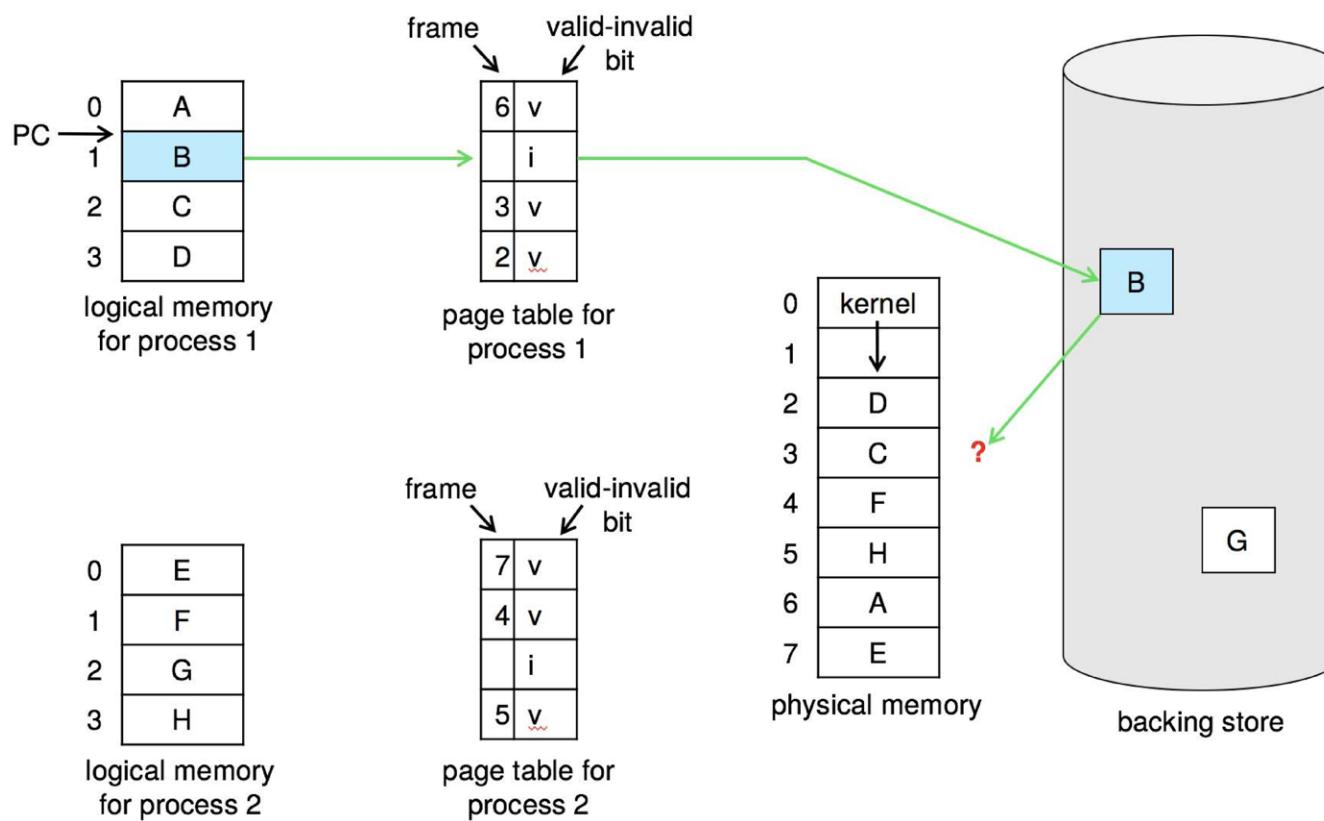
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** - find some page in memory, but not really in use, page it out
  - Algorithm - terminate? swap out? replace the page?
  - Performance - want an algorithm which will result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

# Page Replacement

---

- Memory is an important resource, system may run out of memory
- To prevent out-of-memory, swap out some pages
  - page replacement usually is a part of the page fault handler
  - policies to select victim page require careful design
    - need to reduce overhead and avoid **thrashing**
  - use modified (dirty) bit to reduce number of pages to swap out
    - only modified pages are written to disk
  - select some processes to kill (last resort)
- Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory

# Need For Page Replacement

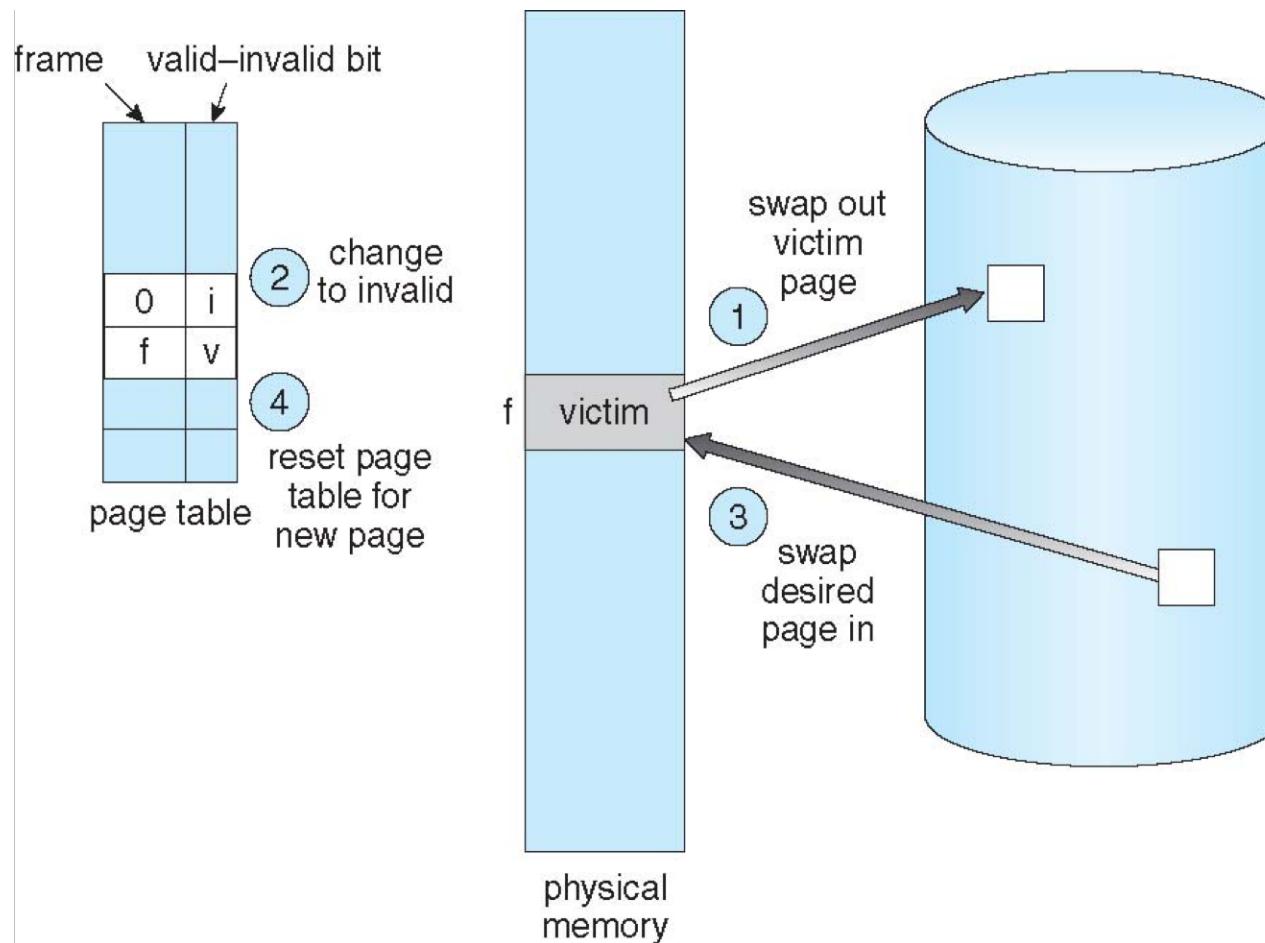


# Page Fault Handler (with Page Replacement)

---

- To **page in** a page:
  - find the location of the desired page on disk
  - find a free frame:
    - if there is a free frame, use it
    - if there is none, use a page replacement policy to pick a victim frame, write victim frame to disk if dirty
  - bring the desired page into the free frame; update the page tables
  - restart the instruction that caused the trap
- Note now potentially **2 page I/O** for **one page fault** ➡ increase EAT

# Page Replacement



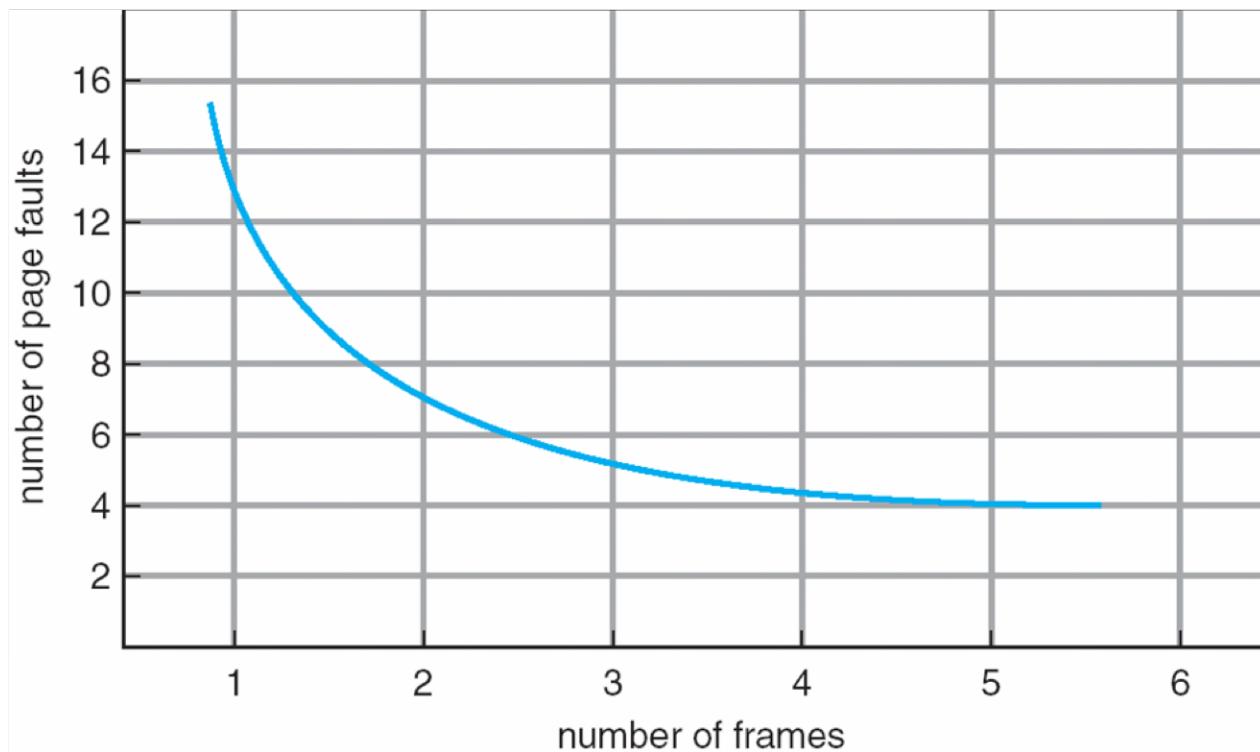
# Page Replacement Algorithms

---

- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - FIFO, optimal, LRU, LFU, MFU...
- To evaluate a page replacement algorithm:
  - run it on a particular string of memory references (reference string)
    - string is just page numbers, not full addresses
    - compute the number of page faults on that string
      - repeated access to the same page does not cause a page fault
    - in all our examples, the reference string is  
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

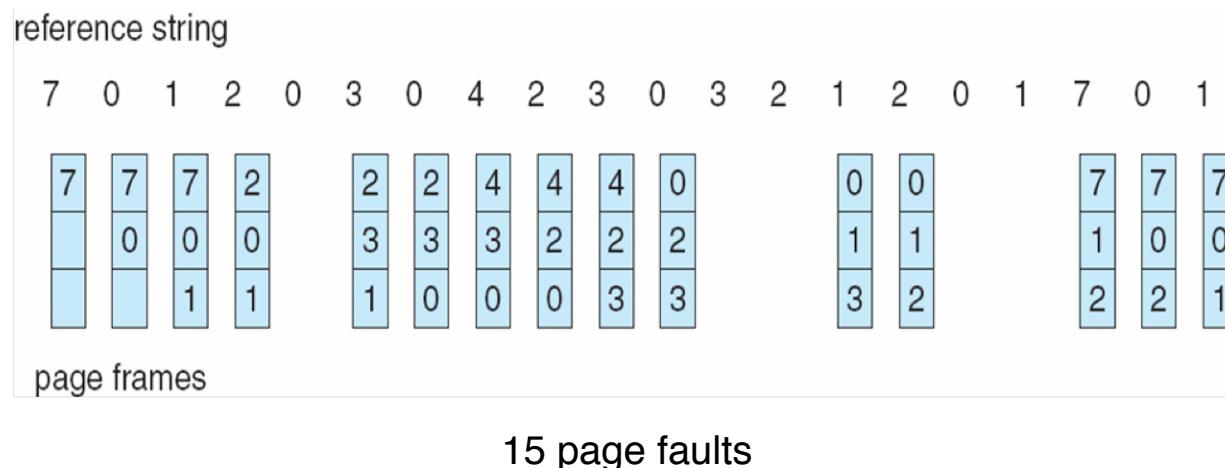
# Page Faults v.s. Number of Frames

---



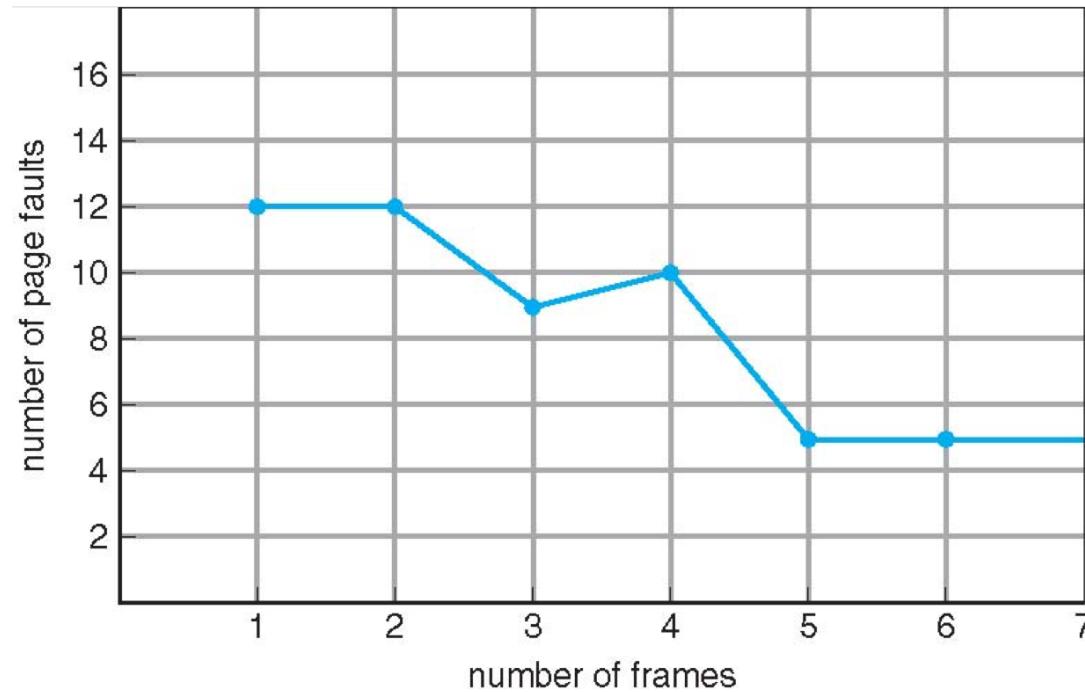
# First-In-First-Out (FIFO)

- **FIFO**: replace the first page loaded
  - similar to sliding a window of n in the reference string
  - our reference string will cause 15 page faults with 3 frames
  - how about reference string of 1,2,3,4,1,2,5,1,2,3,4,5 /w 3 or 4 frames?
- For FIFO, adding **more frames** can cause **more page faults!**
  - **Belady's Anomaly**



# FIFO Illustrating Belady's Anomaly

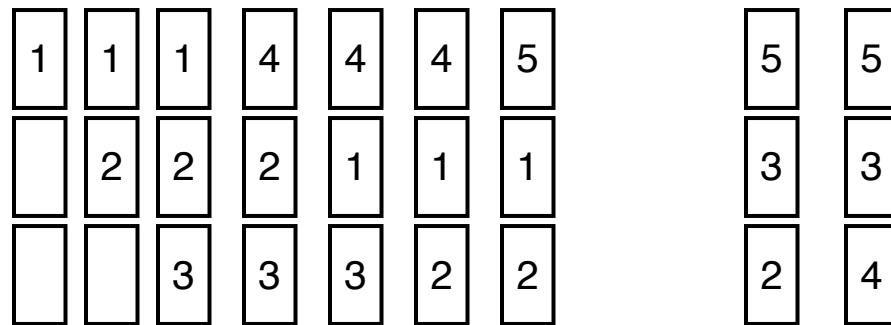
---



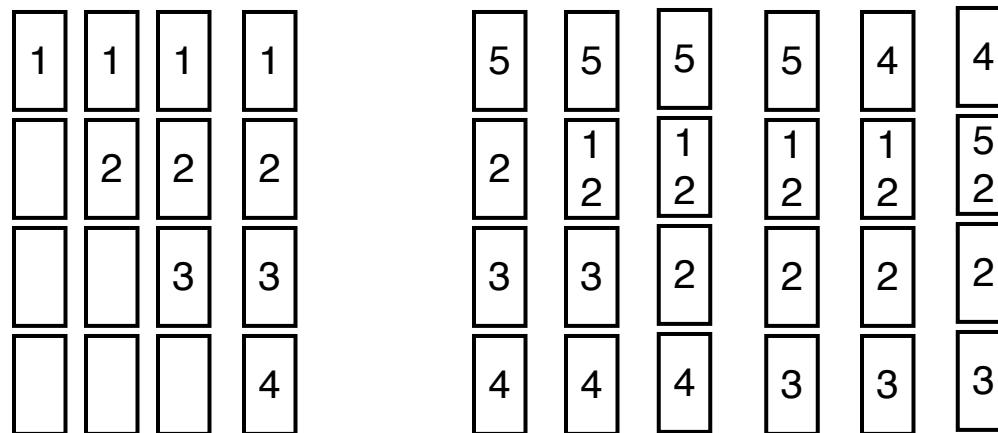
1 2 3 4 1 2 5 1 2 3 4 5

# Belady's Anomaly

1 2 3 4 1 2 5 1 2 3 4 5



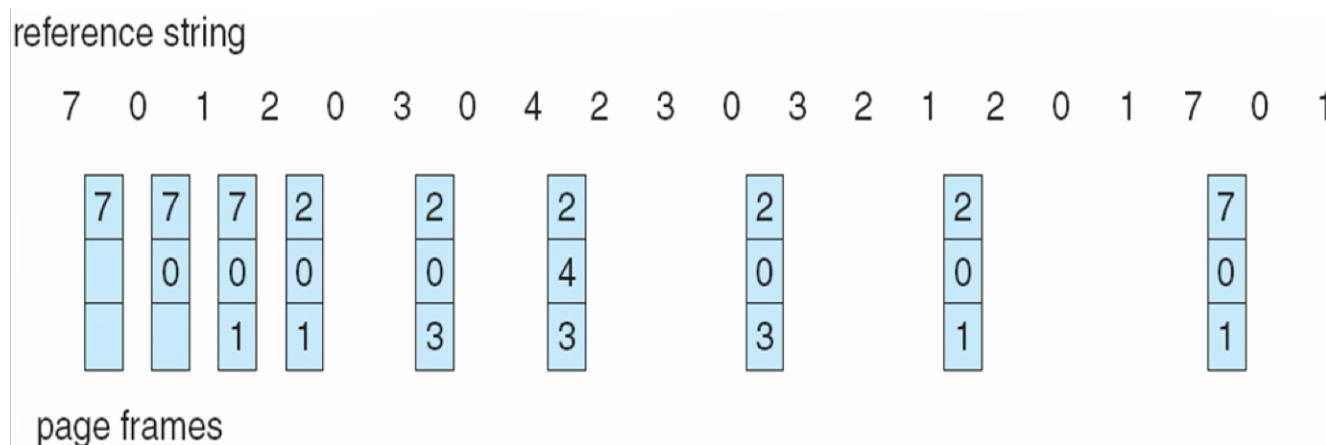
## 9 page faults



# 10 page faults!

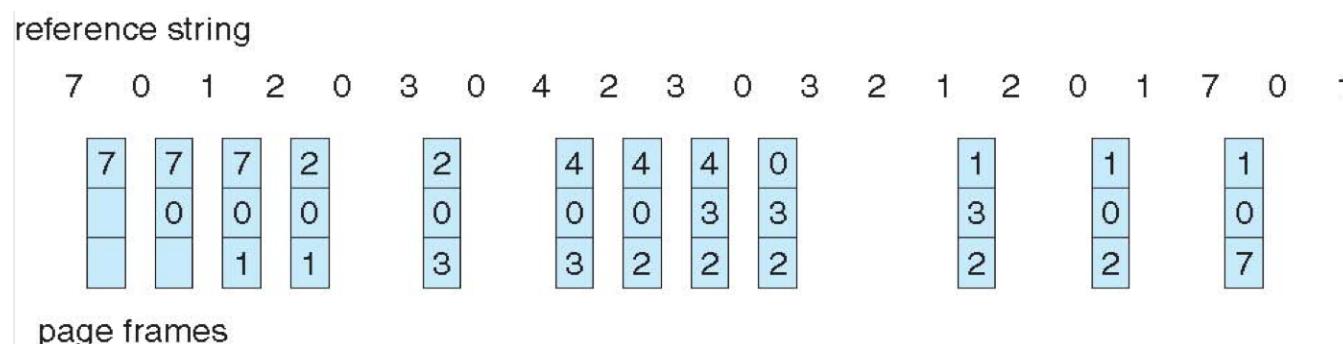
# Optimal Algorithm

- **Optimal** : replace page that will not be used for the longest time
  - 9 page fault is optimal for the example on the next slide
- How do you know which page will not be used for the longest time?
  - can't read the future
  - used for measuring how well your algorithm performs



# Least Recently Used (LRU)

- LRU replaces pages that have not been used for the longest time
  - associate time of last use with each page, select pages w/ oldest timestamp
  - generally good algorithm and frequently used
  - 12 faults for our example, better than FIFO but worse than OPT
- LRU and OPT do **NOT** have Belady's Anomaly
- How to implement LRU?
  - **counter-based**
  - **stack-based**

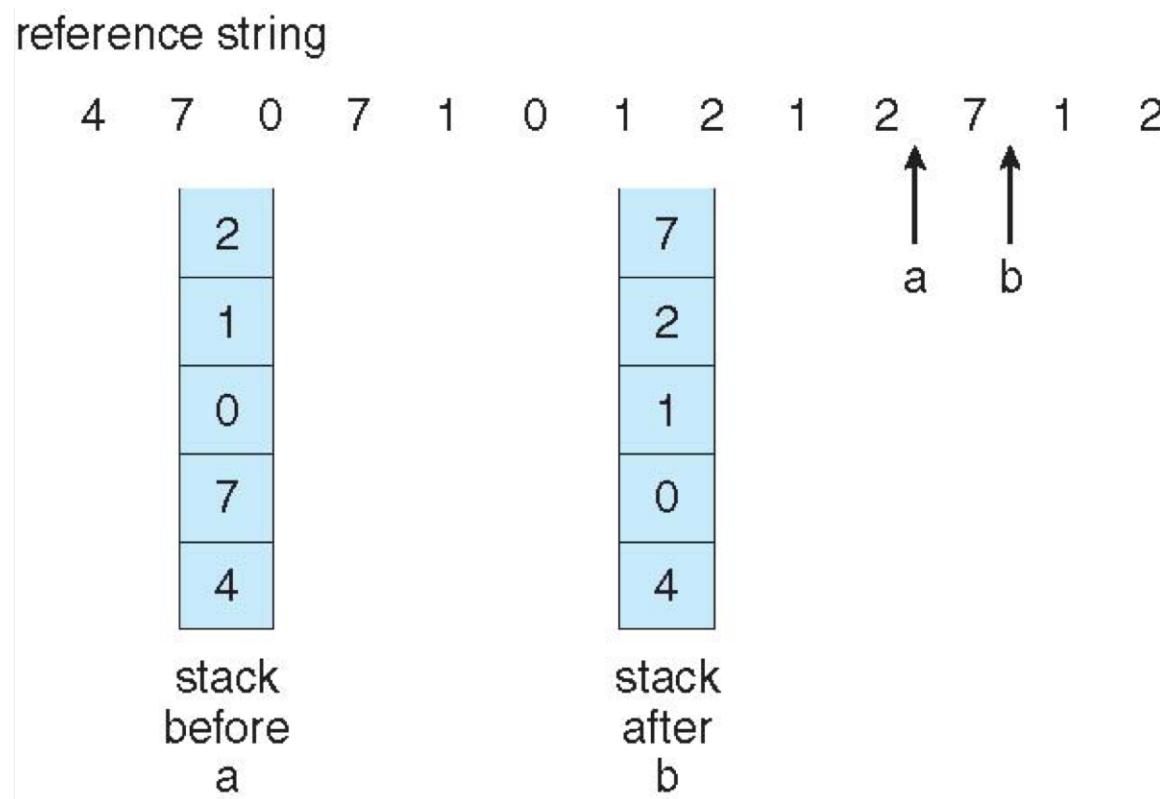


# LRU Implementation

---

- **Counter-based implementation**
  - every page table entry has a counter
  - every time page is referenced, copy the **clock** into the counter
  - when a page needs to be replaced, search for page with smallest counter
    - min-heap can be used
- **Stack-based implementation**
  - keep a stack of page numbers (in double linked list)
  - when a page is referenced, move it to the top of the stack
  - each update is more expensive, but no need to search for replacement

# Stack-based LRU



# LRU Approximation Implementation

---

- Counter-based and stack-based LRU have high performance overhead
- Hardware provides a **reference bit**
- LRU approximation with a **reference bit**
  - associate with each page a reference bit, initially set to 0
  - when page is referenced, set the bit to 1 (done by the hardware)
  - replace any page with reference bit = 0 (if one exists)
    - We do not know the **order**, however

# Additional-Reference-Bits Algorithm

---

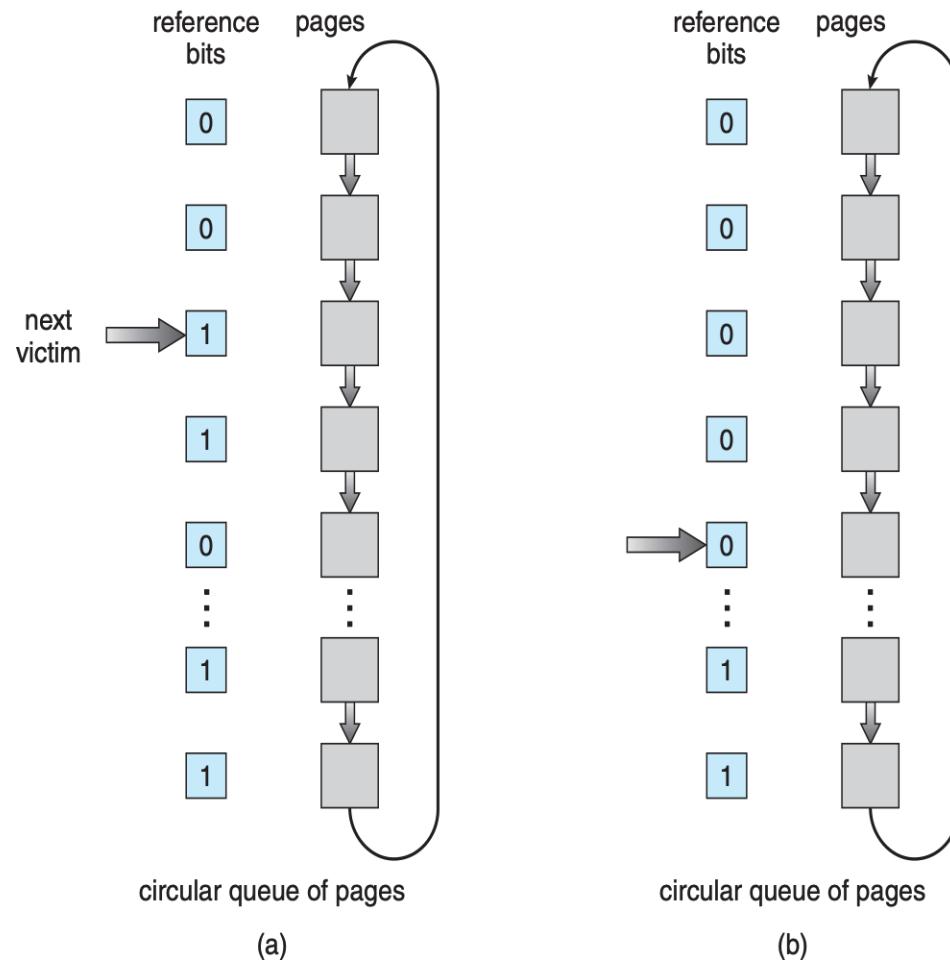
- Reordering the bits at regular intervals
  - Suppose we have 8-bits byte for each page
  - During a time interval (100ms), sets the high bit if used and shifts bit rights by 1 bit, and then discards the low-order bits
  - 00000000 => has not been used in 8 time intervals
  - 11111111 => has been used in all time intervals
  - 11000100 vs 01110111 : which one is used more recently?

# LRU Implementation

---

- **Second-chance** algorithm
  - Generally FIFO, plus hardware-provided **reference bit**
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

# Second-chance (clock) Page-replacement Algorithm



# Enhanced Second-Chance Algorithm

---

- Improve algorithm by using **reference bit** and **modify bit** (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used nor modified - best page to replace
  - (0, 1) not recently used but modified - not quite as good, must write out before replacement
  - (1, 0) recently used but clean - probably will be used again soon
  - (1, 1) recently used and modified - probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

# Counting-based Page Replacement

---

- Keep the number of references made to each page
- **LFU** replaces page with the smallest counter
  - A page is heavily used during process initialization and then never used
- **MFU** replaces page with the largest counter
  - based on the argument that page with the smallest count was probably just brought in and has yet to be used
- LFU and MFU are not common

# Page-Buffering Algorithms

---

- Keep a pool of free frames, always
  - Then frame available when needed, no need to find at fault time
  - Read page into free frames without waiting for victims to write out
    - Restart as soon as possible
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty: this page can be replaced without writing pages to backing store
- Possibly, keep free frame contents intact and note what is in them - a kind of cache
  - If referenced again before reused, no need to load contents again from disk
    - cache hit

# Applications and Page Replacement

---

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge - i.e. databases
- Memory intensive applications can cause **double buffering** - a waste of memory
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk mode**
  - Bypasses buffering, locking, etc

# Allocation of Frames

---

- Each process needs **minimum number** of frames -according to instructions semantics
- Example: IBM 370 - **6 pages** to handle **SS MOVE** instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- **Maximum** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

# Fixed Allocation

---

- Equal allocation - For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 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  $p_i = \frac{s_i}{S} \times m$

$$m = 62$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

# Global vs. Local Allocation

---

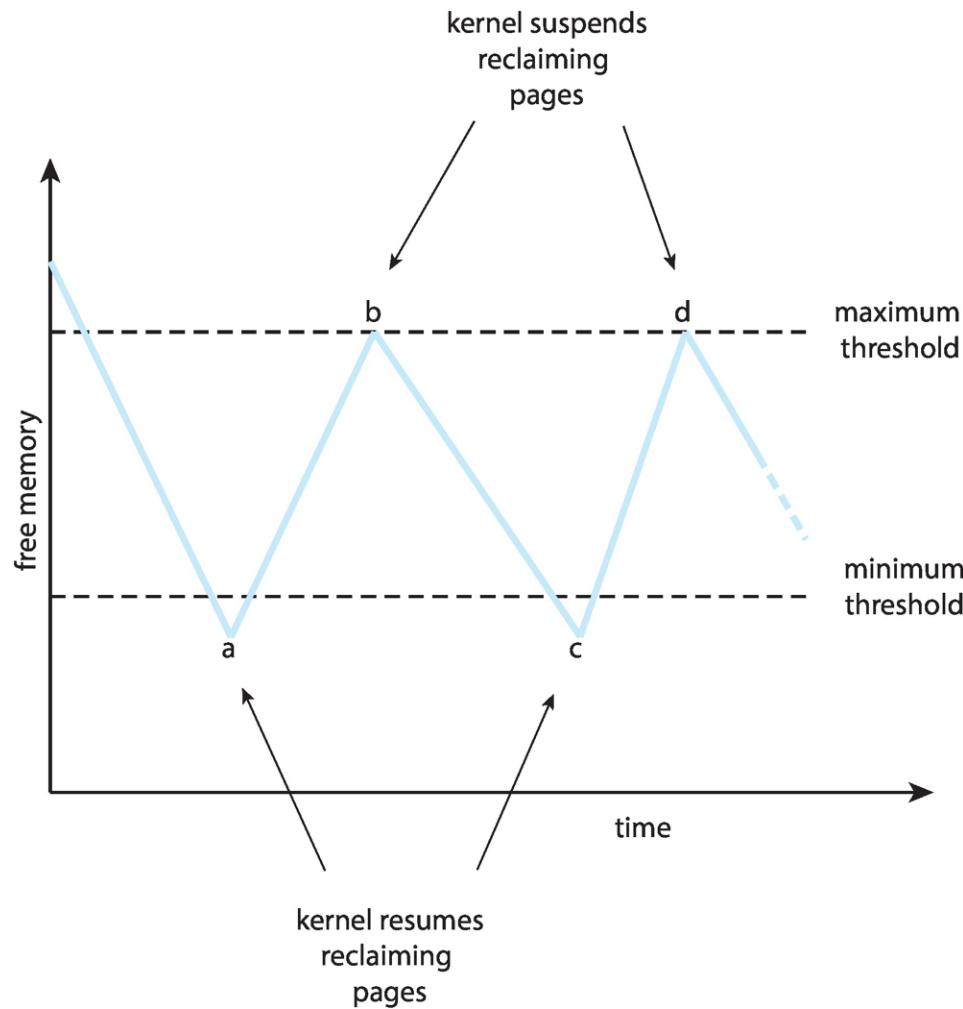
- **Global replacement** - process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly - depends on others
  - But greater throughput so more common
- **Local replacement** - each process selects from only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

# Reclaiming Pages

---

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure **there is always sufficient** free memory to satisfy new requests.

# Reclaiming Pages Example



## What happens if memory is below the minimum threshold

---

- Reclaim pages aggressively
  - Kill some processes
  - OOM score

# Major and minor page faults

---

- Major: page is referenced but not in memory
- Minor: mapping does not exist, but the page is in memory
  - Shared library
  - Reclaimed and not freed yet

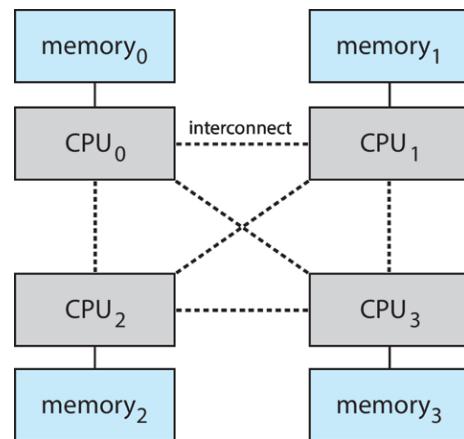
```
wenbo@parallels:~$ ps eo min_flt,maj_flt,cmd
  MINFL  MAJFL CMD
      704      1 /usr/lib/gdm3/gdm-x-session --run-script env
  59284     17 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /ru
    8439     12 /usr/lib/gnome-session/gnome-session-binary -
  73769     25 /usr/bin/gnome-shell USER=wenbo LC_TIME=zh_CN
   1346      0 ibus-daemon --xim --panel disable USER=wenbo
    376      1 /usr/lib/ibus/ibus-dconf USER=wenbo LC_TIME=z
   1659      0 /usr/lib/ibus/ibus-x11 --kill-daemon USER=wen
```

Thanks to shared libraries!

# Non-Uniform Memory Access

---

- So far all memory accessed equally
- Many systems are NUMA - speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



# Non-Uniform Memory Access (Cont.)

---

- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the **scheduler** to schedule the thread on the same system board when possible
  - Linux
    - Kernel maintains scheduling domains: does not allow threads to migrate across domains
    - A separate free-frame list for each NUMA node - allocating memory from the node it is running

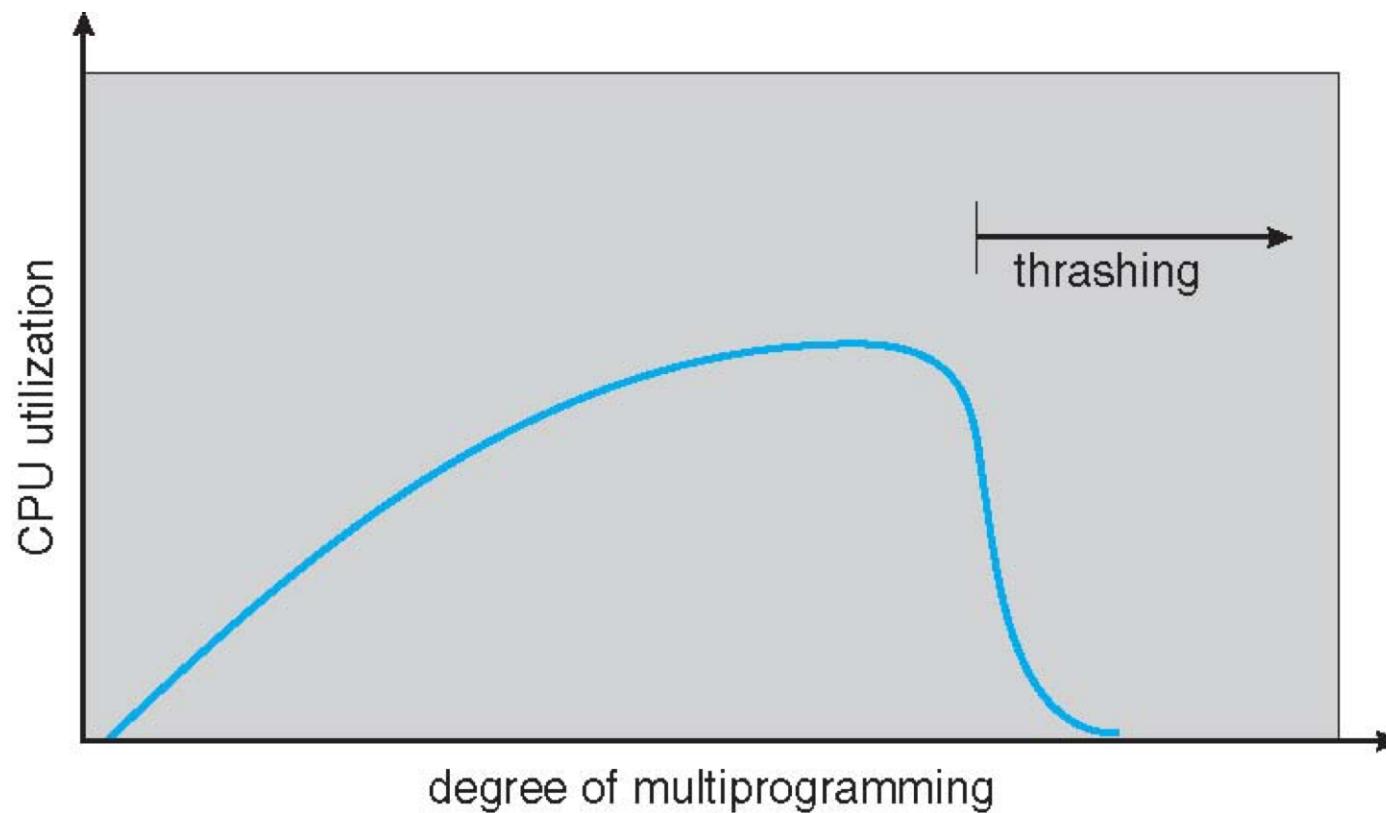
# Thrashing

---

- If a process doesn't have "enough" pages, page-fault rate may be high
  - page fault to get page, replace some existing frame
  - but quickly need replaced frame back
  - this leads to:
    - low CPU utilization ➔
    - kernel thinks it needs to increase the degree of multiprogramming to maximize CPU utilization ➔
    - another process added to the system
- **Thrashing:** a process is busy swapping pages in and out

# Thrashing

---



# Demand Paging and Thrashing

---

- Why does demand paging work?
  - process memory access has **high locality**
  - process migrates from one locality to another, localities may overlap
- Why does thrashing occur?
  - total size of locality > total memory size

# Option I

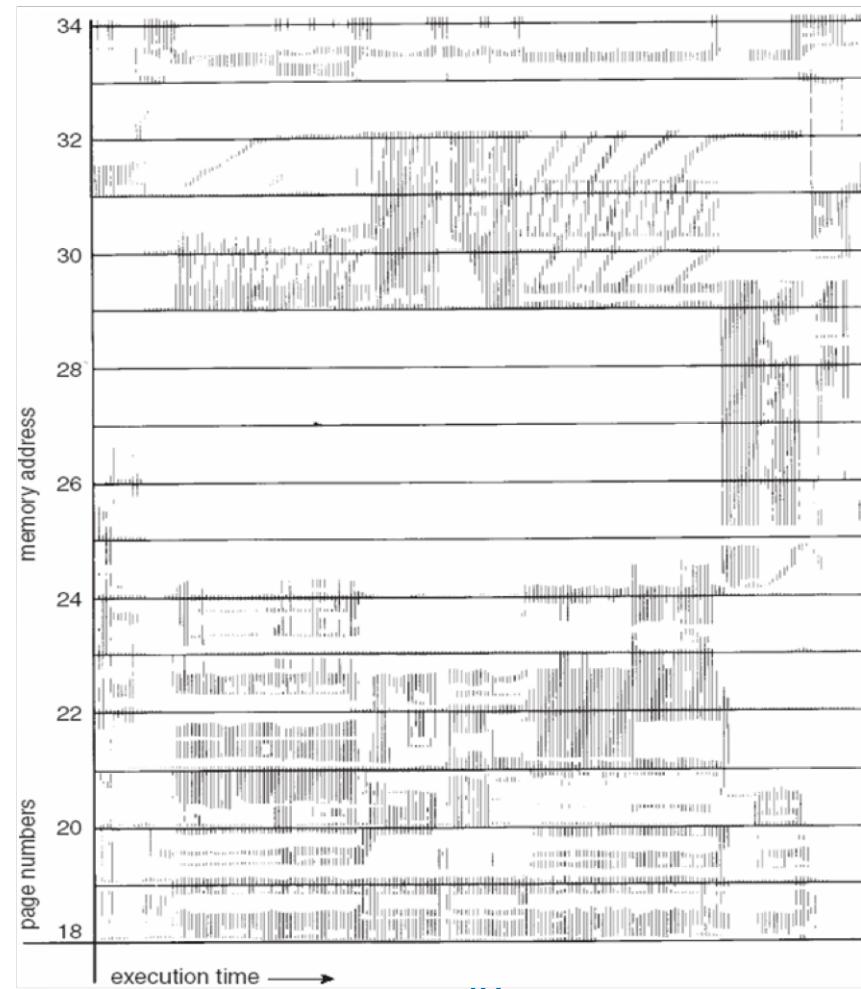
---

- Limit thrashing effects by **using local or priority page replacement**
  - One process starts thrashing does not affect others -> it cannot cause other processes thrashing

# Option II

---

Provide a process with **as many frames as it needs**. How?

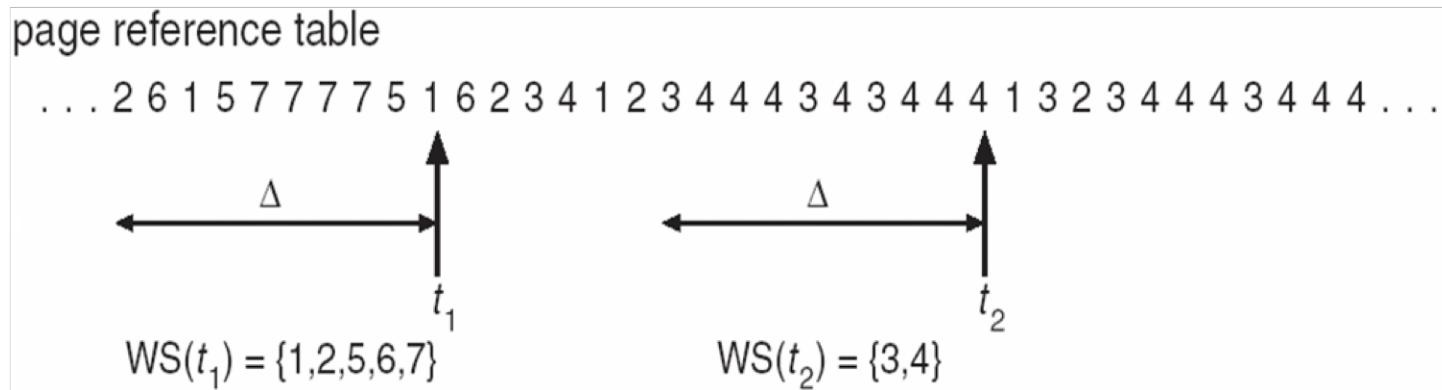


# Working-Set Model

---

- **Working-set window( $\Delta$ ):** a fixed number of page references
  - if  $\Delta$  too small  $\Rightarrow$  will not include entire locality
  - if  $\Delta$  too large  $\Rightarrow$  will include several localities
  - if  $\Delta = \infty \Rightarrow$  will include entire program
- **Working set of process  $p_i$  ( $WSS_i$ ):** total number of pages referenced in the most recent  $\Delta$  (varies in time)
- **Total working sets:**  $D = \sum WSS_i$ 
  - approximation of total locality
  - if  $D > m \Rightarrow$  possibility of thrashing
  - to avoid thrashing: if  $D > m$ , suspend or swap out some processes

# Working-Set Model



# Challenge: Keeping Track of the Working Set

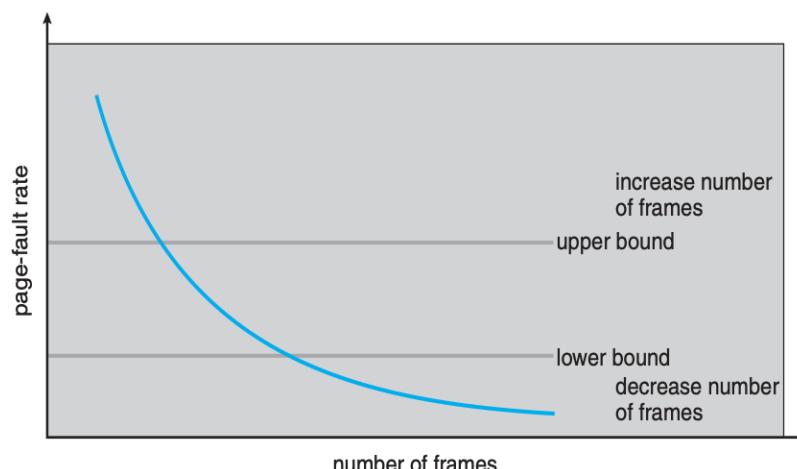
---

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units
  - Keep in memory **2 bits for each page**
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set
- Why is this not completely accurate? - we can not tell **when (in 5000 time unites)** the access occurs
- Improvement = 10 bits and interrupt every **1000 time units**

# Page-Fault Frequency

---

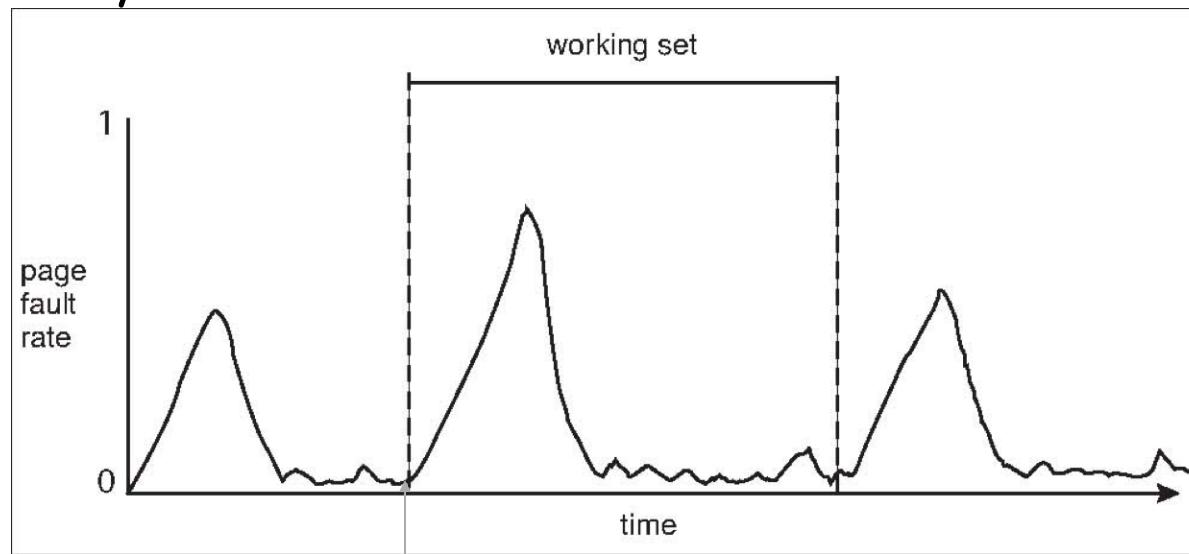
- More direct approach than WSS
- Establish “acceptable” page-fault frequency (PFF) rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Need to swap out a process if no free frames are available



# Working Sets and Page Fault Rates

---

- Assumes there is no thrashing
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Page fault increases due to new locality

# Kernel Memory Allocation

---

- Kernel memory allocation is treated differently from user memory, it is often allocated from a free-memory pool
  - kernel requests memory for structures of varying sizes → minimize waste due to fragmentation
  - Some kernel memory needs to be **physically contiguous**
    - e.g., for device I/O

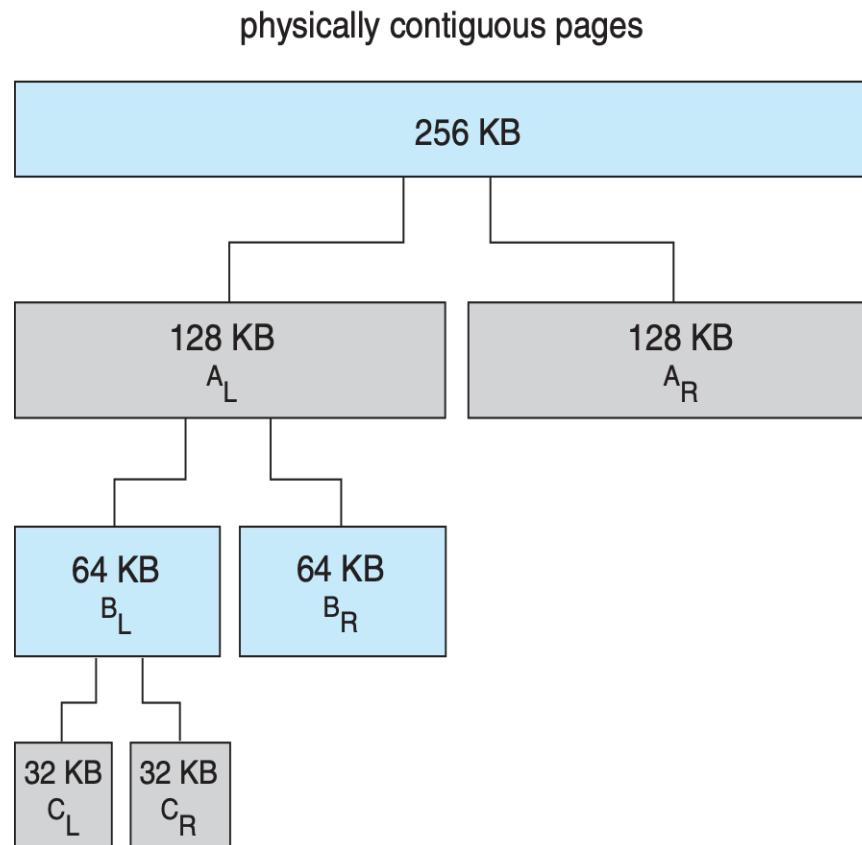
# Buddy System

---

- Memory allocated using power-of-2 allocator
  - memory is allocated in units of the size of **power of 2**
    - round up a request to the closest allocation unit
    - split the unit into two "buddies" until a proper sized chunk is available
  - e.g., assume only 256KB chunk is available, kernel requests 21KB
    - split it into  $A_l$  and  $A_r$  of 128KB each
    - further split an 128KB chunk into  $B_l$  and  $B_r$  of 64KB
    - again, split a 64KB chunk into  $C_l$  and  $C_r$  of 32KB each
    - give one chunk for the request
- advantage: it can quickly merge unused chunks into larger chunk
- disadvantage: **internal fragmentation**
  - 33k request  $\rightarrow$  64k segment

# Buddy System Allocator

---

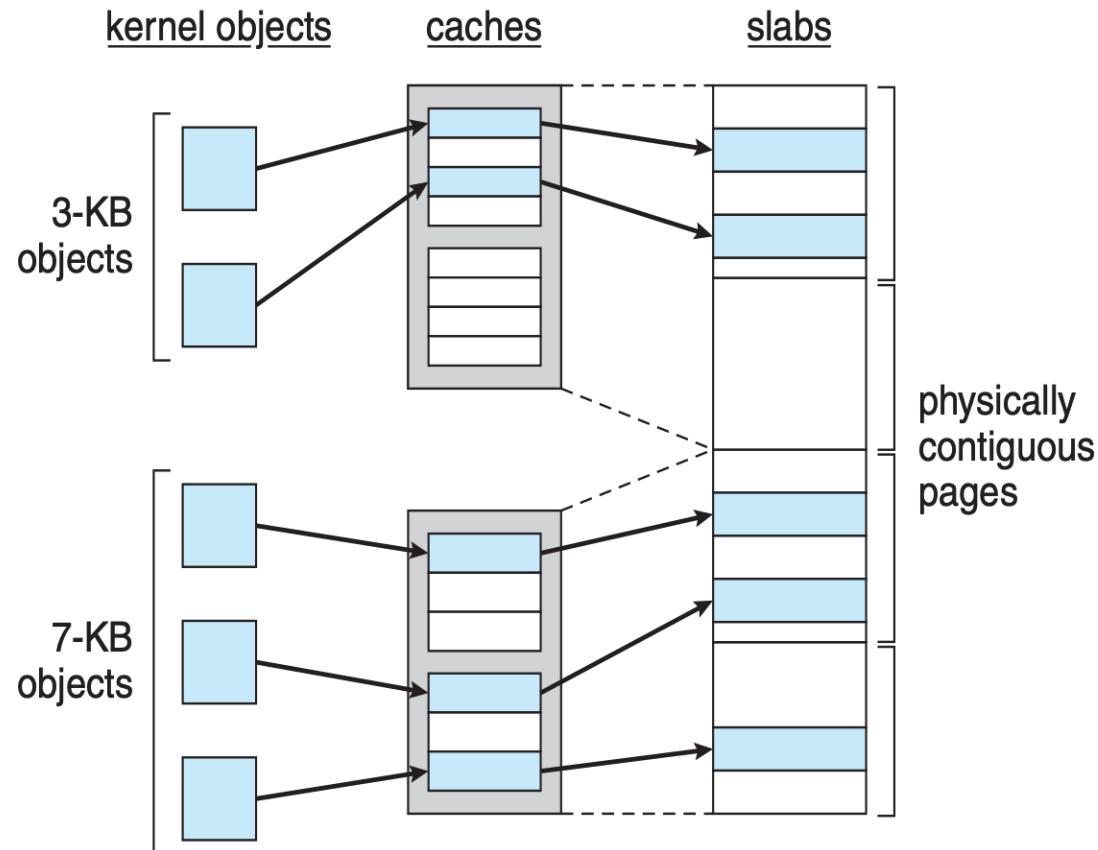


# Slab Allocator

---

- Slab allocator is a **cache of objects**
  - a **cache** in a slab allocator consists of one or more slabs
  - a **Slab contains one or more pages**, divided into **equal-sized objects**
  - kernel uses one cache for each unique kernel data structure
    - when cache created, allocate a slab, divided the slab into free objects
    - objects for the data structure is allocated from free objects in the slab
    - if a slab is full of used objects, next object comes from an empty/new slab
- Benefits: **no fragmentation** and fast memory allocation
  - some of the object fields may be reusable; no need to initialize again

# Slab Allocation



A 12k slab (3 pages) can store 4 3k objects.

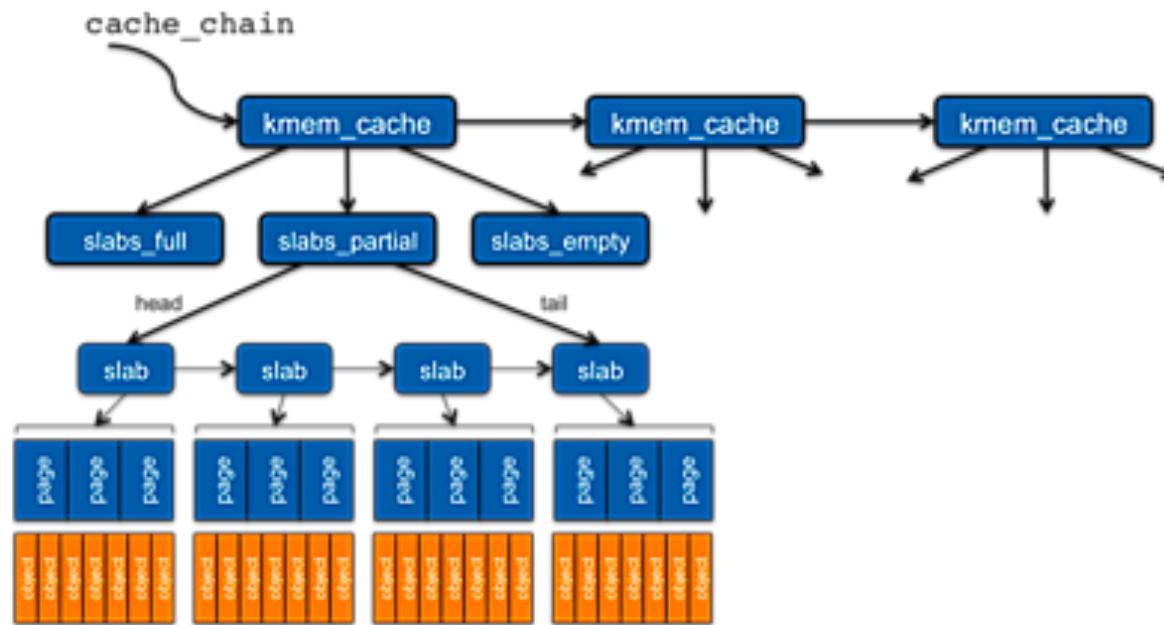
# Slab Allocator in Linux

---

- For example process descriptor is of type *struct task\_struct*
- Approx 1.7KB of memory (some old linux version)
- New task → allocate new struct from cache
  - Will use existing free *struct task\_struct*
- A Slab can be in three possible states
  - **Full** - all used
  - **Empty** - all free
  - **Partial** - mix of free and used
- Upon request, slab allocator
  - Uses free struct in **partial** slab
  - If none, takes one from **empty** slab
  - If no empty slab, create new empty

# Slab in Linux

---



# Slab Allocator in Linux (Cont.)

---

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks - maintains 3 list objects for small, medium, large objects
- SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

# Other Considerations

---

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

# 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 fraction  $a$  of these pages is used
  - Is cost of  $s * a$  save pages faults > or < than the cost of prepaging
  - $s * (1 - a)$  unnecessary pages?
  - $a$  near zero  $\rightarrow$  prepaging loses

# Page Size

---

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

# TLB Reach

---

- **TLB reach:** the amount of memory accessible from the TLB
  - $\text{TLB reach} = (\text{TLB size}) \times (\text{page size})$
- Ideally, the working set of each process is stored in the TLB
  - otherwise there is a high degree of page faults
- **Increase the page size to reduce TLB pressure**
  - it may increase fragmentation as not all applications require large page sizes
  - multiple page sizes allow applications that require larger page sizes to use them without an increase in fragmentation

# Other Issues: Program Structure

---

- Program structure can affect page faults

- int[128,128] data; each row is stored in one page
  - Program 1:

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

$128 \times 128 = 16,384$  page faults (assume TLB only has one entry)

- Program 2:

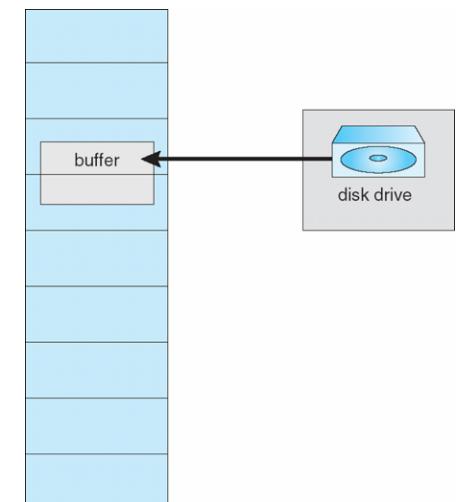
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

# I/O interlock

---

- I/O Interlock - Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be **locked from being selected for eviction by a page replacement algorithm**



# Windows XP

---

- Uses demand paging with clustering
  - clustering brings in **pages surrounding the faulting page**: locality
- Processes are assigned **working set** minimum and set maximum
  - *wsmin*: minimum number of pages the process is guaranteed to have
  - *wsmax*: a process may be assigned as many pages up to its *wsmax*
- When the amount of free memory in the system falls below a threshold:
  - automatic working set trimming to restore the amount of free memory
  - it removes pages from processes that have more pages than the *wsmin*

# Takeaway

---

- Page fault
  - Valid virtual address, invalid physical address
- Page replacement
  - FIFO, Optimal, LRU, 2nd chance
- Thrashing and working set
- Buddy system
- slab