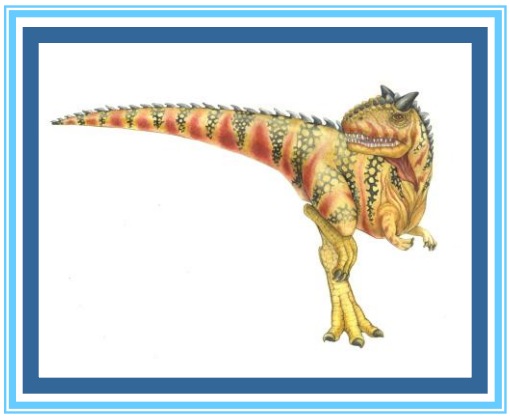
Chapter 10: Virtual Memory

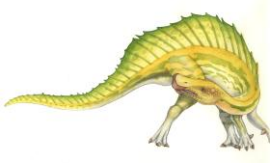


Silberschatz, Galvin and Gagne ©2018



Chapter 10: Virtual Memory

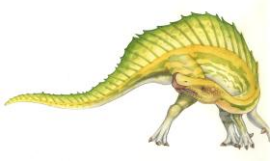
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples





Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

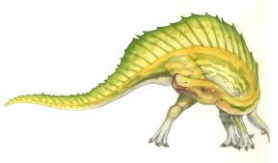


Troy University Resources



Why Virtual Memory Helps

- Programs often contain code and data that are not used during a particular execution phase.
- Virtual memory lets the OS keep only the active pages resident while providing a larger logical address space.
- This can reduce startup I/O and allow more useful work to fit in RAM.
- Performance depends on locality, memory pressure, and storage cost. Excessive page faults can make execution much slower.





Virtual memory

- **Virtual memory** – separation of user 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
  - More programs running concurrently
  - Less I/O needed to load or swap processes



Troy University Resources



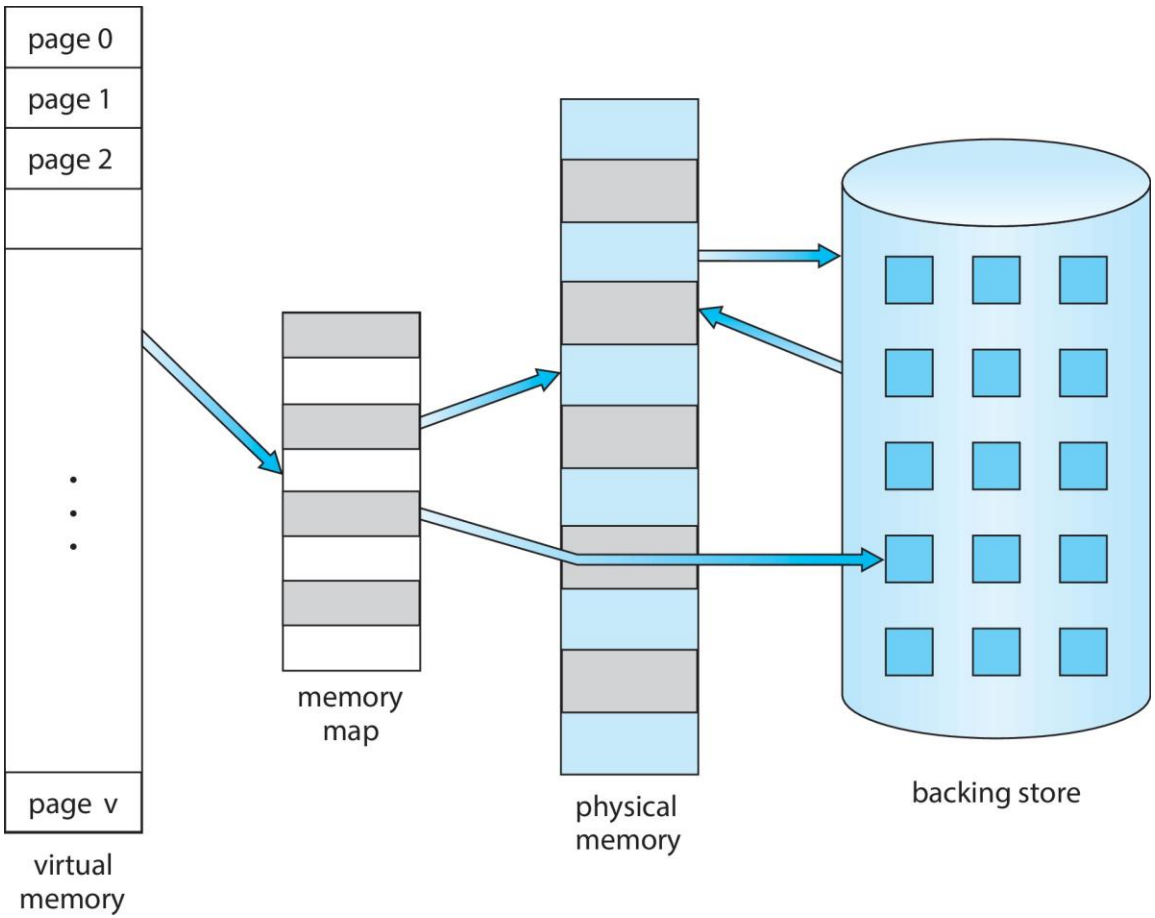
Virtual Address Spaces

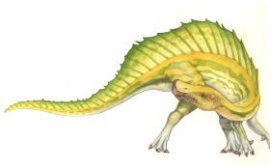
- Each process sees a virtual address space with mapped regions, permissions, and unmapped holes.
- Different processes can map the same virtual address to different physical frames.
- Mappings can also intentionally share a physical frame across address spaces.
- The MMU translates resident mappings. The OS handles faults, creates mappings, and supplies pages as needed.
- Demand paging is the principal model used in this chapter.





Virtual Memory That is Larger Than Physical Memory



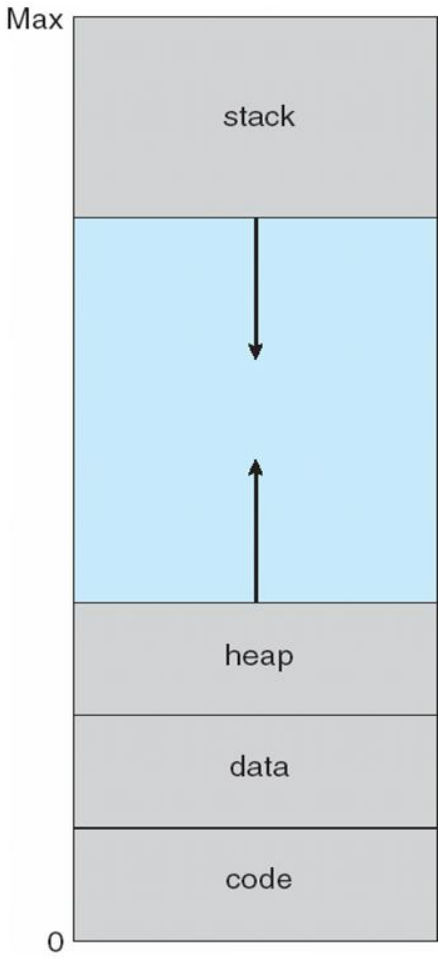


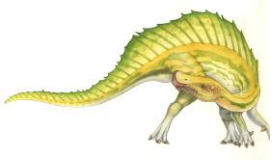
Troy University Resources



Virtual-address Space

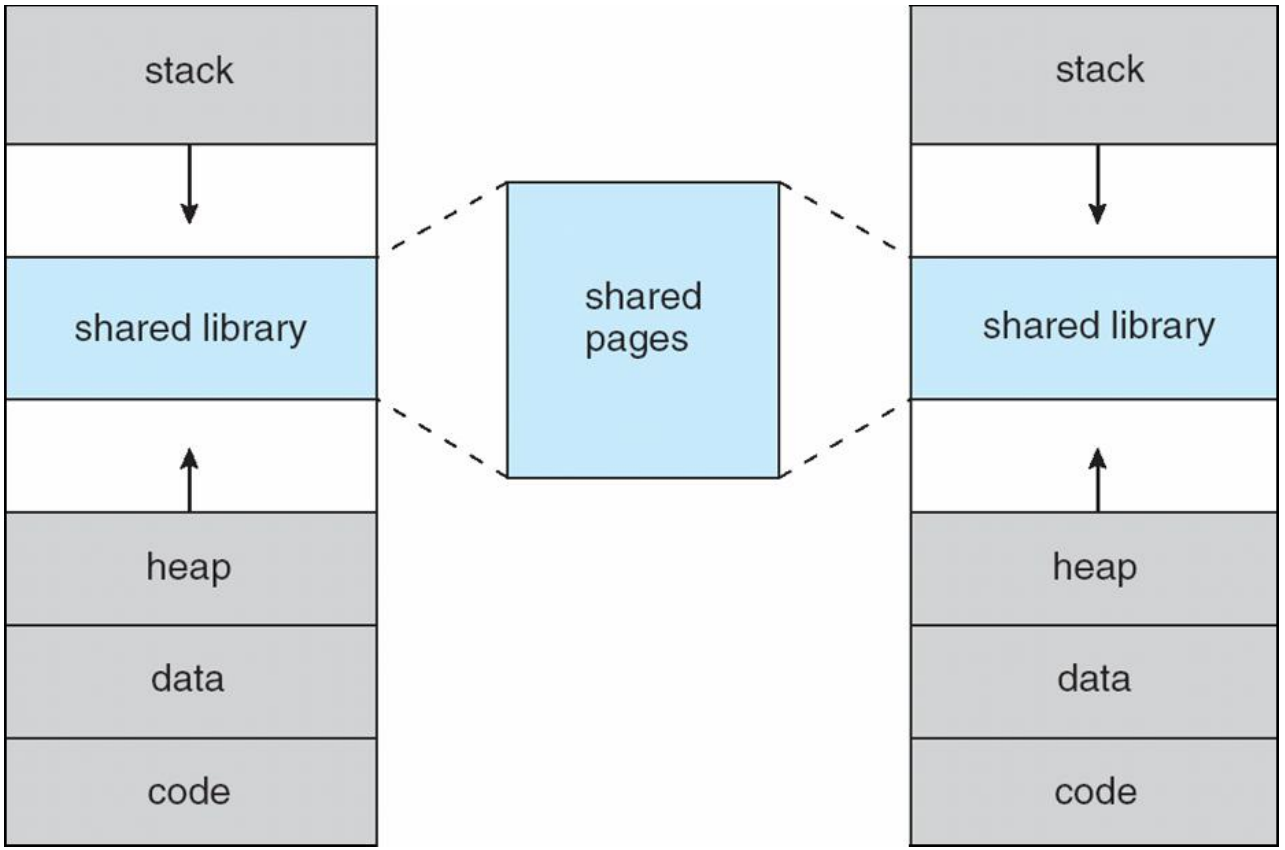
- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

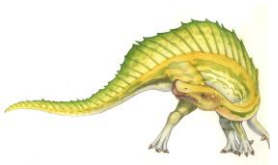






Shared Library Using Virtual Memory





Troy University Resources



Demand Paging

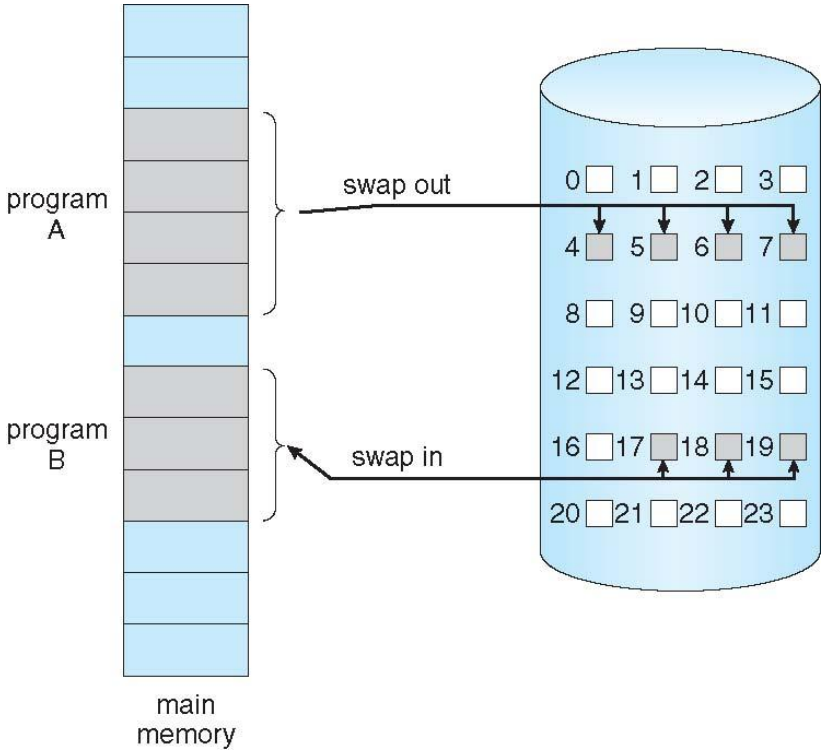
- A pager supplies a page when the process first needs it, rather than loading every page in advance.
- A resident, permitted mapping can be used immediately.
- A legal nonresident page may require zero-fill, a cached page, decompression, or a read from backing storage.
- An unmapped address or disallowed access is a different fault case and may cause an exception or process termination.
- Demand paging saves work when pages are never touched, but faults have a cost.





Paging Between RAM and Backing Storage

- Demand paging loads a page when an access requires it.
- A not-present file or swap-backed page may require a storage read.
- Already resident pages need no storage read.
- The diagram illustrates transfers between backing storage and RAM.



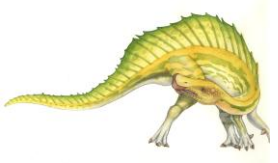


Troy University Resources



Demand-Paging Requirements

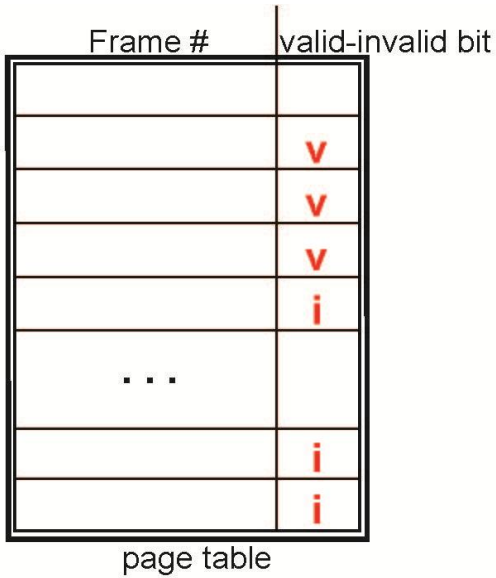
- Hardware identifies a missing or disallowed translation and traps to the kernel.
- The OS records which virtual regions are legal and where their contents come from.
- File-backed pages can be loaded from a file. New anonymous pages can be zero-filled.
- The CPU must support correct resumption or restart of the faulting instruction.
- The program usually uses ordinary loads and stores without explicit paging code.





Present and Not-Present Pages

- A present bit records whether a page has a resident physical frame.
- A not-present access traps. The OS then checks whether the virtual address is legal.



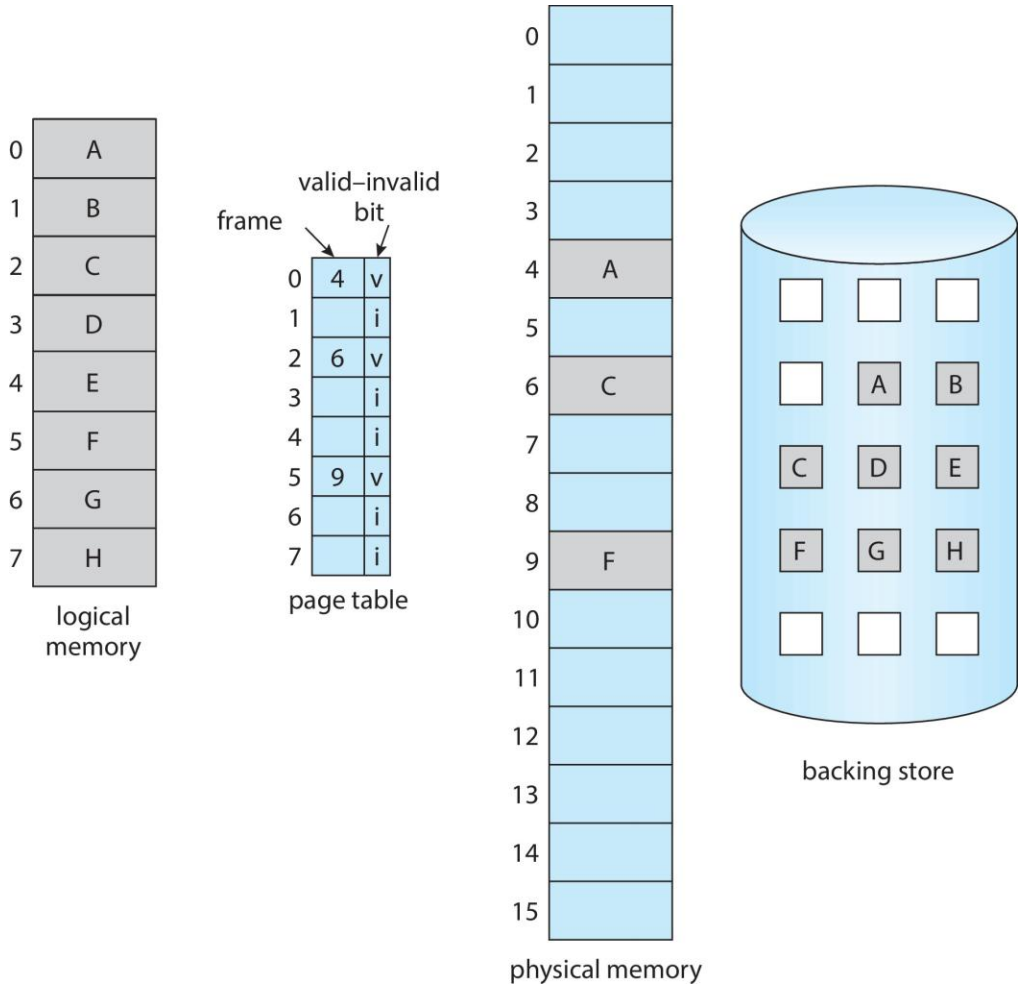
Legal but nonresident pages can be supplied on demand. An illegal mapping or forbidden access is a different fault case.



Troy University Resources



Page Table When Some Pages Are Not in Main Memory

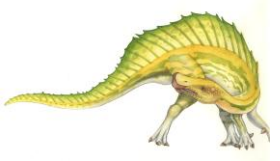






Handling a Page Fault

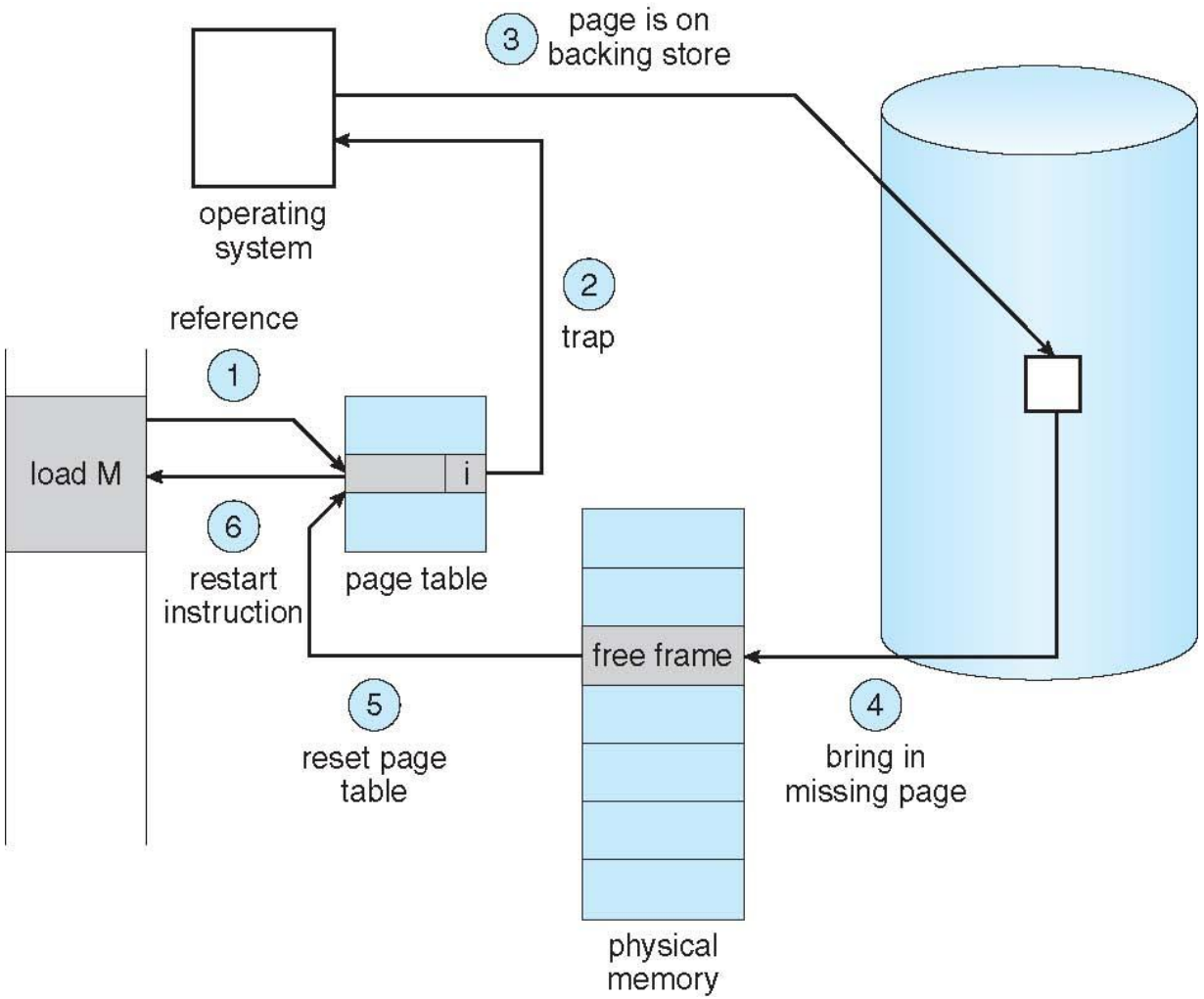
- Trap to the OS and identify the address and type of access.
- Check the virtual mapping and permissions. Reject or report an invalid access.
- For a legal fault, obtain a frame and supply the contents by the appropriate mechanism.
- Update the page tables and required translation-cache state.
- Resume the process and restart or complete the faulting instruction correctly.



Troy University Resources



Steps in Handling a Page Fault (Cont.)







Pure Demand Paging

- In the extreme model, execution starts with no user pages resident. The first instruction fetch faults.
- Later first touches fault only for pages that are not already mapped and resident.
- One instruction can touch several pages, so the hardware must handle partial progress and restart correctly.
- Locality limits how many pages need to remain active during one execution phase.
- Swap space is useful for anonymous-page eviction, but file-backed and zero-filled demand paging can operate without it.

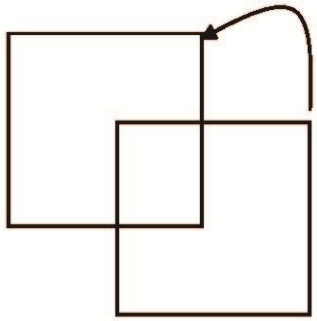


Troy University Resources



Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?



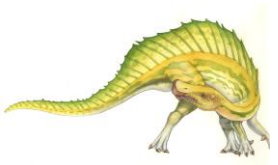


Free Frames and Zero-Fill

- A free-frame list supplies physical frames for allocation.
- A page fault can map an existing page or allocate a zero-filled page without reading disk.



- A file or swap-backed page may instead require a read from storage.
- Clear a frame before exposing it to another protection domain.
- At boot, some RAM is already reserved for the kernel, firmware, and devices.

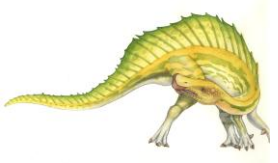


Troy University Resources



Stages in Demand Paging: Disk Read

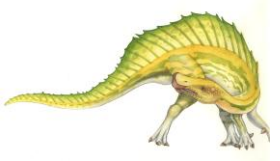
1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   a) Wait in a queue for this device until the read request is serviced
   b) Wait for the device seek and/or latency time
   c) Begin the transfer of the page to a free frame





Disk-Read Fault: Completion

- While the faulting thread waits for I/O, the scheduler can run another ready thread.
- The storage subsystem reports completion and the kernel records that the page is available.
- The kernel installs or updates the translation, then makes the faulting thread ready.
- When the scheduler runs it again, execution resumes at the faulting instruction.
- This is a disk-read example. Other fault types can complete without these I/O steps.



Troy University Resources



Effective Access Time with Page Faults

- Let p be the page-fault probability per memory reference.
- Let M be normal memory-access time and F the average total time for a faulting reference.
- EAT = (1 - p) × M + p × F.
- F can include trap handling, victim writeback, a page read, scheduling delay, and restart. Include each component only once.
- Faults resolved in RAM have a different cost from faults requiring storage I/O.





Demand-Paging Cost Example

- Assume M = 200 ns and F = 8 ms = 8,000,000 ns, including fault service and restart.
- EAT = (1 - p) × 200 + p × 8,000,000 = 200 + 7,999,800p ns.
- At p = 0.001: EAT = 8199.8 ns ≈ 8.2 µs, about 41 times the normal access time.
- For less than 10% overhead: EAT < 220 ns, so p < 20 / 7,999,800 ≈ 2.5×10^{-6}.
- That is approximately fewer than one fault per 400,000 memory references.

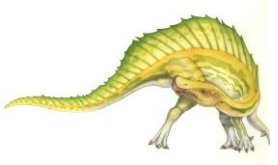


Troy University Resources



Paging and Backing Storage

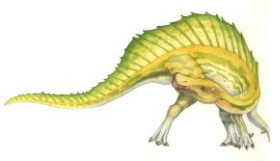
- A clean file-backed page can be discarded and later reloaded from its file.
- A dirty shared file mapping needs appropriate file writeback. Anonymous data needs another preservation mechanism before its frame can be reused.
- Swap storage, in-memory compression, and page caches serve different roles.
- Swap I/O is not universally faster than file I/O. Device layout, filesystem behavior, and workload matter.
- Android's zRAM is an example of compressed swap in RAM, so mobile systems cannot be described as universally having no swap.





Copy-on-Write and fork()

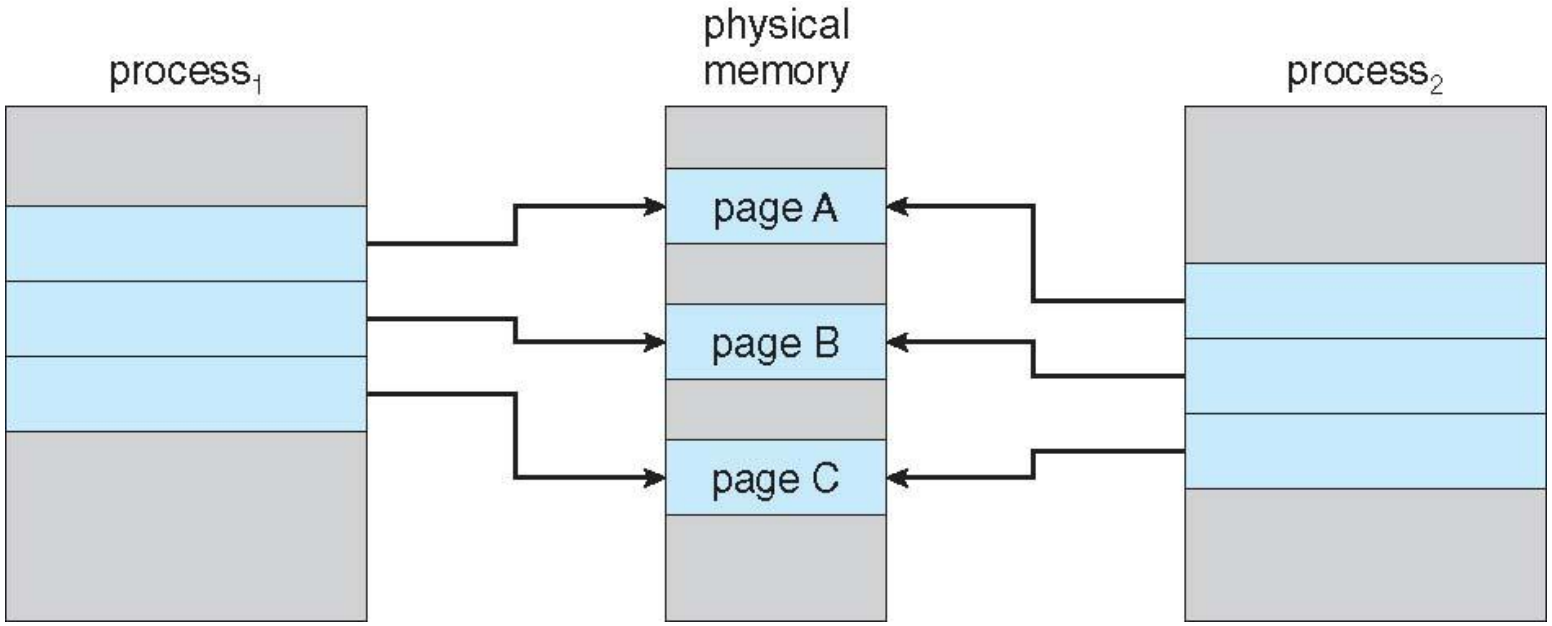
- After fork(), parent and child can initially share physical pages while each keeps its own virtual address space.
- A write to a private copy-on-write mapping faults. The OS provides a private copy when sharing requires it.
- This avoids copying pages the child never modifies, especially before exec().
- vfork() has different semantics: the child temporarily shares the parent's address space and suspends the parent until execve() or _exit().
- The child must obey vfork's strict restrictions. vfork is not copy-on-write fork.

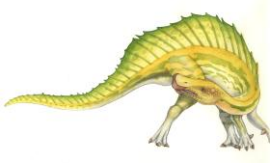


Troy University Resources



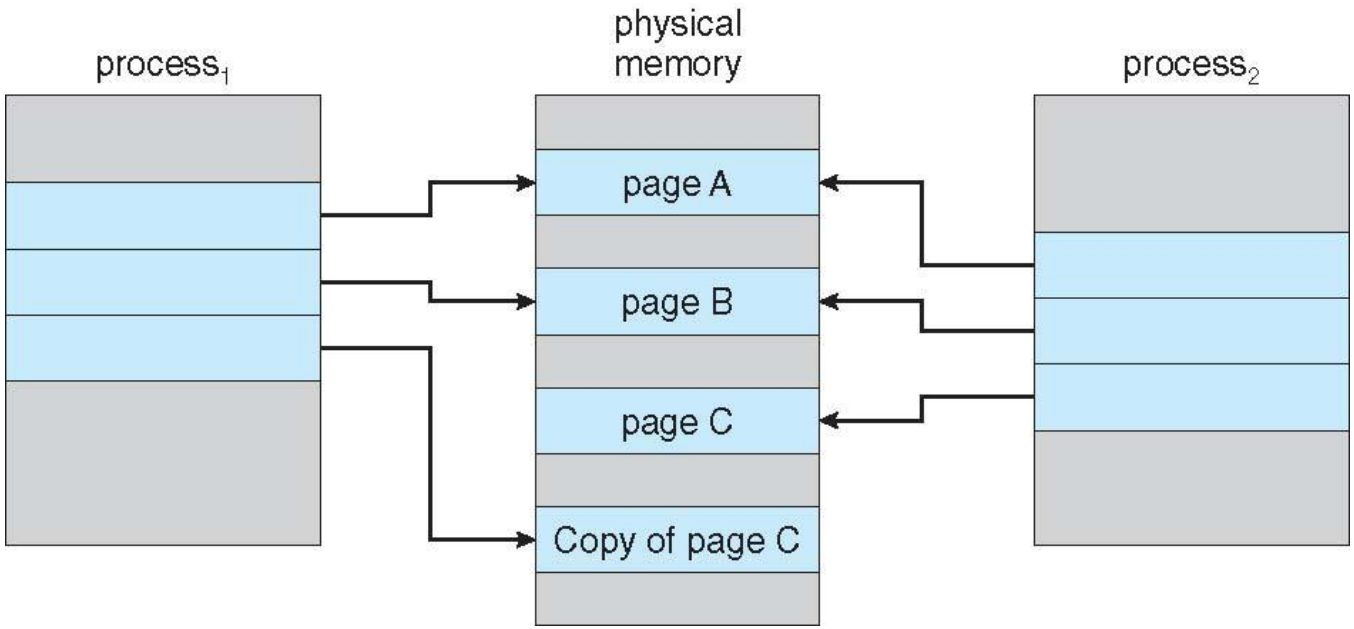
Before Process 1 Modifies Page C







After Process 1 Modifies Page C

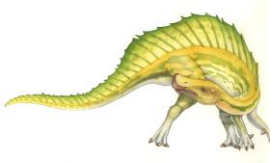






What Happens if There is no Free Frame?

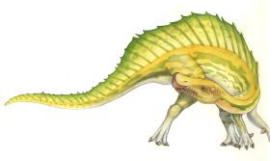
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result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





Replacing a Page

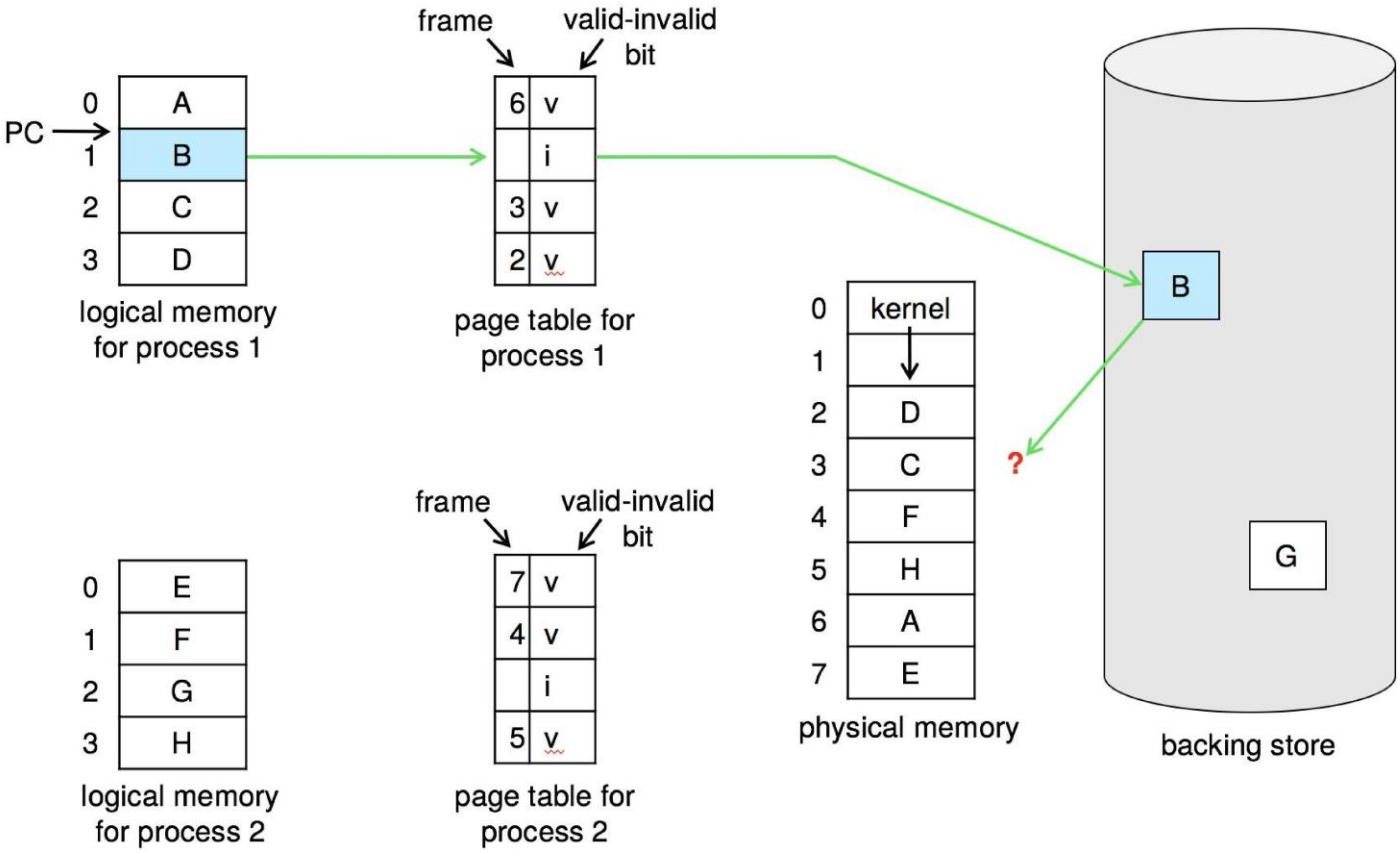
- When a free frame is unavailable, select an eligible victim frame according to the replacement policy.
- A clean file-backed page can often be discarded without a write.
- Preserve dirty contents when they are still needed, through file writeback, swap, or another supported mechanism.
- Invalidate obsolete mappings and TLB entries before reusing a frame.
- A page must not be reused while it is pinned for an operation that requires its physical location.



Troy University Resources



Need For Page Replacement







Basic Page Replacement

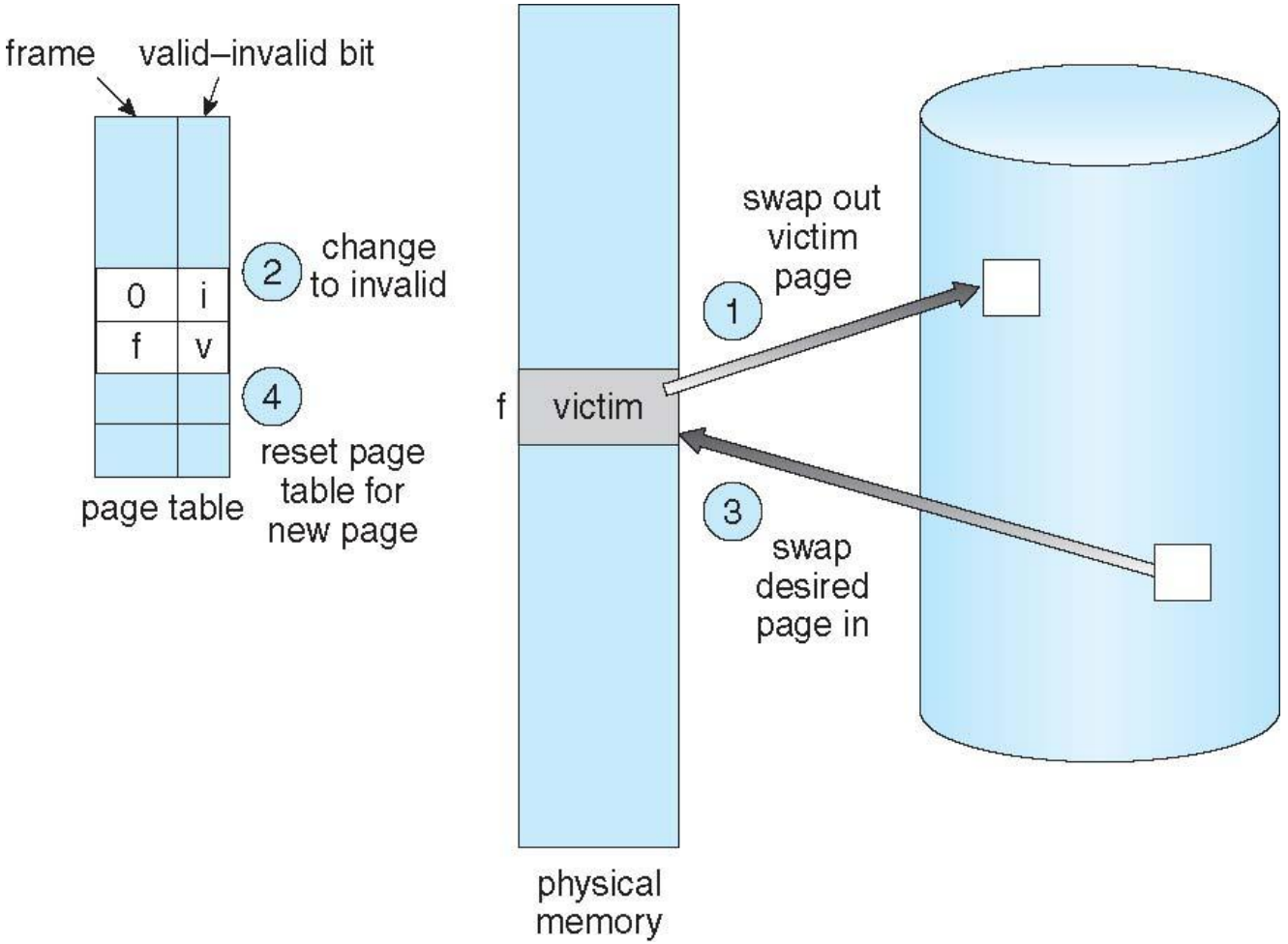
- Validate the requested mapping and determine how to supply its contents.
- Use a free frame, or select a victim and preserve its contents if required.
- Update affected mappings and translation caches before reusing the frame.
- Load, copy, decompress, or zero-fill the requested page as appropriate.
- Install the new translation and resume execution. A disk-backed case can require both a victim write and a requested-page read.



Troy University Resources



Page Replacement







Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **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 a page that remains resident does not cause another not-present fault
  - Results depend on number of frames available
- The following FIFO, OPT, and LRU examples use this reference string:

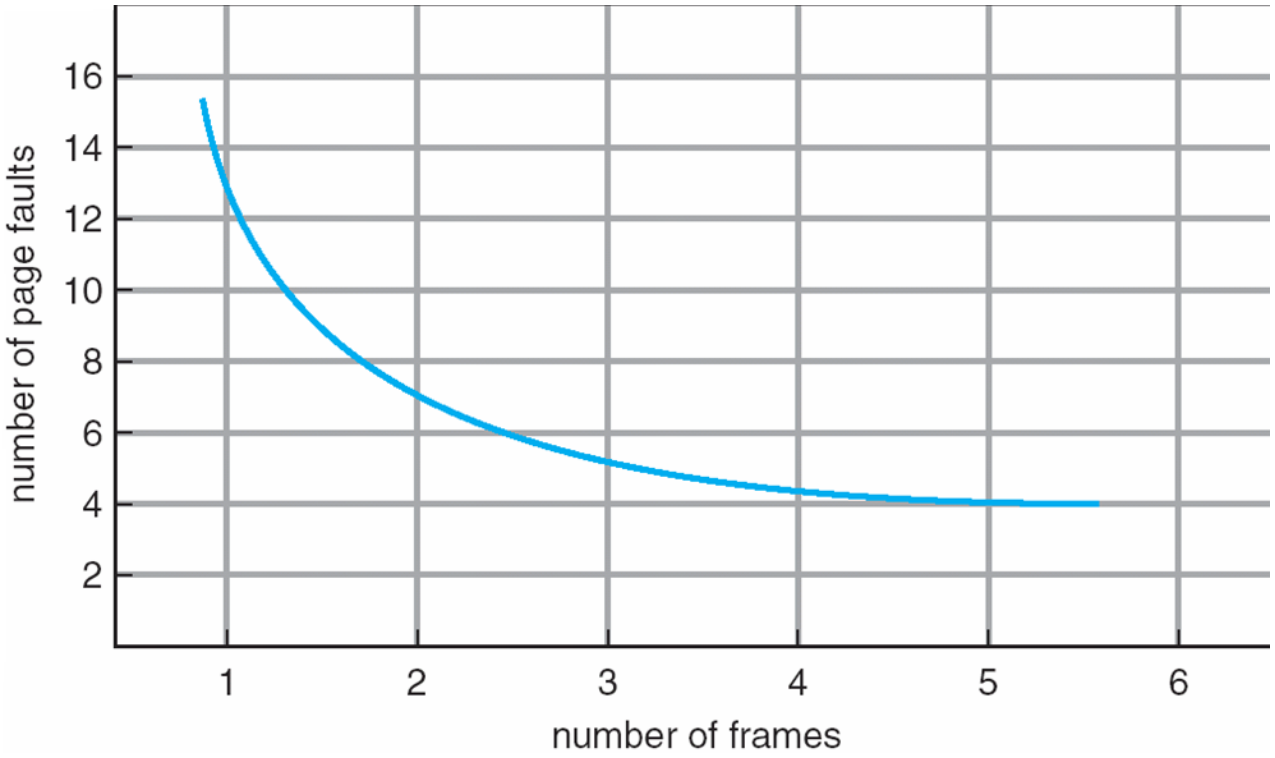
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



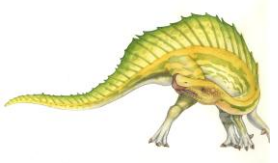
Troy University Resources



Page Faults and Frame Count



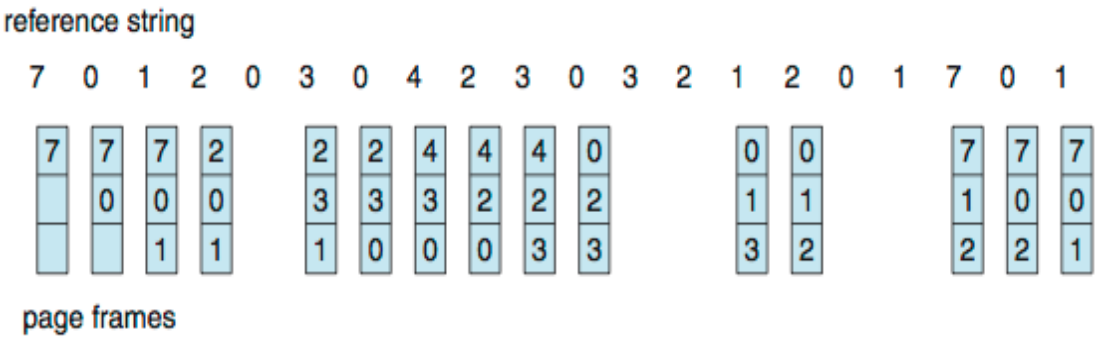
A typical trend. FIFO can violate monotonicity through Belady's anomaly.





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

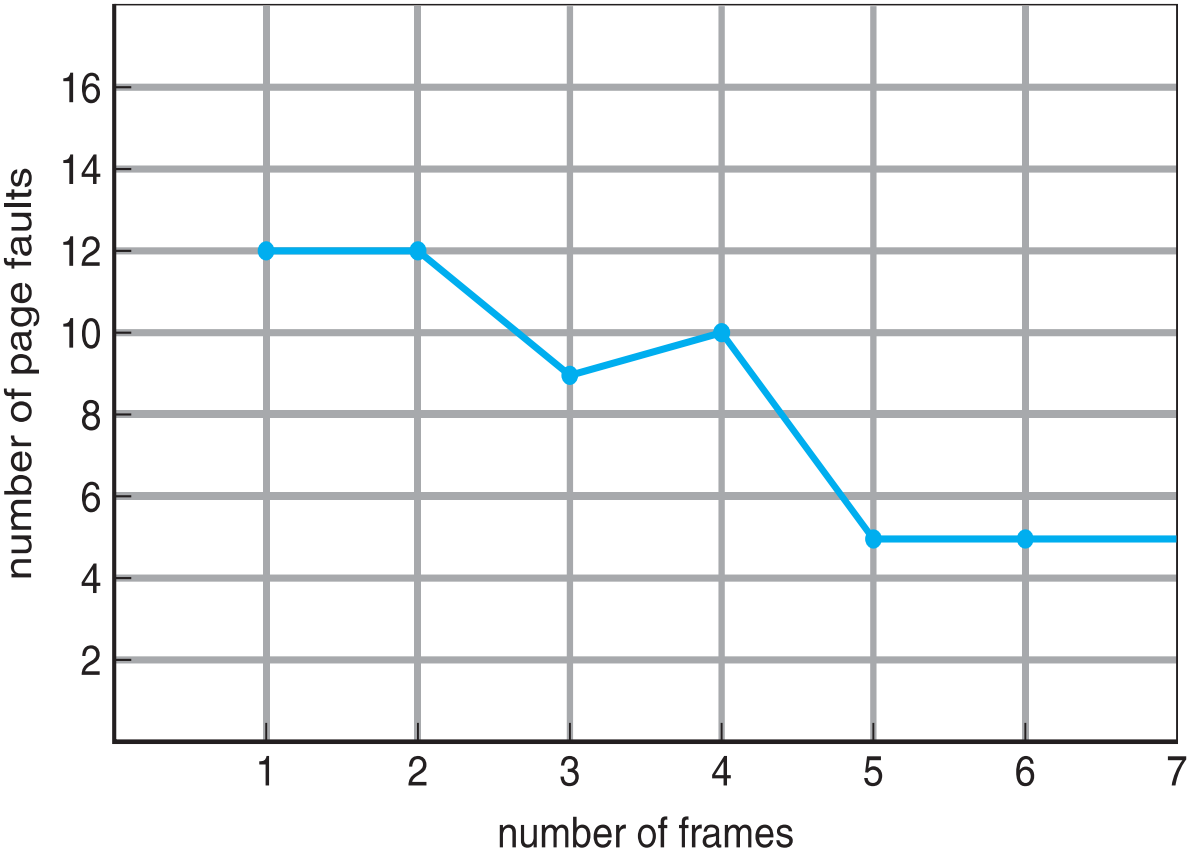
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

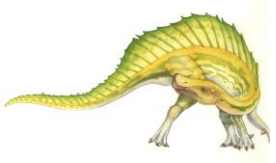


Troy University Resources



FIFO Illustrating Belady's Anomaly

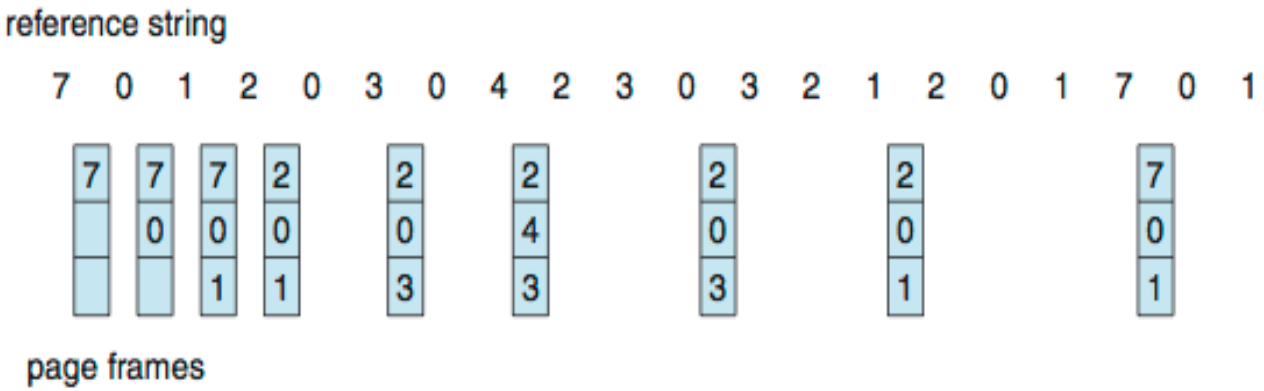






Optimal Algorithm

- Replace page that will not be used for longest period of time
  - With 3 initially empty frames, the reference string requires 9 faults under OPT.
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

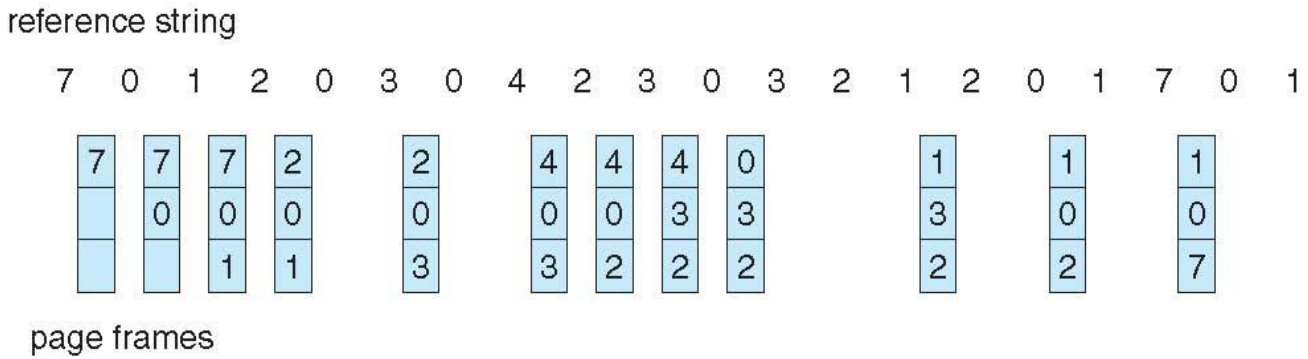




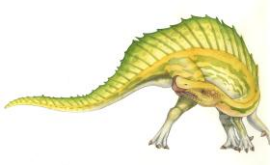


Least Recently Used (LRU) Algorithm

- 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



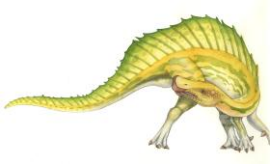
- With 3 initially empty frames: LRU = 12 faults, FIFO = 15, OPT = 9.
- Exact LRU is costly; practical systems commonly approximate recent use.
- But how to implement?





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
    - updates links and head/tail state
  - But each update more expensive
  - No search for replacement

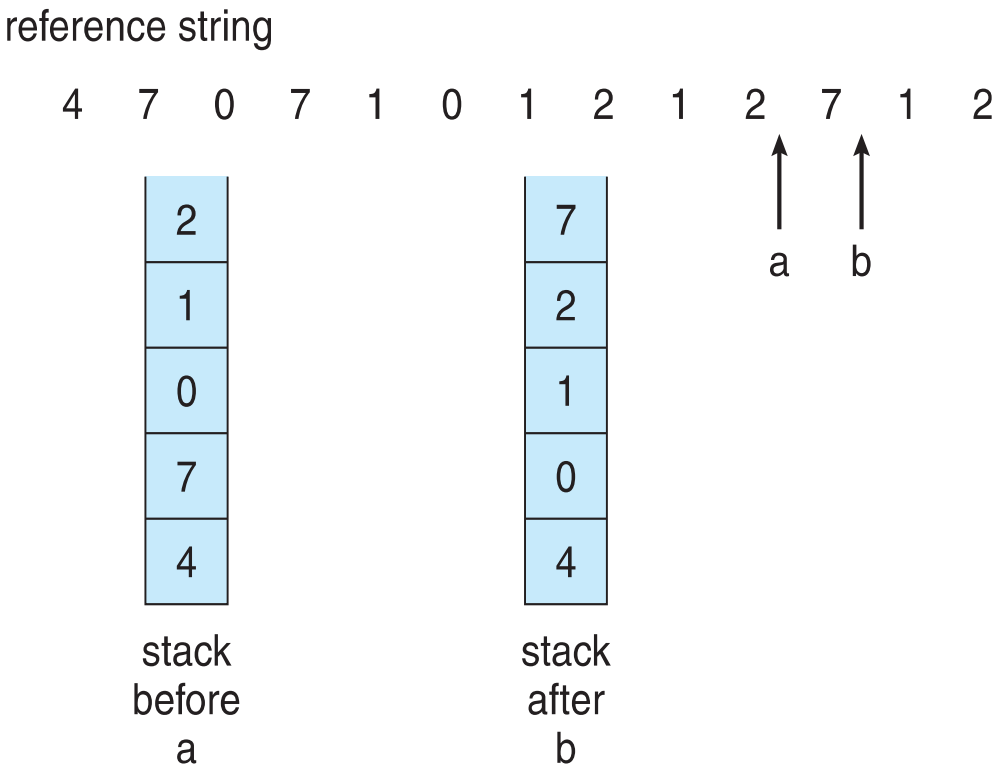


Troy University Resources



LRU Algorithm (Cont.)

- 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

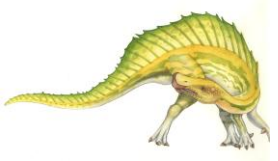






Approximating LRU

- Exact LRU needs a record of access order and can be too expensive to update on every reference.
- Hardware accessed/reference bits provide a cheaper indication that a page was used.
- The OS periodically samples or clears those bits to estimate recency.
- A clear bit identifies a candidate, but does not fully order all pages by last access time.
- Clock and aging algorithms trade replacement quality for lower tracking overhead.

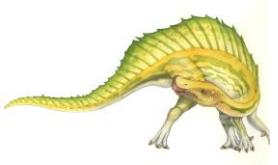


Troy University Resources



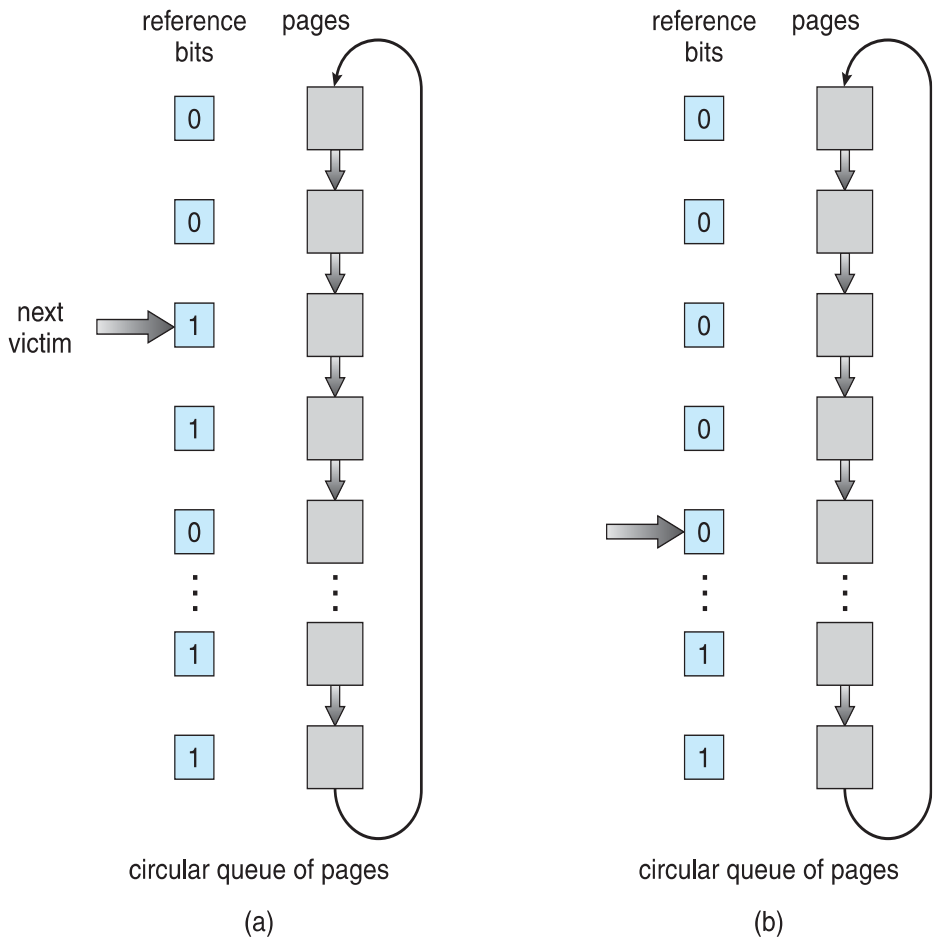
LRU Approximation Algorithms (cont.)

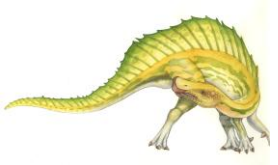
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same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





Second-chance Algorithm



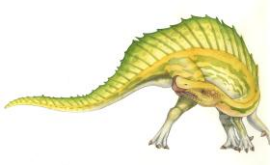


Troy University Resources



Enhanced Second-Chance Algorithm

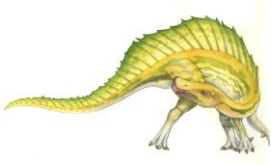
- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0): not recently used and not modified; preferred victim in this policy.
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (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 Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **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

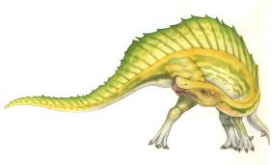


Troy University Resources



Page-Buffering Algorithms

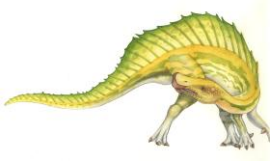
- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected





Applications and Page Replacement

- A database may keep its own buffer pool while the OS also caches the same file data.
- This double caching can consume memory, though application and OS caches can have different purposes.
- Direct I/O can bypass some page-cache buffering when the OS and filesystem support it.
- Direct I/O has alignment and consistency requirements. It does not automatically bypass locking or provide complete durability.
- The best approach depends on the application and storage workload.

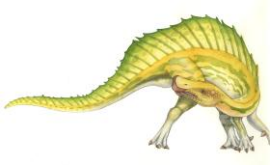


Troy University Resources



Allocation of Frames

- Each process needs ***minimum*** number of frames
- 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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \square 62 \square 4$$
$$a_2 = \frac{127}{137} \square 62 \square 57$$

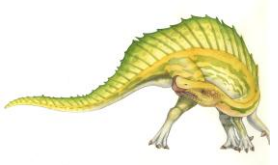


Troy University Resources



Priority-Based Frame Allocation

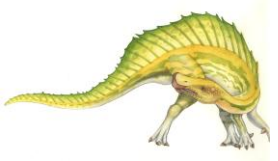
- A policy can assign more frames to higher-priority work instead of allocating only by process size.
- On a fault, it may choose a victim within the requesting process or from lower-priority work, subject to its rules.
- Priority numbers have no universal direction. State explicitly which value means higher priority.
- Low-priority processes can suffer poor progress if higher-priority demand dominates. A fair policy needs limits or admission control.





Global and Local Replacement

- Global replacement can choose an eligible victim from frames used by multiple processes.
- It adapts memory allocation to demand, but one process can affect another process's residency.
- Local replacement restricts victims to the process's own allocation.
- This gives more predictable isolation, but can leave one process short of frames while another has spare capacity.
- Real systems can apply additional limits and policies by process, memory group, NUMA node, or priority.

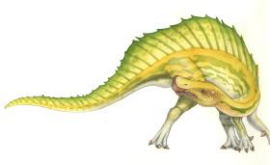


Troy University Resources



Reclaiming Pages

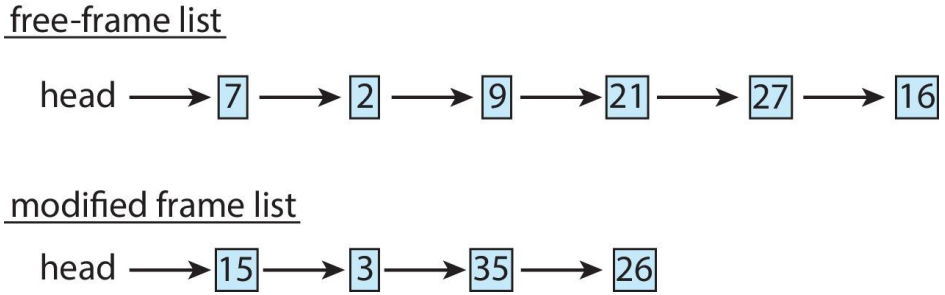
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This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



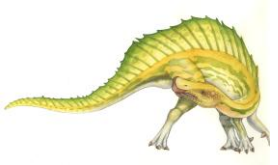


Memory Compression

Compression can store several pages in less RAM, reducing the need to write them to swap.



- The figure shows the initial free-frame list.
- Selected pages can be compressed into fewer physical frames. Their released frames become available for allocation.
- Compression trades CPU time and metadata for reduced memory use. Some pages compress poorly.

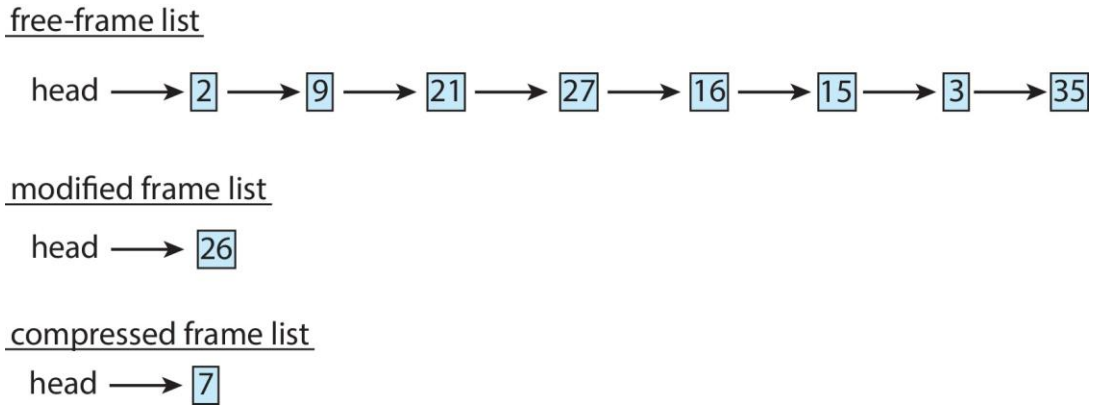


Troy University Resources



Memory Compression: Reusing Frames

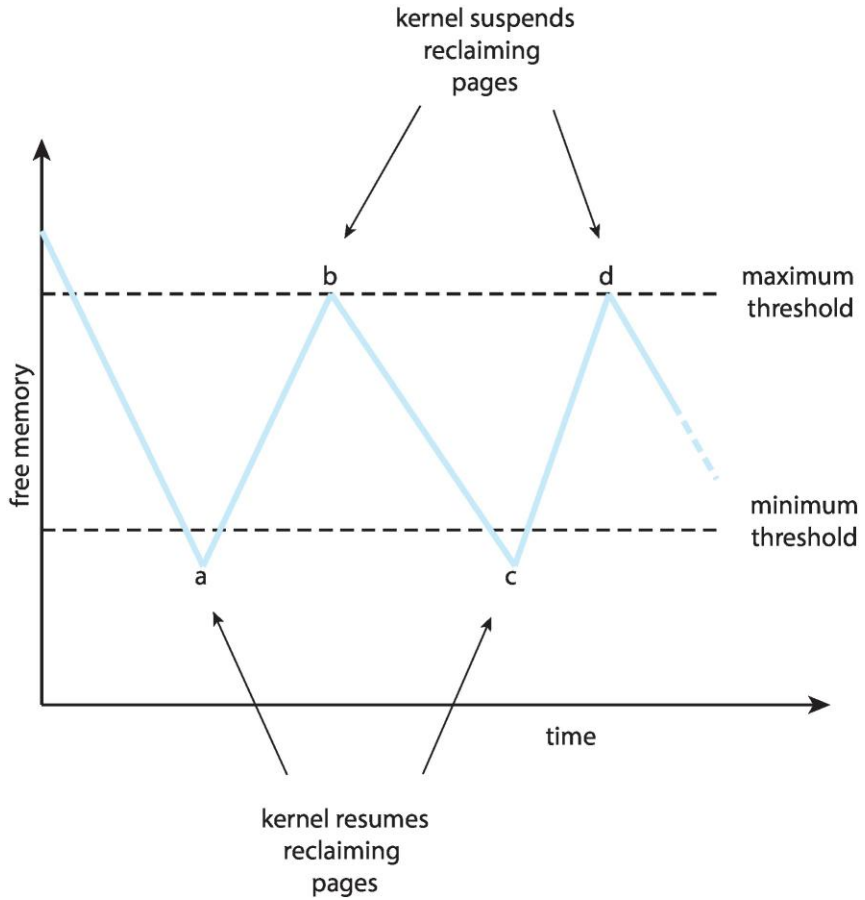
- The figure illustrates several pages packed into fewer RAM frames.
- Virtual pages retain their identity. The OS tracks where their compressed contents are stored.
- On access, the OS restores a page into an ordinary frame. Disk I/O is needed only if the data is no longer available in RAM.







Reclaiming Pages Example

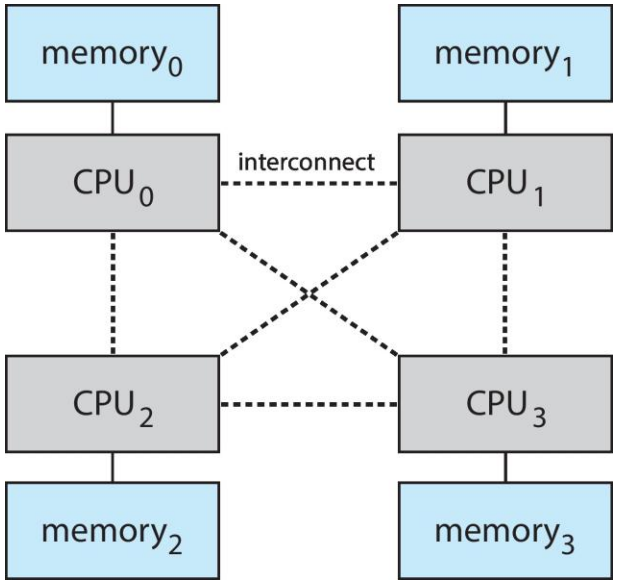


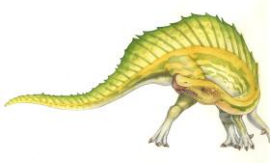




Non-Uniform Memory Access

- So far, we assumed that 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







NUMA Placement

- Accessing local memory is generally faster than accessing a remote NUMA node's memory.
- The OS can place a thread near its pages or allocate pages near the CPU that will use them.
- Placement must also account for load balancing, shared pages, and memory capacity.
- Solaris locality groups (lgroups) represent CPU and memory locality for scheduling and allocation.
- Keeping every thread and page of a process on one node is not always possible or optimal.

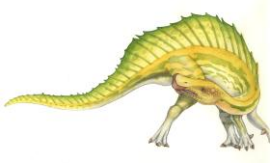


Troy University Resources



Thrashing

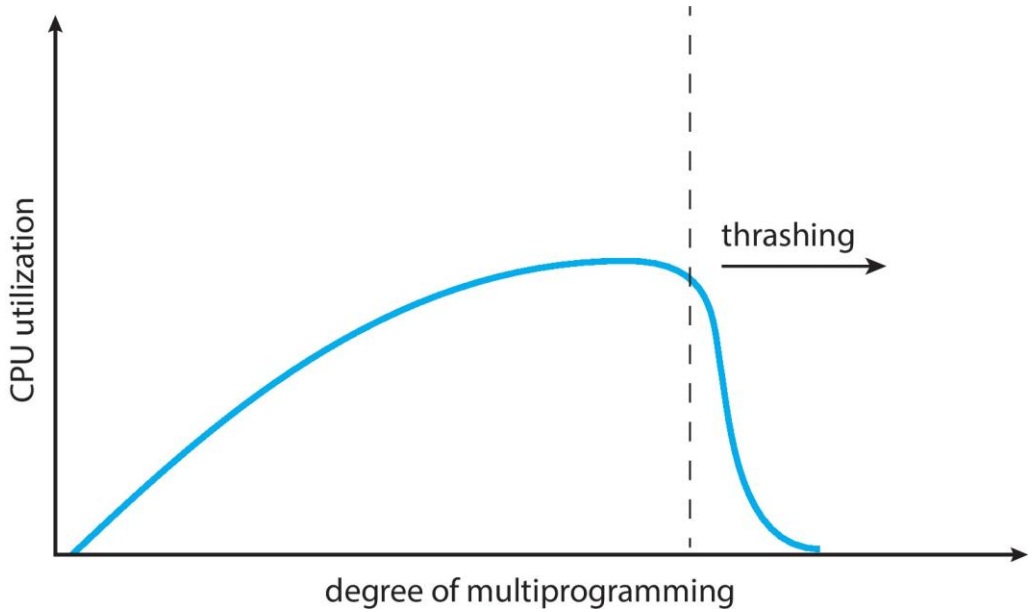
- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming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

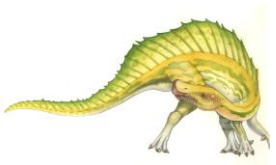




Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out

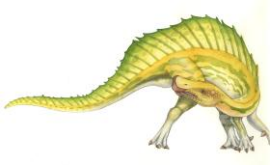






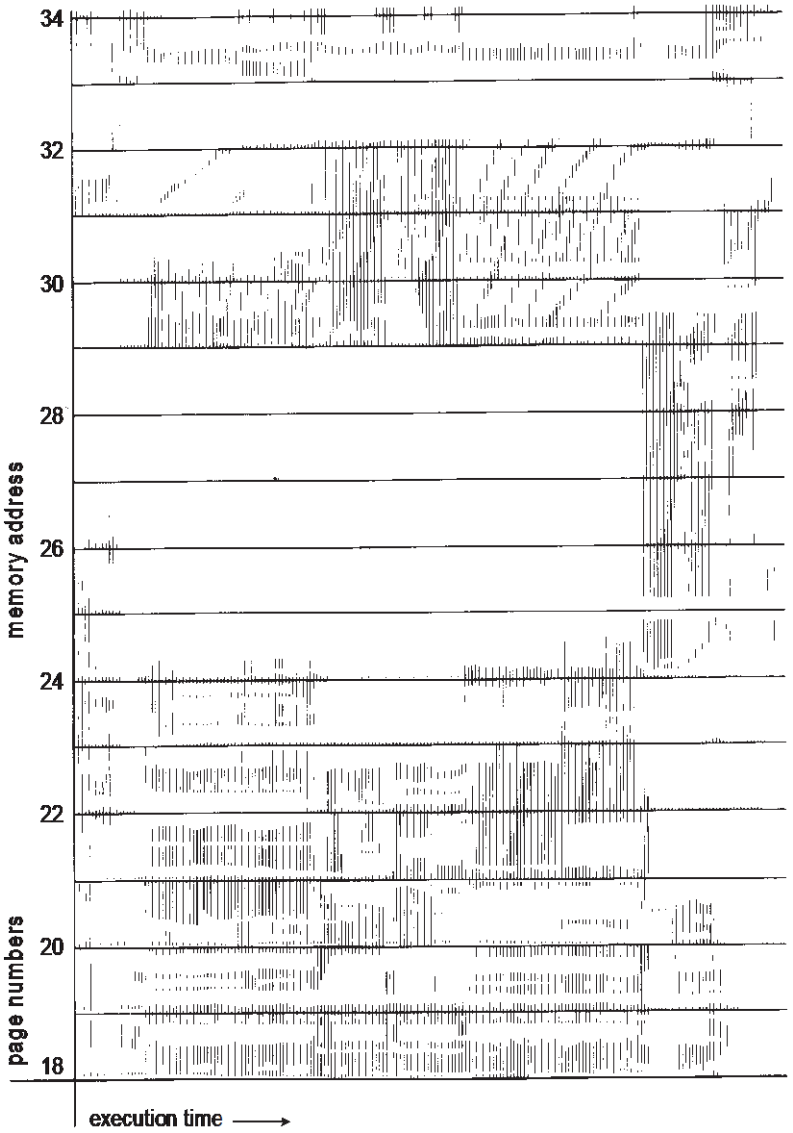
Demand Paging and Thrashing

- Locality makes a limited resident set effective while a process stays within an execution phase.
- Thrashing occurs when useful pages are repeatedly evicted and quickly faulted back in.
- A combined active demand larger than available frames signals memory pressure and a risk of thrashing.
- Local replacement can limit interference, but it cannot create additional RAM.
- Reducing concurrent memory demand or increasing available capacity can restore progress.





Locality In A Memory-Reference Pattern

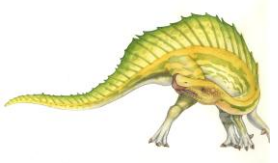






Working-Set Model

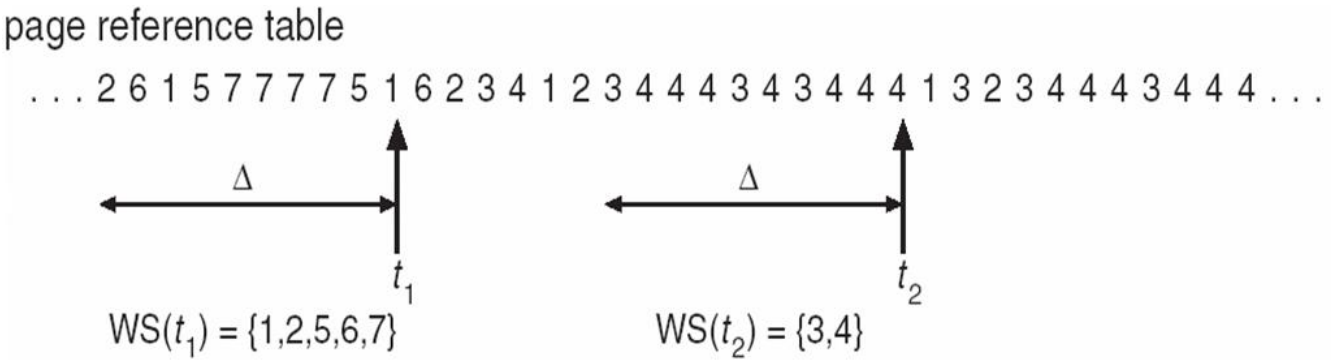
- Choose a window Δ measured consistently, for example the last 10,000 memory references.
- $WSS_i(t, \Delta)$ is the number of distinct pages process i referenced within that window.
- A small window can miss part of the current locality. A large window can combine several localities.
- If Δ includes the entire execution history, it includes all pages referenced so far, not every possible page in the program.
- $D = \Sigma\ WSS_i$ estimates total resident demand. Shared pages require care to avoid counting physical demand twice.



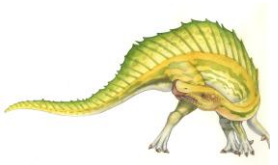


Working Set and Available Frames

- Let D be the sum of processes' working-set demands and m the available frames.
- If D exceeds m, processes cannot all keep their working sets resident.



- This memory pressure can cause thrashing.
- The OS may reduce multiprogramming, suspend a process, or otherwise reclaim enough memory.





Approximating a Working Set

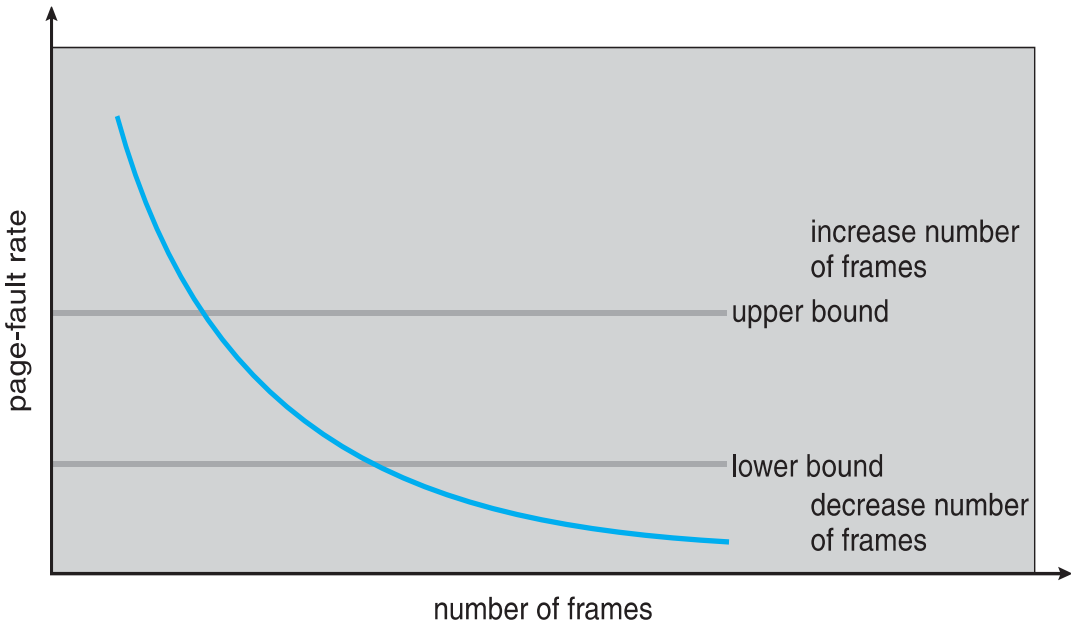
- A hardware reference bit records whether a page was accessed since the last reset.
- At regular sampling points, shift that bit into a per-page history and clear the hardware bit.
- The union of recent history bits approximates which pages were used within the window.
- For a 10,000-reference model window, two 5000-reference intervals give a coarse approximation; ten 1000-reference intervals give finer resolution.
- A real timer samples elapsed time, not an exact number of references. The sampling unit must match the approximation being described.





Page-Fault Frequency

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



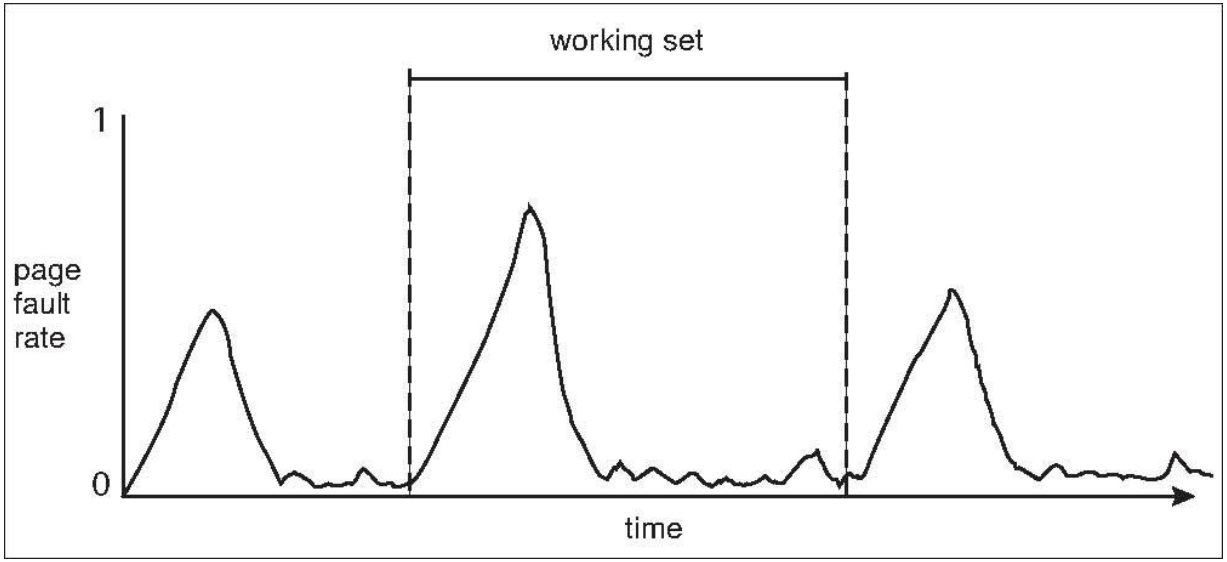


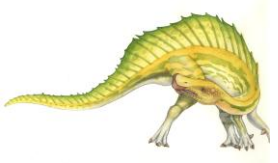
Troy University Resources



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

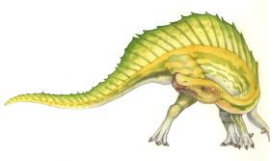






Memory-Mapped Files

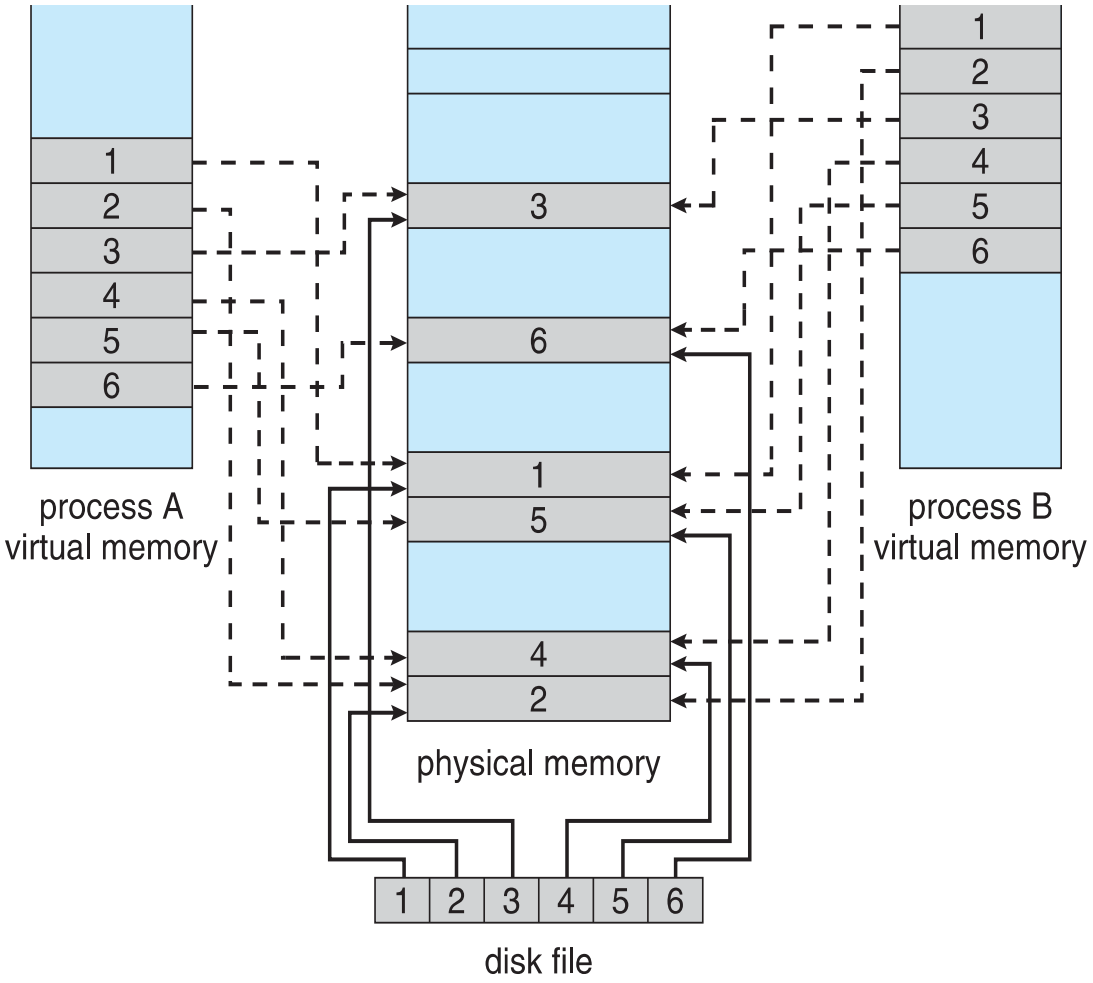
- A file mapping associates a virtual address range with part of a file.
- The program uses ordinary loads and stores to access the mapped bytes.
- On demand, the OS can map a resident page-cache page or load file contents into a frame.
- Several processes can map the same file and share physical pages.
- Mapping can avoid some explicit read/write copies, but it does not guarantee that every workload becomes faster.



Troy University Resources



Shared File Mappings







Shared and Private POSIX Mappings

mmap() creates a mapping. It does not load the entire file eagerly in the ordinary demand-paged case.

Mode	What writes do	What the program still needs
MAP_SHARED	Updates are shared with other mappings and can be written to the backing file.	A synchronization protocol; use msync when explicit mapped-data synchronization is needed.
MAP_PRIVATE	Creates private copy-on-write changes; those changes do not update the backing file.	A correct mapping length and protection settings.

Closing the file descriptor does not unmap the region. Use munmap() to remove the mapping.

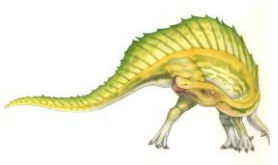


Troy University Resources



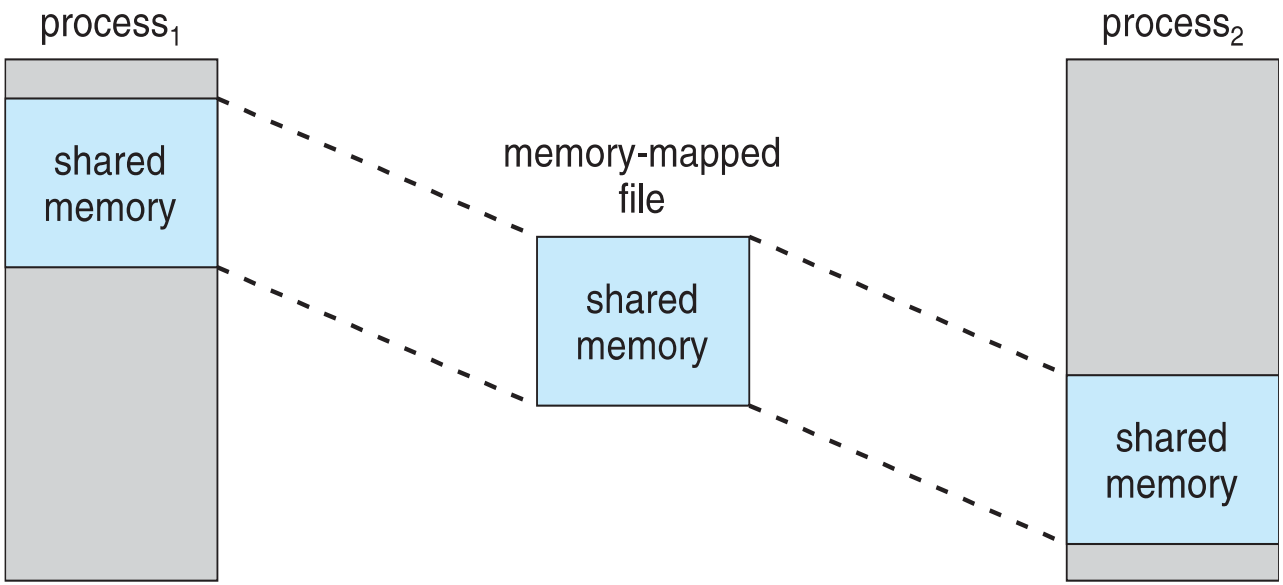
File I/O and the Page Cache

- On Linux, buffered read/write and ordinary file mappings use the page cache for file data.
- read() normally copies bytes into a user buffer. A file mapping lets the process access mapped cache pages.
- Dirty shared pages are written back according to OS policy. close() alone is not a general durability guarantee.
- msync(MS_SYNC) synchronizes modified mapped pages with their backing storage, subject to the platform's guarantees.
- File mappings differ from device MMIO, where addresses represent device registers or device memory.





Shared Memory Through File Mapping



Processes map the same backing object into their virtual address spaces. They synchronize access to shared data separately.

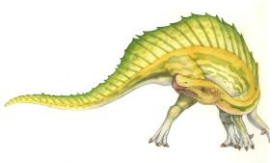


Troy University Resources



Windows File-Mapping API

- CreateFile() opens a file and returns a file handle.
- CreateFileMapping() creates or opens a mapping object. INVALID_HANDLE_VALUE can create a paging-file-backed shared-memory object.
- MapViewOfFile() maps a view into the calling process's virtual address space.
- Other processes can open the named mapping object and map their own views.
- Unmap views with UnmapViewOfFile() and close handles. Shared access still needs synchronization.





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

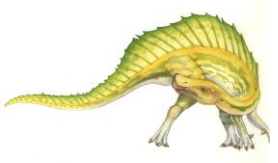


Troy University Resources



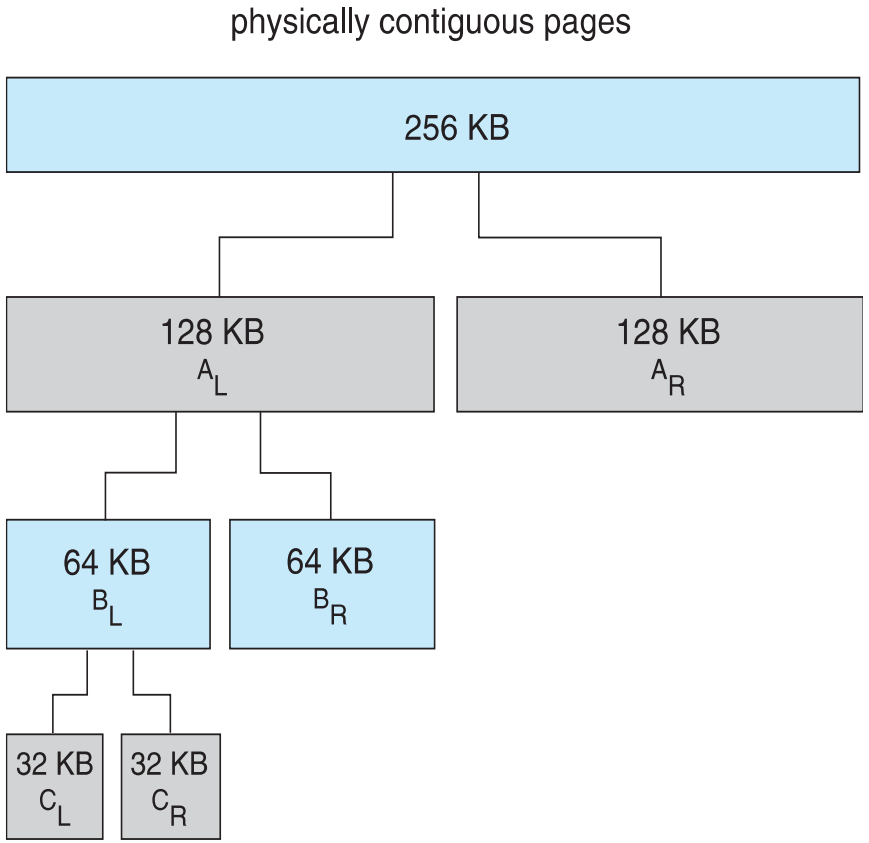
Buddy System

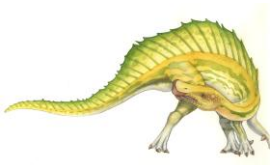
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk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





Buddy System Allocator





Troy University Resources



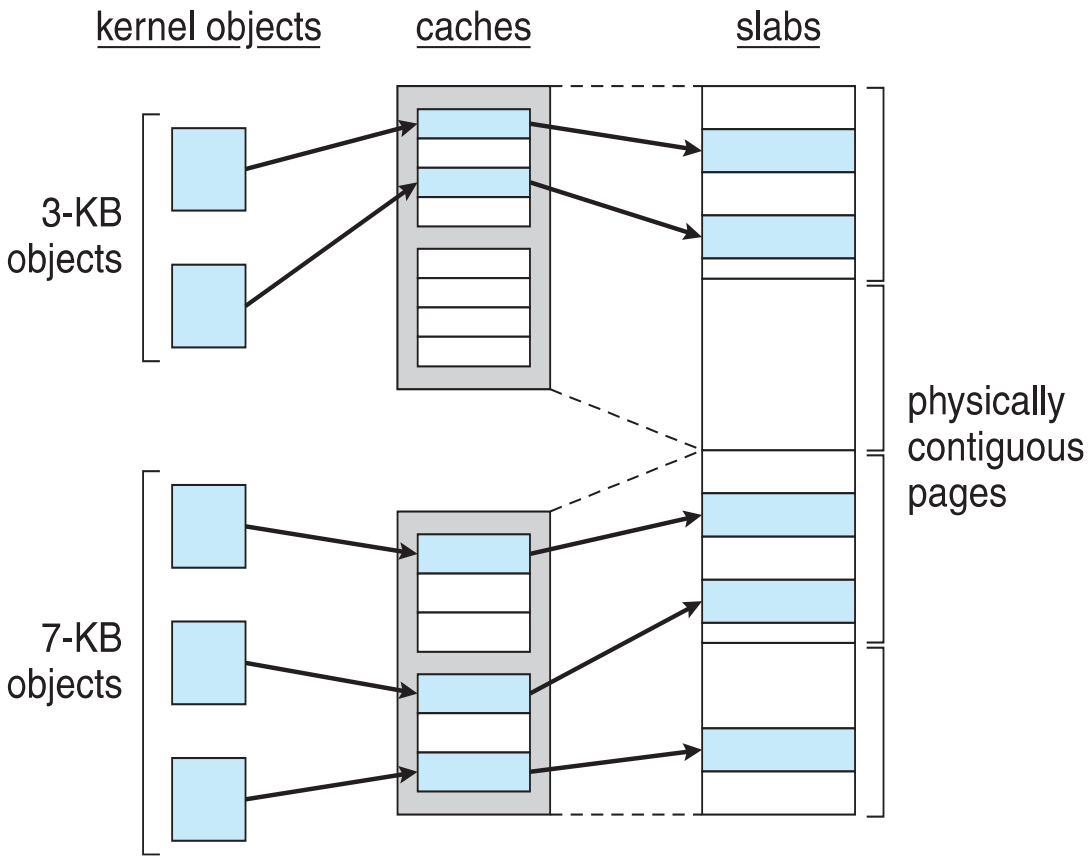
Slab Allocation

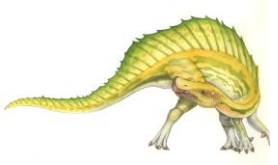
- A slab cache holds reusable objects of a particular size or type.
- One or more backing pages supply slots for those objects.
- Allocation can reuse a free slot instead of creating a new backing allocation for each object.
- Caching initialized objects and grouping similar sizes can improve speed and reduce some fragmentation.
- Slabs can still waste space through alignment, unused slots, and partially occupied slabs. They do not eliminate all fragmentation.





Slab Allocation







Linux Slab Caches

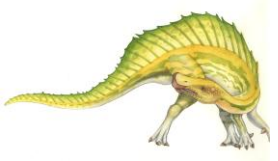
- Kernel structures such as task_struct can be allocated from dedicated caches.
- The size of task_struct depends on the architecture, configuration, and kernel version.
- A slab can have all slots used, some used, or no objects in use.
- The allocator reuses available objects and obtains more backing pages when necessary.
- CPU-local fast paths and shared-cache management help reduce allocation overhead and contention.





Linux SLUB

- SLUB provides Linux's slab-object allocation mechanism in current mainline kernels.
- It builds object caches over the page allocator and uses fast paths designed for frequent small allocations.
- The older SLAB and SLOB implementations belong to historical kernel versions.
- SLUB supports per-CPU allocation state, debugging, and object validation facilities.
- Configuration and workload determine the actual object sizes, overhead, and performance.

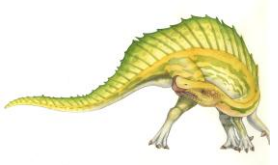


Troy University Resources



Other Considerations

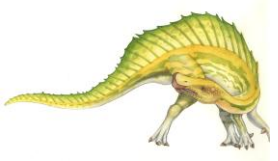
- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking





Prepaging

- Prepaging supplies pages before the program explicitly faults on them.
- It can reduce later fault latency or combine I/O, but unused pages waste memory and transfer work.
- If s pages are fetched and fraction α is used, then sα pages are useful and s(1 - α) are unused.
- A useful decision compares saved fault and I/O costs with the cost of fetching and retaining the pages.
- Read-ahead and application access hints are practical related mechanisms.

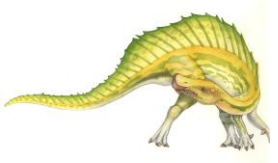


Troy University Resources



Page-Size Tradeoffs

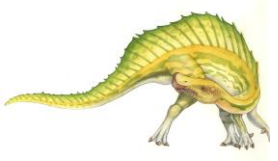
- Larger pages reduce the number of translation entries and can increase TLB reach.
- They can increase internal fragmentation and make allocation or compaction more difficult.
- Smaller pages provide finer protection and allocation granularity but require more translation metadata.
- Supported sizes come from the architecture and OS configuration. Examples include 4 KiB base pages and much larger huge pages.
- Page size affects locality, I/O behavior, and translation cost. The best choice depends on the workload.





TLB Reach

- For one page size, TLB reach ≈ number of entries × bytes per page.
- Example: 512 entries × 4 KiB = 2 MiB of translations.
- If active translations exceed effective TLB capacity, TLB misses and page-table walks can increase.
- A TLB miss does not mean the data page is absent from RAM.
- Larger mappings can improve reach, with tradeoffs in allocation and fragmentation. Mixed page sizes and set associativity complicate the simple estimate.





Program Structure and Page Locality

Assume 4-byte int, 512-byte pages, page-aligned data[128][128], one resident data frame, and no other faults. Each row occupies one page.

```
/* column first */
for (int j=0; j<128; j++)
  for (int i=0; i<128; i++)
    data[i][j] = 0;
```

```
/* row first */
for (int i=0; i<128; i++)
  for (int j=0; j<128; j++)
    data[i][j] = 0;
```

Column-first: 16,384 page faults.
Row-first: 128 page faults.

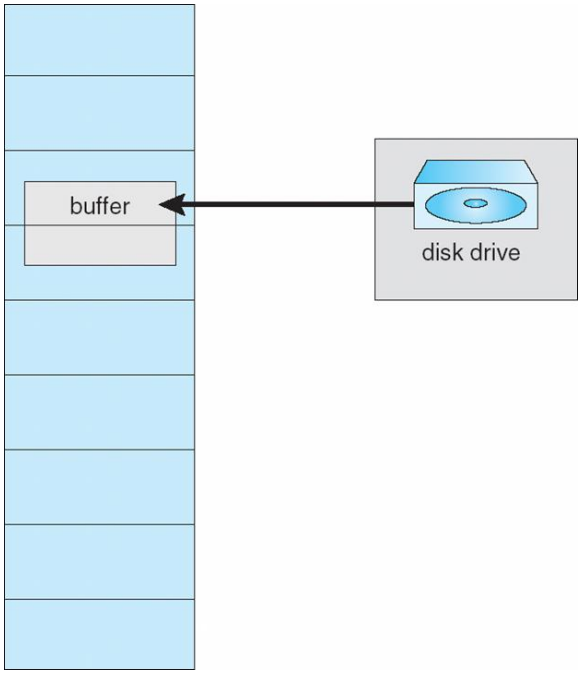
These counts depend on the stated frame limit. With enough frames to retain every row, the difference in page faults disappears.





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
- **Pinning** of pages to lock into memory



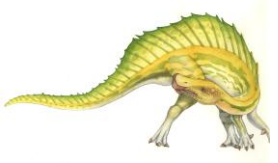


Troy University Resources



Operating-System Examples

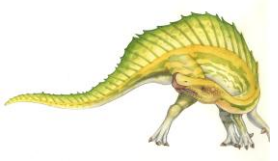
- Linux: page cache, anonymous memory, reclaim, swap, and optional multi-generation LRU.
- Windows: resident working sets, demand paging, and memory-pressure trimming.
- Solaris: historical page-scanner and free-memory threshold example.





Windows Working Sets

- A process's working set is its currently resident pageable virtual memory.
- A valid resident mapping can be used without a not-present fault. Other pages can fault back into the working set.
- Working-set minimum and maximum values influence policy and can be configured with different enforcement flags.
- They are not universal hard guarantees that every process always retains its minimum or never exceeds its maximum.
- The memory manager can trim working sets under pressure. A removed page can remain elsewhere in RAM and return through a soft fault.

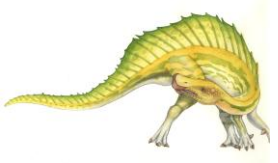


Troy University Resources



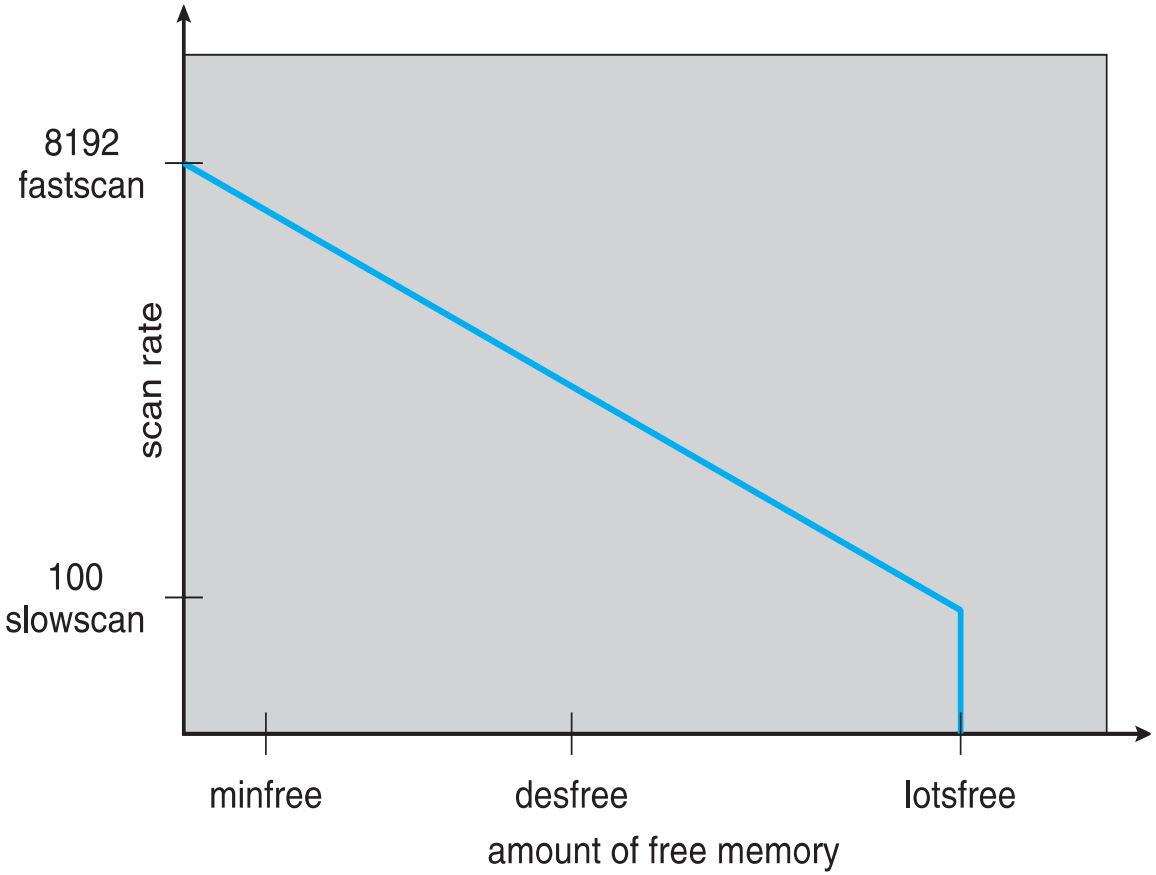
Historical Solaris Page Reclaim

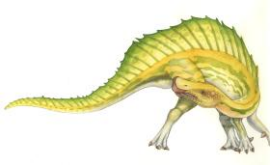
- A free-page list supplies frames to processes that fault on legal nonresident pages.
- The pageout process scans pages using a clock-based replacement policy.
- The textbook thresholds lotsfree, desfree, and minfree mark increasing memory pressure.
- The scan rate rises from slowscan toward fastscan as free memory declines.
- The illustrated priority-paging policy favors process code pages. These parameters describe the historical Solaris example.





Historical Solaris Page Scanner



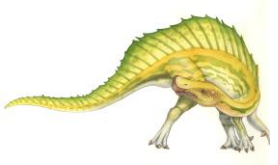




Performance of Demand Paging

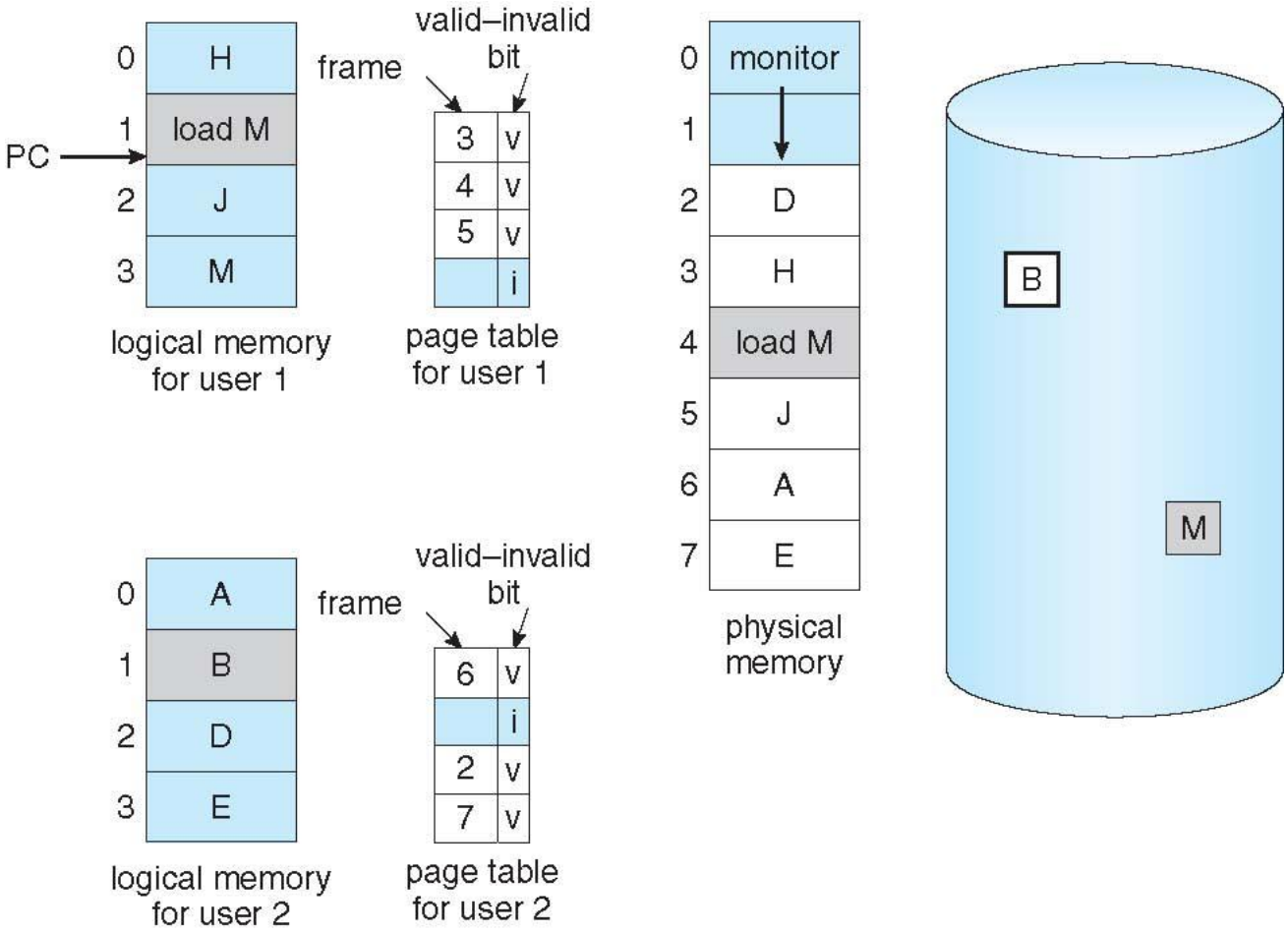
- Stages in Demand Paging (worse case)

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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
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





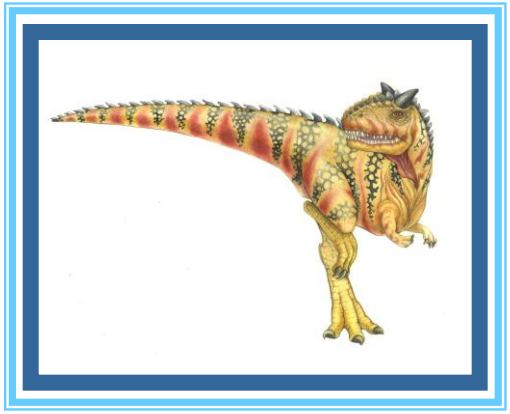
Need For Page Replacement





Troy University Resources

End of Chapter 10



Silberschatz, Galvin and Gagne ©2018