Operating Systems Notes - Chapter 10

July 27, 2025

Contents

1 10.1 Background

1.1 Introduction

- Virtual memory: technique allowing execution of processes not entirely in memory.
- Major advantage: programs larger than physical memory.
- Abstracts main memory into large, uniform storage array.
- Separates logical memory (programmer's view) from physical memory.
- Frees programmers from memory-storage limitations.
- Allows processes to share files, libraries, and implement shared memory.
- Efficient mechanism for process creation.
- Implementation complex, can decrease performance if used carelessly.

1.2 Things to learn

- 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.

1.3 Background

- Traditional memory management: entire process in physical memory for execution.
- Limitation: program size limited by physical memory.
- Real programs often don't need entire code:
  - Error handling code: seldom executed.
  - Arrays/lists: often allocated more memory than needed.
  - Rare program options/features: rarely used.
- Benefits of partial program execution in memory:
  - Programs not constrained by physical memory size; large **virtual** address space.
  - Less physical memory per program → more programs run concurrently → increased CPU utilization and throughput (no increase in response/turnaround time).
  - Less I/O for loading/swapping → faster program execution.
- **Virtual memory**: separation of logical memory (programmer's view) from physical memory.
- Allows large virtual memory with smaller physical memory.
- Simplifies programming: no worry about physical memory limits.

1.4 Virtual Address Space and Shared Memory

- **Virtual address space**: logical view of process storage in memory.
- Typically: process starts at logical address 0, contiguous memory.
- Physical memory: organized in page frames, not necessarily contiguous.
- Memory-management unit (MMU): maps logical pages to physical page frames.
- Heap grows upward, stack grows downward.
- Large blank space between heap/stack: part of virtual address space, requires physical pages only if heap/stack grows.
- **Sparse** address spaces: virtual address spaces with holes.
- Benefits of sparse address spaces:
  - Holes filled as stack/heap grow.
  - Dynamic linking of libraries/shared objects during execution.
- Virtual memory allows file/memory sharing via page sharing:
  - System libraries (e.g., standard C library) shared by mapping into virtual address space.
  - Libraries mapped read-only, physical pages shared by processes.
  - Processes share memory regions for communication.
  - Pages shared during process creation (`fork()`) → speeds up process creation.

Section glossary

Term	Definition
virtual memory	Technique allowing execution of a process not completely in memory; separates logical from physical memory.
virtual address space	Logical view of how a process is stored in memory.
sparse	Describes a page table with noncontiguous, scattered entries; an address space with many holes.

2 10.2 Demand paging

2.1 Introduction to Demand Paging

- Program loading:
  - Option 1: Load entire program into physical memory at execution.
  - Problem: May not need entire program initially (e.g., unselected options).
- Alternative: Load pages only as needed → **demand paging**.
- Pages loaded only when **demanded** during execution.
- Unaccessed pages never loaded into physical memory.
- Similar to paging with swapping.
- Benefit: More efficient memory use by loading only needed portions.

2.2 Basic Concepts of Demand Paging

- Load page into memory only when needed.
- Process execution: some pages in memory, some in secondary storage.
- Hardware support needed to distinguish: valid-invalid bit scheme.
- Valid bit: page legal and in memory.
- Invalid bit: page not valid (not in logical address space) or valid but in secondary storage.
- Page-table entry for non-memory-resident page marked invalid.
- Access to invalid page → **page fault**.
- Page fault causes trap to OS.
- Page fault handling procedure:
  1. Check internal table (process control block) for valid/invalid memory access.
  2. If invalid, terminate process. If valid but not in memory, page it in.
  3. Find a free frame.
  4. Schedule secondary storage operation to read page into new frame.
  5. Storage read complete: modify internal table and page table (page now in memory).
  6. Restart interrupted instruction; process accesses page as if always in memory.
- **Pure demand paging**: start process with no pages in memory; fault for pages as needed.
- Programs tend to have **locality of reference** → reasonable demand paging performance.
- Hardware for demand paging:
  - **Page table**: marks entries invalid (valid-invalid bit or protection bits).
  - **Secondary memory**: holds non-main-memory pages (swap device, **swap space**).
- Crucial requirement: ability to restart any instruction after page fault.
- Save process state (registers, condition code, instruction counter) on page fault.
- Restart process in exact same place/state, with desired page in memory.
- Difficulty: instructions modifying multiple locations (e.g., IBM System 360/370 MVC).
  - Solution 1: Microcode accesses both ends of blocks before modification; if fault, happens before modification.
  - Solution 2: Use temporary registers for overwritten values; restore old values on fault before trap.
- Paging should be transparent to process.

2.3 Free-frame List

- OS maintains **free-frame list**: pool of free frames for page faults.
- Free frames also allocated for stack/heap segment expansion.
- Most OS use **zero-fill-on-demand**: frames "zeroed-out" before allocation (security).
- System startup: all available memory on free-frame list.
- Free-frame list shrinks with requests; must be repopulated when low.

2.4 Performance of Demand Paging

- Demand paging significantly affects performance.
- **Effective access time** for demand-paged memory:
  - Memory-access time (ma): 10 nanoseconds.
  - No page faults: effective access time = memory access time.
  - Page fault: read page from secondary storage, then access word.
- Let p = probability of page fault ($0 \leq p \leq 1$).
- Effective access time = $(1 - p) \times ma + p \times$ page fault time.
- Page fault service time components:
  1. Service page-fault interrupt.
  2. Read in the page.
  3. Restart the process.
- First and third tasks: 1 to 100 microseconds.
- HDD page-switch time: 8 milliseconds (3ms latency, 5ms seek, 0.05ms transfer).
- Total paging time: 8 milliseconds (hardware + software).
- Add queuing time if device busy.
- Example: $ma = 200$ ns, page-fault service time = 8 ms.
  - Effective access time (ns) = $(1 - p) \times 200 + p \times 8,000,000$
  - $= 200 + 7,999,800 \times p$.
- Effective access time directly proportional to **page-fault rate**.
- If $p = 1/1000$, effective access time = 8.2 microseconds (40x slowdown).
- To keep slowdown ¡ 10% (e.g., 220 ns effective access time):
  - $220 > 200 + 7,999,800 \times p$
  - $20 > 7,999,800 \times p$
  - $p < 0.0000025$ (fewer than 1 fault per 399,990 accesses).
- Low page-fault rate crucial for demand-paging performance.
- Swap space I/O generally faster than file system I/O (larger blocks, no file lookups).
- Swap space usage strategies:
  - Copy entire file image to swap space at startup, then demand-page from swap space (disadvantage: initial copy).
  - Demand-page from file system initially, write pages to swap space when replaced (Linux, Windows).
  - Demand-page binary executables directly from file system; overwrite frames when replaced (never modified); file system acts as backing store (Linux, BSD UNIX).
  - **Anonymous memory** (stack, heap) still uses swap space.
- Mobile OS (e.g., iOS) typically no swapping: demand-page from file system, reclaim read-only pages if memory constrained. Anonymous memory pages not reclaimed unless app terminated/releases memory.
- Compressed memory is an alternative to swapping in mobile systems.

Section glossary

Term	Definition
demand paging	Bringing in pages from storage as needed rather than entirely at process load time.
page fault	Fault from reference to a non-memory-resident page.
pure demand paging	Demand paging where no page is brought into memory until referenced.
locality of reference	Tendency of processes to reference memory in patterns, not randomly.
swap space	Secondary storage backing-store space for paged-out memory.
free-frame list	Kernel-maintained list of available free physical memory frames.
zero-fill-on-demand	Writing zeros into a page before making it available to a process.
effective access time	Measured/calculated time to access something (e.g., memory).
page-fault rate	Measure of how often a page fault occurs per memory access attempt.
anonymous memory	Memory not associated with a file; stored in swap space if dirty and paged out.

3 10.3 Copy-on-write

3.1 Process Creation with Copy-on-write

- Process creation using `fork()` can bypass demand paging initially.
- Technique similar to page sharing for rapid process creation.
- Minimizes new pages allocated to child process.
- Traditionally, `fork()` copied parent's address space for child.
- Copying may be unnecessary if child immediately calls `exec()`.
- **Copy-on-write**: parent and child processes initially share same pages.
- Shared pages marked as copy-on-write.
- If either process writes to a shared page, a copy of the shared page is created.
- Example: child modifies stack page (copy-on-write) → OS gets free frame, copies page, maps to child's address space.
- Child modifies its copied page, not parent's.
- Only modified pages are copied; unmodified pages (e.g., executable code) can be shared.
- Common technique in Windows, Linux, macOS.

3.2 Virtual Memory Fork (`vfork()`)

- Variation of `fork()` in UNIX (Linux, macOS, BSD UNIX).
- Parent process suspended; child uses parent's address space.
- `vfork()` does not use copy-on-write.
- Child process changes to parent's address space are visible to parent upon resumption.
- Use with caution: child must not modify parent's address space.
- Intended for use when child calls `exec()` immediately after creation.
- Extremely efficient process creation (no page copying).
- Sometimes used to implement UNIX command-line shell interfaces.

Section glossary

Term	Definition
copy-on-write	Write causes data to be copied then modified; on shared page write, page copied, write to copy.
virtual memory fork	`vfork()` system call; child shares parent's address space for read/write, parent suspended.

4 10.4 Page replacement

4.1 Over-allocating Memory

- Demand paging saves I/O by loading only used pages.
- Can increase degree of multiprogramming by **over-allocating** memory.
- Example: 6 processes (10 pages each, use 5) on 40 frames → higher CPU utilization/throughput.
- Problem: Processes may suddenly need all pages (e.g., 60 frames needed, only 40 available).
- System memory also used for I/O buffers, increasing strain on memory-placement.
- Over-allocation manifests as page fault with no free frames.

4.2 Basic Page Replacement

- If no free frame, find one not in use and free it.
- Freeing a frame: write contents to swap space, change page table (page no longer in memory).
- Use freed frame for faulted page.
- Modified page-fault service routine:
  1. Find desired page on secondary storage.
  2. Find a free frame:
     (a) If free frame, use it.
     (b) If no free frame, use page-replacement algorithm to select **victim frame**.
     (c) Write victim frame to secondary storage (if modified); update page/frame tables.
  3. Read desired page into newly freed frame; update page/frame tables.
  4. Continue process from page fault.
- No free frames → two page transfers (page-out, page-in) → doubles page-fault service time.
- Reduce overhead with **modify bit** (or **dirty bit**).
  - Hardware sets modify bit if page written to.
  - If modify bit set: write page to storage before replacement.
  - If modify bit not set: no need to write to storage (page unchanged).
  - Applies to read-only pages (discardable).
  - Significantly reduces page-fault service time if page not modified.
- Page replacement separates logical and physical memory.
- Enormous virtual memory on smaller physical memory.
- Two major problems for demand paging:
  - **Frame-allocation algorithm**: how many frames to allocate to each process.
  - **Page-replacement algorithm**: which frames to replace.
- Goal: lowest page-fault rate.
- Evaluate algorithms using **reference string** (trace of memory accesses).
- Reference string simplification: only page number, ignore immediate repeated references.
- More frames → fewer page faults (generally).

4.3 FIFO Page Replacement

- First-in, first-out (FIFO) algorithm.
- Replaces the oldest page (first one brought into memory).
- Can use a FIFO queue: replace head, insert new page at tail.
- Easy to understand and program.
- Performance not always good: may replace actively used pages.
- Bad replacement choice → increased page-fault rate, slowed execution (not incorrect).
- Suffers from **Belady's anomaly**: page-fault rate may **increase** as allocated frames increase.

4.4 Optimal Page Replacement

- **Optimal page-replacement algorithm** (OPT or MIN).
- Lowest page-fault rate, never suffers from Belady's anomaly.
- Rule: Replace the page that will not be used for the longest period of time.
- Guarantees lowest possible page-fault rate.
- Difficult to implement: requires future knowledge of reference string.
- Used mainly for comparison studies.

4.5 LRU Page Replacement

- **Least recently used (LRU) algorithm**: approximation of optimal.
- Replaces the page that **has not been used** for the longest period of time.
- Optimal algorithm looking backward in time.
- Does not suffer from Belady's anomaly (is a **stack algorithm**).
- Implementation requires substantial hardware assistance.
  - **Counters**: associate time-of-use field with page-table entry; CPU logical clock increments; copy clock to field on reference. Replace page with smallest time value.
  - **Stack**: keep stack of page numbers; on reference, remove page and put on top. Most recently used at top, least recently used at bottom. Best with doubly linked list.
- True LRU implementation is expensive due to per-memory-reference updates.

4.6 LRU-approximation Page Replacement

- Many systems lack hardware for true LRU.
- Use **reference bit**: hardware sets bit when page referenced.
- OS clears bits periodically. Can determine which pages used, but not order.
- Basis for LRU approximation algorithms.
- **Additional-reference-bits algorithm**:
  - Keep 8-bit byte for each page.
  - Timer interrupt (e.g., every 100ms): OS shifts reference bit into high-order bit of byte, shifts others right.
  - 8-bit shift registers show history of page use.
  - Page with lowest number (interpreted as unsigned int) is LRU.
  - Can replace all with smallest value or use FIFO among them.
- **Second-chance page-replacement algorithm** (**clock** algorithm):
  - Basic FIFO.
  - If selected page's reference bit is 0, replace it.
  - If reference bit is 1, give second chance: clear bit, reset arrival time to current time.
  - Page with second chance not replaced until others replaced/given second chances.
  - Implemented as circular queue with pointer.
  - Pointer advances, clearing reference bits, until 0-bit page found.
  - Degenerates to FIFO if all bits set.
- **Enhanced second-chance algorithm**:
  - Considers (reference bit, modify bit) as ordered pair.
  - Four classes:
    1. (0, 0): neither recently used nor modified (best to replace).
    2. (0, 1): not recently used but modified (needs write-out).
    3. (1, 0): recently used but clean (likely used again soon).
    4. (1, 1): recently used and modified (likely used again soon, needs write-out).
  - Replace first page in lowest nonempty class.
  - Prefers clean pages to reduce I/Os.

4.7 Counting-based Page Replacement

- Keep counter of references for each page.
- **Least frequently used (LFU)**: replace page with smallest count.
  - Problem: heavily used page initially, then unused, keeps high count.
  - Solution: shift counts right periodically (exponentially decaying average).
- **Most frequently used (MFU)**: replace page with smallest count (assumes just brought in).
- Neither LFU nor MFU common: expensive, don't approximate OPT well.

4.8 Page-buffering Algorithms

- Used in addition to replacement algorithms.
- Pool of free frames:
  - On page fault, desired page read into free frame from pool before victim written out.
  - Process restarts faster. Victim frame added to pool after write-out.
- List of modified pages:
  - When paging device idle, modified page written to secondary storage; modify bit reset.
  - Increases probability of clean page for replacement (no write-out needed).
- Pool of free frames remembering old page:
  - If old page needed before frame reused, can be reused directly from pool (no I/O).
  - Check free-frame pool first on page fault.
- UNIX uses this with second-chance algorithm.

4.9 Applications and Page Replacement

- Some applications (e.g., databases) perform worse with OS virtual memory buffering.
- Applications understand their memory/storage use better than general-purpose OS algorithms.
- Double buffering if OS and application both buffer I/O.
- Data warehouses: sequential reads, then computations/writes. LRU removes old pages, but older pages might be read again. MFU could be more efficient.
- Some OS allow special programs to use secondary storage as large sequential array of logical blocks → **raw disk**.
- **Raw I/O**: bypasses file-system services (demand paging, locking, prefetching, allocation, names, directories).
- Raw partitions efficient for specific apps, but most apps better with regular file-system services.

Section glossary

Term	Definition
over-allocating	Providing access to more resources than physically available; allocating more virtual memory than physical memory.
page replacement	Selection of a physical memory frame to be replaced when a new page is allocated.
victim frame	Frame selected by page-replacement algorithm to be replaced.
modify bit	MMU bit indicating a frame has been modified (must be saved before replacement).
dirty bit	MMU bit indicating a frame has been modified (must be saved before replacement).
frame-allocation algorithm	OS algorithm for allocating frames among all demands.
page-replacement algorithm	Algorithm choosing which victim frame will be replaced by a new data frame.
reference string	Trace of accesses to a resource; list of pages accessed over time.
Belady's anomaly	Page-fault rate may increase as allocated frames increase for some algorithms.
optimal page-replacement algorithm	Algorithm with lowest page-fault rate, never suffers Belady's anomaly.
least recently used (LRU)	Selects item used least recently; in memory, page not accessed in longest time.
stack algorithm	Class of page-replacement algorithms that do not suffer from Belady's anomaly.
reference bit	MMU bit indicating a page has been referenced.
second-chance page-replacement algorithm	FIFO algorithm; if reference bit set, clear bit and don't replace.
clock	Circular queue in second-chance algorithm containing possible victim frames.
least frequently used (LFU)	Selects item used least frequently; in virtual memory, page with lowest access count.
most frequently used (MFU)	Selects item used most frequently; in virtual memory, page with highest access count.
raw disk	Direct access to secondary storage as array of blocks, no file system.

5 Allocation of frames

- Allocation issue: how to allocate fixed free memory among processes?
- Example: 128 frames, OS takes 35, 93 for user process.
  - Pure demand paging: 93 frames on free-frame list.
  - Page faults get free frames.
  - List exhausted → page-replacement algorithm used.
  - Process terminates → frames back to free-frame list.
- Variations:
  - OS allocates buffer/table space from free-frame list (can be used for user paging when not in use).
  - Keep 3 free frames reserved: free frame for page fault, replacement selected during swap.
  - Basic strategy: user process allocated any free frame.

5.1 Minimum number of frames

- Constraints on frame allocation:
  - Cannot exceed total available frames (unless page sharing).
  - Must allocate at least a minimum number of frames.
- Minimum frames for performance:
  - Fewer frames → higher page-fault rate, slower execution.
  - Page fault before instruction complete → instruction restart.
  - Need enough frames for all pages an instruction can reference.
- Example:
  - Single memory-reference instruction → 1 frame for instruction, 1 for memory reference.
  - One-level indirect addressing → at least 3 frames per process.
- Minimum frames defined by computer architecture.
  - Example: move instruction straddling two frames, two indirect operands → 6 frames.
  - Intel architectures: register-to-register/memory only, limits minimum frames.
- Maximum frames: defined by available physical memory.

5.2 Allocation algorithms

- Easiest way to split m frames among n processes: equal share, m/n frames.
  - Example: 93 frames, 5 processes → 18 frames each, 3 leftover for buffer.
  - Called **equal allocation**.
- Alternative: processes need differing memory.
  - Example: 10KB student, 127KB database, 62 frames.
  - Equal allocation (31 each) wastes frames for student process.
- Solution: **proportional allocation**.
  - Allocate memory by process size.
  - p_i has virtual memory size s_i. Total size $S = \sum S_i$. Available frames m.
  - Allocate $a_i \approx (s_i/S) \times m$ frames to p_i.
  - a_i must be integer, greater than minimum frames, sum $\leq m$.
  - Example: 62 frames, 10 pages & 127 pages. Total 137 pages.
    - $(10/137) \times 62 \approx 4$ frames.
    - $(127/137) \times 62 \approx 57$ frames.
  - Processes share frames by "needs".
- Allocation varies with multiprogramming level.
  - Increased level → processes lose frames.
  - Decreased level → frames spread.
- High/low priority processes treated same in equal/proportional allocation.
  - Solution: proportional allocation based on process priority or size + priority.

5.3 Global versus local allocation

- Page replacement affects frame allocation.
- Two categories of page-replacement algorithms:
  - **Global replacement**: process selects from all frames (can take from others).
  - **Local replacement**: process selects only from its own allocated frames.
- Example: high-priority process takes frames from low-priority.
  - Local replacement: frames allocated to process don't change.
  - Global replacement: process can increase its frames by taking from others.
- Global replacement problem: process performance depends on other processes' paging behavior (external circumstances).
- Local replacement: performance depends only on its own paging behavior.
- Global replacement: generally greater system throughput, more common.

5.4 Major and minor page faults

- Page fault: page no valid mapping.
- Two types: **major** and **minor** faults (Windows: **hard** and **soft** faults).
- **Major page fault**: page referenced, not in memory.
  - Requires reading from backing store, updating page table.
  - Demand paging → initially high major faults.
- **Minor page faults**: process no logical mapping to page, but page in memory.
  - Reasons:
    - Shared library in memory, no mapping → update page table.
    - Page reclaimed to free-frame list, not zeroed/allocated → frame removed from list, reassigned.
  - Less time consuming than major faults.
- Linux command to observe: `ps -eo min_flt,maj_flt,cmd`.
- Observation: major faults low, minor faults high. Linux processes use shared libraries heavily.

5.5 Reclaiming pages

- Global page-replacement strategy:
  - Satisfy requests from free-frame list.
  - Trigger replacement when list below threshold, not at zero.
  - Ensures sufficient free memory.
- Strategy depicted in Figure 10.5.1:
  - Keep free memory above minimum threshold.
  - Below threshold → kernel routine (**reapers**) triggered.
  - Reclaims pages from all processes (excluding kernel). Uses page-replacement algorithms.
  - Reaches max threshold → reaper suspended. Resumes when free memory below min threshold.
  - Continuous process.
- Reaper routine typically uses LRU approximation.
- If unable to maintain free frames: reclaims more aggressively (e.g., pure FIFO).
- Linux: **out-of-memory** (**OOM**) **killer** terminates processes at very low free memory.
  - OOM score: higher score → higher termination likelihood.
  - Calculated by memory usage percentage.
  - View OOM scores: `/proc/<pid>/oom_score`.
- Reaper routines vary aggressiveness. Min/max thresholds configurable by system administrator.

5.6 Non-uniform memory access

- Virtual memory assumption: uniform memory access.
- Not true for **non-uniform memory access** (**NUMA**) systems (multiple CPUs).
  - CPU accesses local memory faster than remote.
  - NUMA systems slower than uniform access, but allow more CPUs, greater throughput/parallelism.
- NUMA performance: managing page frame location critical.
- NUMA-aware allocation: frames allocated "as close as possible" to CPU (minimum latency, same system board).
  - Page fault → NUMA-aware system allocates frame close to CPU.

- NUMA consideration: scheduler tracks last CPU.
  - Schedule process on previous CPU + allocate frames close to CPU → improved cache hits, decreased memory access.
- Threads complicate NUMA: process threads on different system boards. Memory allocation challenge.
- Linux solution:
  - Kernel identifies scheduling domains.
  - CFS scheduler prevents thread migration across domains (avoids memory access penalties).
  - Separate free-frame list per NUMA node → thread allocated memory from its running node.
- Solaris solution: **lgroups** (locality groups) in kernel.
  - Each lgroup: CPUs + memory, CPU accesses memory in group within defined latency.
  - Hierarchy of lgroups.
  - Solaris schedules threads/allocates memory within lgroup; if not possible, uses nearby lgroups.
  - Minimizes memory latency, maximizes CPU cache hit rates.

Section glossary

Term	Definition
equal allocation	Assigns equal amounts of a resource to all requestors; in virtual memory, equal frames to each process.
proportional allocation	Assigns a resource in proportion to some aspect of the requestor; in virtual memory, page frames in proportion to process size.
global replacement	Process selects replacement frame from all frames in system, even if allocated to another process.
local replacement	Process selects replacement frame only from its own allocated frames.
major fault	Page fault resolved by I/O to bring page from secondary storage.
minor fault	Page fault resolved without paging in data from secondary storage.
reapers	Routines that scan memory, freeing frames to maintain minimum free memory.
out-of-memory (OOM) killer	Linux routine that terminates processes to free memory when free memory is very low.
non-uniform memory access (NUMA)	Architecture where memory access time varies based on CPU core.
lgroups	Solaris locality groups in kernel; gather CPUs and memory for optimized access in NUMA

6 Thrashing

- Process without "enough" frames (minimum needed for working set) → quickly page-faults.
- Replaces page needed immediately → faults again and again.
- High paging activity called **thrashing**.
- Thrashing: spending more time paging than executing.
- Results in severe performance problems.

6.1 Cause of thrashing

- Scenario: OS monitors CPU utilization. Low CPU utilization → increase multiprogramming (new process).
- Global page-replacement algorithm used (replaces pages without regard to process).
- Process needs more frames → starts faulting, takes frames from other processes.
- Other processes fault → take frames from others.
- Faulting processes use paging device → ready queue empties.
- Processes wait for paging device → CPU utilization decreases.
- CPU scheduler sees decreasing CPU utilization → **increases** multiprogramming.
- New process takes frames → more page faults, longer paging device queue.
- CPU utilization drops further → CPU scheduler increases multiprogramming more.
- Thrashing occurs → system throughput plunges.
- Page-fault rate increases tremendously → effective memory-access time increases.
- No work done, processes spend all time paging.
- Illustrated in Figure 10.6.1: CPU utilization vs. degree of multiprogramming.
  - Multiprogramming increases → CPU utilization increases (slower) until max.
  - Further increase → thrashing, CPU utilization drops sharply.
  - To stop thrashing: **decrease** degree of multiprogramming.
- Limit thrashing effects: use **local replacement algorithm** (or **priority replacement algorithm**).
  - Local replacement: process selects only from its own frames.
  - Thrashing process cannot steal frames from others.
  - Problem not entirely solved: thrashing processes queue for paging device → increased average service time for page fault → increased effective access time for all processes.
- To prevent thrashing: provide process with enough frames.
- How many frames needed? Look at frames actually used → **locality model**.
- **Locality model**: process moves from locality to locality during execution.
  - Locality: set of pages actively used together.
  - Running program: several overlapping localities.
  - Example: function call → new locality (function instructions, local variables, global variables subset).
  - Exit function → leave locality.
  - Localities defined by program structure and data structures.
  - Principle behind caching: accesses are patterned, not random.
- Allocate enough frames for current locality: faults until pages in memory, then no faults until locality changes.
- Not enough frames for locality → thrashing.

6.2 Working-set model

- Based on locality assumption.
- Uses parameter Δ to define **working-set window**.
- Examine most recent Δ page references.
- Set of pages in most recent Δ references = **working set** (Figure 10.6.3).
- Page in active use → in working set.
- No longer used → drops from working set Δ time units after last reference.
- Working set: approximation of program's locality.
- Example (Figure 10.6.3): $\Delta = 10$ references.
  - Working set at t_1: {1, 2, 5, 6, 7}.
  - Working set at t_2: {3, 4}.
- Accuracy of working set depends on Δ selection.

  - Δ too small: won't encompass entire locality.
  - Δ too large: may overlap several localities.
  - Extreme: Δ infinite $\rightarrow$ working set is all pages touched during execution.
- Most important property: working-set size.
- Compute working-set size, WSS_i, for each process.
- Total demand for frames: $D = \sum WSS_i$.
- If $D > m$ (total available frames) $\rightarrow$ thrashing (some processes lack frames).
- Working-set model usage:
  - OS monitors working set of each process.
  - Allocates enough frames for its working-set size.
  - Enough extra frames $\rightarrow$ new process initiated.
  - Sum of working-set sizes exceeds available frames $\rightarrow$ OS suspends a process.
  - Suspended process's pages swapped out, frames reallocated. Restarted later.
- Prevents thrashing, keeps multiprogramming high, optimizes CPU utilization.
- Difficulty: tracking moving working-set window.
- Approximation: fixed-interval timer interrupt + reference bit.
  - Example: $\Delta = 10,000$ references, interrupt every 5,000 references.
  - Timer interrupt: copy and clear reference bits.
  - Page fault: examine current reference bit and two in-memory bits.
  - Used within last 10,000-15,000 references $\rightarrow$ at least one bit on $\rightarrow$ in working set.
  - Not entirely accurate (cannot tell exact reference time within interval).
  - Reduce uncertainty: increase history bits, interrupt frequency (higher cost).

6.3 Page-fault frequency

- Working-set model successful, useful for prepaging, but clumsy for thrashing control.
- **Page-fault frequency** (**PFF**) strategy: more direct.
- Problem: prevent thrashing (high page-fault rate).
- Control page-fault rate:
  - Too high $\rightarrow$ process needs more frames.
  - Too low $\rightarrow$ process may have too many frames.
- Establish upper and lower bounds on desired page-fault rate (Figure 10.6.4).
  - Actual PFF exceeds upper limit $\rightarrow$ allocate another frame.
  - Actual PFF falls below lower limit $\rightarrow$ remove a frame.
- Directly measure and control PFF to prevent thrashing.
- If PFF increases and no free frames: select process, swap out to backing store. Freed frames distributed to high-PFF processes.

6.4 Current practice

- Thrashing and swapping: high performance impact.
- Best practice: include enough physical memory to avoid thrashing/swapping.
- Provides best user experience (smartphones to large servers).

Section glossary

Term	Definition
thrashing	High rate of paging memory; occurs when insufficient physical memory to meet virtua memory demand.
local replacement algorithm	Page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.
priority replacement algorithm	Page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.
locality model	Model for page replacement based on the working-set strategy.
working-set model	Memory access model based on tracking the set of most recently accessed pages.
working-set window	Limited set of most recently accessed pages (a "window" view of the entire set of accessed pages).
working set	The set of pages in the most recent page references.
page-fault frequency	The frequency of page faults.

7 Memory compression

- Alternative to paging: **memory compression**.
- Compress several frames into a single frame.
- Reduces memory usage without swapping pages.
- Example (Figure 10.7.1):
  - Free-frame list below threshold → triggers page replacement.
  - Selects frames (e.g., 15, 3, 35, 26) to place on modified-frame list.
  - Instead of writing to swap space, compress frames (e.g., three) into single page frame.
- Example (Figure 10.7.2):
  - Frame 7 removed from free-frame list.
  - Frames 15, 3, 35 compressed and stored in frame 7.
  - Frame 7 stored in list of compressed frames.
  - Frames 15, 3, 35 moved to free-frame list.
  - If compressed frame referenced → page fault, decompressed, restoring original pages.
- Mobile systems (Android, iOS) generally don't support standard swapping/paging.
  - Memory compression integral to their memory-management strategy.
- Windows 10 and macOS support memory compression.
  - Windows 10: **Universal Windows Platform** (**UWP**) apps on mobile devices are candidates.
  - macOS (Version 10.9+): compresses LRU pages when free memory is short, then pages if needed.
  - Performance tests: memory compression faster than paging to SSD on macOS.
- Memory compression requires allocating free frames for compressed pages.
  - Significant memory saving possible (e.g., 3 frames to 1).
- Contention between compression speed and **compression ratio** (amount of reduction).
  - Higher compression ratios → slower, more computationally expensive algorithms.
  - Most algorithms balance factors: high ratios with fast algorithms.
  - Improved by parallel compression using multiple cores.
  - Examples: Microsoft's Xpress, Apple's WKdm → fast, compress to 30-50% original size.

Section glossary

Term	Definition
memory compression	Alternative to paging; compresses frame contents to decrease memory usage.
universal Windows platform (UWP)	Windows 10 architecture providing common app platform for all devices running it.
compression ratio	Measurement of compression effectiveness (ratio of compressed to uncompressed space).

8 Allocating kernel memory

- User-mode process requests memory → pages allocated from kernel's free page frame list.
  - List populated by page-replacement algorithms (e.g., Section 10.4).
  - Free pages scattered throughout physical memory.
  - Single byte request → entire page frame granted → internal fragmentation.
- Kernel memory often allocated from a different free-memory pool. Reasons:
  1. Kernel requests varying data structure sizes, some less than a page.
     - Must use memory conservatively, minimize fragmentation waste.
     - Many OS do not subject kernel code/data to paging.
  2. User-mode pages don't need contiguous physical memory.
     - Hardware devices interact directly with physical memory (no virtual memory interface).
     - May require physically contiguous pages.
- Strategies for managing kernel free memory: "buddy system" and "slab allocation".

8.1 Buddy system

- Allocates memory from fixed-size segment of physically contiguous pages.
- Uses a **power-of-2 allocator**: satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, etc.).
- Request not appropriately sized → rounded up to next highest power of 2.
  - Example: 11 KB request → satisfied with 16-KB segment.
- Example (Figure 10.8.1):
  - Initial segment: 256 KB. Kernel requests 21 KB.
  - Segment divided into two **buddies** (A_L and A_R), each 128 KB.
  - One buddy (A_L) divided into two 64-KB buddies (B_L and B_R).
  - Next-highest power of 2 for 21 KB is 32 KB.
  - One 64-KB buddy (B_L) divided into two 32-KB buddies (C_L and C_R).
  - One 32-KB buddy (C_L) used for 21-KB request.
- Advantage: quickly combine adjacent buddies to form larger segments using **coalescing**.
  - Example: kernel releases C_L unit → coalesce C_L and C_R into 64-KB segment (B_L).
  - B_L can coalesce with B_R to form 128-KB segment (A_L).
  - Can eventually form original 256-KB segment.
- Drawback: rounding up to next highest power of 2 causes internal fragmentation.
  - Example: 33-KB request → 64-KB segment allocated.
  - Cannot guarantee less than 50% of allocated unit wasted.

8.2 Slab allocation

- Second strategy for kernel memory allocation.
- A **slab**: one or more physically contiguous pages.
- A **cache**: one or more slabs.
- Single cache for each unique kernel data structure (e.g., process descriptors, file objects, semaphores).
- Each cache populated with **objects** (instantiations of kernel data structure).
- Relationship (Figure 10.8.2):
  - Two 3 KB kernel objects, three 7 KB objects, each in separate cache, linked to slab.
- Algorithm uses caches to store kernel objects.
  - Cache created → objects (initially `free`) allocated to cache.
  - Number of objects depends on slab size (e.g., 12-KB slab → six 2-KB objects).
  - New object needed → allocator assigns any `free` object from cache. Object marked `used`.
- Scenario: kernel requests memory for process descriptor (`struct task_struct`, 1.7 KB).
  - Cache fulfills request with pre-allocated, free `struct task_struct` object.
- Slab states in Linux:
  1. **Full**: All objects in slab `used`.
  2. **Empty**: All objects in slab `free`.
  3. **Partial**: Slab has both `used` and `free` objects.

- Slab allocator request satisfaction:
  - First: free object in a partial slab.
  - If none: free object from an empty slab.
  - If no empty slabs: new slab allocated from contiguous physical pages, assigned to cache; object memory allocated from new slab.
- Two main benefits of slab allocator:
  1. No memory wasted due to fragmentation.
     - Each kernel data structure has associated cache.
     - Caches made of slabs divided into object-sized chunks.
     - Kernel requests memory → exact amount returned.
  2. Memory requests satisfied quickly.
     - Effective for frequent object allocation/deallocation (common in kernel).
     - Objects created in advance → quickly allocated from cache.
     - Released objects marked free, returned to cache → immediately available.
- History:
  - First appeared in Solaris 2.4 kernel.
  - Now used for some user-mode requests in Solaris.
  - Linux (Version 2.2+): adopted slab allocator (referred to as SLAB).
- Recent Linux kernel memory allocators: SLOB and SLUB.
  - **SLOB allocator**: for systems with limited memory (e.g., embedded systems).
    - Maintains three lists: `small` (¡256 bytes), `medium` (¡1,024 bytes), `large` (other objects ¡ page size).
    - Allocates from appropriate list using first-fit policy.
  - **SLUB allocator**: default for Linux kernel (Version 2.6.24+), replaced SLAB.
    - Reduced SLAB overhead.
    - Stores metadata in `page` structure (not with each slab).
    - No per-CPU queues for objects (significant memory saving on multi-processor systems).
    - Better performance with more processors.

Section glossary

Term	Definition
power-of-2 allocator	Allocator in buddy system; satisfies memory requests in units sized as a power of 2.
buddies	Pairs of equal size in buddy memory allocation.
coalescing	Combining freed memory in adjacent buddies into larger segments.
slab allocation	Memory allocation method; slab split into object-sized chunks, eliminating fragmentation.
slab	Section of memory (one or more contiguous pages) used in slab allocation.
cache	Temporary data copy for performance; in slab allocator, consists of one or more slabs.
object	Instance of a class or data structure.

9 Other considerations

9.1 Prepaging

- Pure demand paging: large number of page faults when process starts (initial locality).
- **Prepaging**: attempt to prevent high initial paging.
- Strategy: bring some/all needed pages into memory at once.
- Example: working-set model, remember working set for suspended process.
- Resume process: automatically bring entire working set back before restarting.
- Advantage: cost of prepaging vs. cost of servicing page faults.
- Risk: many prepaged pages may not be used.
- Cost analysis:
  - s pages prepaged.
  - α fraction of s pages actually used ($0 \leq \alpha \leq 1$).
  - Question: cost of $s \cdot (1 - \alpha)$ unnecessary pages vs. $s \cdot \alpha$ saved page faults.
  - If $\alpha \approx 0$, prepaging loses.
  - If $\alpha \approx 1$, prepaging wins.
- Prepaging executable programs: difficult, unclear what pages to bring in.
- Prepaging files: more predictable, often accessed sequentially.
- Linux `readahead()` system call: prefetches file contents into memory.

9.2 Page size

- New machine design: decision on best page size.
- No single best page size; factors support various sizes.
- Page sizes: invariably powers of 2, typically 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.
- **Page table size**:
  - Decreasing page size increases number of pages, thus page table size.
  - Example: 4 MB virtual memory.
  - 4,096 pages of 1,024 bytes vs. 512 pages of 8,192 bytes.
  - Large page size desirable for page table size (each active process has own copy).
- **Memory utilization / Internal fragmentation**:
  - Better utilized with smaller pages.
  - Process likely won't end exactly on page boundary.
  - Part of final page allocated but unused (internal fragmentation).
  - Average waste: half of final page.
  - 256 bytes waste for 512-byte page; 4,096 bytes for 8,192-byte page.
  - Minimize internal fragmentation: small page size needed.
- **I/O time**:
  - Read/write page time: seek, latency, transfer times.
  - Transfer time: proportional to page size (argues for small page size).
  - Latency/seek time: dwarf transfer time (e.g., 3ms latency, 5ms seek vs. 0.01ms transfer for 512 bytes).
  - Doubling page size: minimal increase in total I/O time.
  - Reading 1,024 bytes: 8.02ms for one 1,024-byte page vs. 16.02ms for two 512-byte pages.
  - Minimize I/O time: argues for larger page size.
- **Locality / Resolution**:
  - Smaller page size: total I/O reduced, improved locality.
  - Each page matches program locality more accurately.
  - Better **resolution**: isolate only memory actually needed.
  - Larger page size: allocate/transfer needed + unneeded data in same page.
  - Smaller page size: less I/O, less total allocated memory.
- **Number of page faults**:
  - Page size of 1 byte: page fault for each byte.
  - 200 KB process, half used: 1 page fault (200 KB page) vs. 102,400 page faults (1 byte page).

  - Each page fault: large overhead (interrupt, saving registers, replacing page, queuing, updating tables).
  - Minimize page faults: large page size needed.
- Historical trend: toward larger page sizes, even for mobile systems.
- Modern systems: much larger page sizes (e.g., Linux huge pages).

9.3 TLB reach

- **Hit ratio** of TLB: percentage of virtual address translations resolved in TLB.
- Increase hit ratio: increase number of entries (expensive, power hungry associative memory).
- **TLB reach**: amount of memory accessible from TLB.
- Calculation: number of entries × page size.
- Ideally: working set for process stored in TLB.
- Insufficient TLB reach: process spends time resolving memory references in page table.
- Doubling TLB entries: doubles TLB reach.
- Another approach: increase page size or provide multiple page sizes.
- Increase page size (e.g., 4 KB to 16 KB): quadruples TLB reach.
- Downside of larger page size: increased fragmentation for some applications.
- Most architectures: support for multiple page sizes.
- Linux example: default 4 KB, also **huge pages** (e.g., 2 MB).
- ARM v8 architecture: support for different page/region sizes.
- ARM v8 TLB entry: **contiguous bit**.
- Contiguous bit set: entry maps contiguous (adjacent) blocks of memory.
- Three arrangements for contiguous blocks in single TLB entry:
  - 64-KB TLB entry: 16 × 4 KB adjacent blocks.
  - 1-GB TLB entry: 32 × 32 MB adjacent blocks.
  - 2-MB TLB entry: 32 × 64 KB adjacent blocks, or 128 × 16 KB adjacent blocks.
- Multiple page sizes: OS (not hardware) may manage TLB.
- TLB entry field: indicates page frame size or contiguous block.
- Software TLB management: performance cost, but offset by increased hit ratio and TLB reach.

9.4 Inverted page tables

- Purpose: reduce physical memory needed for virtual-to-physical address translations.
- Method: one entry per page of physical memory, indexed by <process-id, page-number>.
- Benefit: reduces physical memory for translation info.
- Downside: no longer contains complete info about logical address space of a process.
- Demand paging requires this info for page faults.
- Solution: external page table (one per process) kept.
- External page table: looks like traditional per-process page table, contains virtual page location.
- Utility of inverted page tables: external tables referenced only on page fault, don't need to be quickly available.
- External tables: themselves paged in/out as necessary.
- Special case: page fault may cause another page fault (paging in external page table).
- Requires careful kernel handling, delay in page-lookup processing.

9.5 Program structure

- Demand paging: designed to be transparent to user program.
- System performance improved if user/compiler aware of demand paging.
- Example: Initializing 128x128 array, 128-word pages.
- Row major order (`data[i][j]` with outer loop `j`, inner loop `i`):
  - Zeros one word in each page, then another word in each page.
  - If <128 frames allocated: 128 × 128 = 16,384 page faults.
- Column major order (`data[i][j]` with outer loop `i`, inner loop `j`):
  - Zeros all words on one page before next.
  - Reduces page faults to 128.
- Careful selection of data structures/programming structures: increase locality, lower page-fault rate, smaller working set.

- Good locality: stack (access always to top).
- Bad locality: hash table (scatters references).
- Locality of reference: one measure of data structure efficiency.
- Other factors: search speed, total memory references, total pages touched.
- Compiler and loader effects on paging:
  - Separating code and data, reentrant code: code pages read-only, never modified.
  - Clean pages: don't need to be paged out.
  - Loader: avoid placing routines across page boundaries (keep routine in one page).
  - Pack frequently calling routines into same page.
  - Variant of bin-packing problem: pack variable-sized load segments into fixed-sized pages, minimize interpage references.
  - Useful for large page sizes.

9.6 I/O interlock and page locking

- Demand paging: sometimes need to allow pages to be **locked** in memory.
- Situation: I/O to/from user (virtual) memory.
- I/O often by separate I/O processor (e.g., USB storage controller given bytes/buffer address).
- Problem scenario:
  - Process issues I/O, put in queue.
  - CPU given to other processes.
  - Other processes cause page faults, replace page with I/O buffer (global replacement).
  - Pages paged out.
  - I/O request advances, I/O occurs to specified address.
  - Frame now used for different page of another process.
- Solutions:
  - Never execute I/O to user memory: data copied between system memory and user memory.
  - I/O only between system memory and device.
  - Extra copying: potentially high overhead.
  - Allow pages to be locked into memory: **lock bit** associated with every frame.
  - Locked frame: cannot be selected for replacement.
  - To write block to disk: lock pages containing block into memory.
  - When I/O complete: pages unlocked.
- Lock bits used in various situations:
  - OS kernel: some/all locked into memory (many OS cannot tolerate kernel page fault).
  - User processes: may need to lock pages (**pinning**).
  - Database process: manage memory chunk, move blocks between secondary storage/memory.
  - Pinning: common, most OS have system call for application to request.
  - Feature abuse: could stress memory-management algorithms.
  - Application often requires special privileges for pinning.
- Lock bit for normal page replacement:
  - Low-priority process faults, page read into memory.
  - High-priority process faults, needs replacement.
  - Sees newly brought-in page (clean, not referenced/modified).
  - Looks like perfect replacement.
  - Policy decision: allow replacement or not.
  - Preventing replacement of newly brought-in page until used once: use lock bit.
  - Page selected for replacement: lock bit on.
  - Remains on until faulting process dispatched again.
- Danger of lock bit: may get turned on but never off (bug).
- Locked frame becomes unusable.
- Solaris: allows locking "hints", but can disregard if free-frame pool too small or process requests too many locked pages.

Term	Definition
prepaging	Bringing pages into memory before they are requested.
hit ratio	Percentage of times a cache provides a valid lookup (e.g., TLB effectiveness).
TLB reach	Amount of memory addressable by the translation look-aside buffer.
huge pages	Feature designating a region of physical memory for especially large pages.
contiguous bit	In ARM v8 CPUs, a TLB bit indicating mapping to contiguous memory blocks.
locked	Fixed in place; pages locked in memory to prevent paging out.
pinning	Locking pages into memory to prevent them from being paged out.

10 Operating-system examples

10.1 Linux

- Linux manages virtual memory using demand paging.
- Allocates pages from a list of free frames.
- Global page-replacement policy: similar to LRU-approximation clock algorithm (second-chance).
- Maintains two page lists:
  - `active_list`: pages considered in use.
  - `inactive_list`: pages not recently referenced, eligible for reclamation.
- Each page has an **accessed** bit (set when referenced).
- Page first allocated: accessed bit set, added to rear of `active_list`.
- Page in `active_list` referenced: accessed bit set, moves to rear of list.
- Periodically: accessed bits for pages in `active_list` reset.
- Least recently used page: at front of `active_list`, may migrate to rear of `inactive_list`.
- Page in `inactive_list` referenced: moves back to rear of `active_list`.
- Lists kept in relative balance.
- `active_list` grows larger than `inactive_list`: pages from front of `active_list` move to `inactive_list` (eligible for reclamation).
- Linux kernel: page-out daemon process `kswapd`.
- `kswapd` periodically awakens, checks free memory.
- If free memory falls below threshold: `kswapd` scans `inactive_list`, reclaims pages for free list.

10.2 Windows

- Windows 10: supports 32- and 64-bit systems (Intel, ARM).
- 32-bit systems: default 2 GB virtual address space (extendable to 3 GB), 4 GB physical memory.
- 64-bit systems: 128-TB virtual address space, up to 24 TB physical memory (Windows Server up to 128 TB).
- Implements: shared libraries, demand paging, copy-on-write, paging, memory compression.
- Virtual memory: demand paging with **clustering**.
- **Clustering**: handles page faults by bringing in faulting page + several immediately preceding/following pages.
- Cluster size varies by page type:
  - Data page: 3 pages (faulting + one before + one after).
  - Other page faults: 7 pages.
- Key component: working-set management.
- Process creation: assigned **working-set minimum** (50 pages) and **working-set maximum** (345 pages).
- **Working-set minimum**: minimum pages guaranteed in memory.
- **Working-set maximum**: maximum pages allowed if sufficient memory.
- **Hard working-set limits**: if configured, values may be ignored.
- Process can grow beyond maximum if memory available.
- Memory allocated to process can shrink below minimum during high demand.
- Page replacement: LRU-approximation clock algorithm (second-chance) with local and global policies.
- Virtual memory manager: maintains free page frames list with threshold.
- Page fault for process below working-set maximum: allocates page from free list.
- Process at working-set maximum, page fault, sufficient memory: allocated free page (grows beyond maximum).
- Insufficient free memory: kernel selects page from process's working set for replacement (local LRU policy).
- Free memory falls below threshold: global replacement tactic **automatic working-set trimming**.
- **Automatic working-set trimming**: evaluates pages allocated to processes.
- If process has more pages than working-set minimum: removes pages until sufficient memory or process reaches minimum.
- Larger, idle processes targeted before smaller, active processes.
- Trimming continues until sufficient free memory, even if below working-set minimum.
- Windows performs trimming on user-mode and system processes.

10.3 Solaris

- Thread incurs page fault: kernel assigns page from free list.
- Imperative: kernel keeps sufficient free memory.
- `lotsfree` parameter: threshold to begin paging (typically 1/64 physical memory).
- Kernel checks free memory vs. `lotsfree` four times per second.
- If free pages < `lotsfree`: **pageout** process starts.
- Pageout process: similar to second-chance algorithm, uses two hands.
- Pageout process steps:
  - Front hand: scans all pages, sets reference bit to 0.
  - Back hand: examines reference bit; if still 0, appends page to free list, writes to secondary storage if modified.
- Solaris manages minor page faults: process reclaims page from free list if accessed before reassigned.
- Pageout algorithm: uses parameters to control `scanrate` (pages per second).
- `scanrate` ranges from `slowscan` to `fastscan`.
- Free memory falls below `lotsfree`: scanning at `slowscan` (default 100 pages/sec).
- Progresses to `fastscan` (total physical pages/2, max 8,192 pages/sec) depending on free memory.
- Distance between hands: determined by `handspread` system parameter.
- Time between clearing and checking bit: depends on `scanrate` and `handspread`.
- Pageout process checks memory four times per second.
- If free memory falls below `desfree` (desired free memory): pageout runs 100 times per second.
- Goal: keep at least `desfree` memory available.
- If unable to maintain `desfree` for 30-second average: kernel swaps processes (freeing all pages).
- Kernel looks for idle processes.
- If system unable to maintain `minfree`: pageout process called for every new page request.
- Page-scanning algorithm skips shared library pages (even if eligible).
- Distinguishes between pages for processes and regular data files.
- Known as **priority paging**.

Term	Definition
clustering	Paging in a group of contiguous pages when a single page is requested via a page fault.
working-set minimum	Minimum number of frames guaranteed to a process in Windows.
working-set maximum	Maximum number of frames allowed to a process in Windows.
hard working-set limit	Maximum amount of physical memory a process is allowed to use in Windows.
automatic working-set trimming	In Windows, decreasing working-set frames for processes if minimum free memory threshold is reached.
priority paging	Prioritizing selection of victim frames based on criteria, e.g., avoiding shared library pages

11 Summary

- Virtual memory: abstracts physical memory into extremely large uniform array of storage.
- Benefits of virtual memory:
  - Program can be larger than physical memory.
  - Program does not need to be entirely in memory.
  - Processes can share memory.
  - Processes can be created more efficiently.
- **Demand paging**: pages loaded only when demanded during program execution.
- Pages never demanded are never loaded.
- **Page fault**: occurs when page not in memory is accessed.
- Page must be brought from backing store into available page frame.
- **Copy-on-write**: child process shares same address space as parent.
- If child or parent modifies page, copy of page is made.
- When available memory low: page-replacement algorithm selects existing page to replace.
- Page-replacement algorithms: FIFO, optimal, LRU.
- Pure LRU: impractical to implement; most systems use LRU-approximation algorithms.
- **Global page-replacement algorithms**: select page from any process for replacement.
- **Local page-replacement algorithms**: select page from faulting process.
- **Thrashing**: system spends more time paging than executing.
- **Locality**: set of pages actively used together.
- Process execution: moves from locality to locality.
- **Working set**: based on locality, set of pages currently in use by a process.
- **Memory compression**: compresses number of pages into single page.
- Alternative to paging, used on mobile systems without paging support.
- **Kernel memory**: allocated differently than user-mode processes.
- Allocated in contiguous chunks of varying sizes.
- Two common techniques for kernel memory allocation:
  - Buddy system.
  - Slab allocation.
- **TLB reach**: amount of memory accessible from TLB.
- Equal to number of entries in TLB $\times$ page size.
- Technique to increase TLB reach: increase page size.
- Linux, Windows, Solaris: manage virtual memory similarly.
- Use demand paging, copy-on-write, and variations of LRU approximation (clock algorithm).