Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 2.7: Virtualization VII

Swapping: Mechanisms and Policies

Opening Thought

> *"Thus far, we've assumed that every address space is small enough to fit entirely in physical memory. We will now relax this assumption and ask: what happens when the sum of all process address spaces exceeds the amount of physical memory available?"*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 21*

We have built a complete picture of paging — virtual-to-physical translation, TLBs, multi-level page tables. But all of this assumed physical RAM is large enough to hold every page of every running process simultaneously. Today we shatter that assumption and ask: what happens when memory runs out? The answer — swapping — is one of the most impactful mechanisms in all of operating systems design.

Lecture Roadmap

This lecture is organized into **9 major sections**:

1. **Motivation: Beyond Physical Memory** — Why swapping exists and what it provides
2. **Swap Space: The Mechanics** — Present bit, swap entries, page-in and page-out
3. **The Page Fault Handler: Full Mechanics** — Step-by-step OS response to a fault
4. **Page Replacement: The Core Problem** — OPT and why we cannot implement it
5. **Simple Replacement Policies** — FIFO, Belady's Anomaly, and why LRU is better
6. **LRU: Mechanics, Performance, and Implementation** — The ideal and its limits
7. **Approximating LRU: Clock and Its Variants** — Clock, NRU, Aging, WSClock
8. **Policy Considerations and Real Systems** — Thrashing, Linux kswapd, global replacement
9. **Worked Examples, Summary & Takeaways** — Exam preparation

> *Let's begin by understanding why swapping is necessary.*

Prerequisite Check

To follow this lecture, ensure you are comfortable with:

- **Page table entries (PTEs):** VPN→PFN mapping, valid bit, protection bits (Lecture 2.6).
- **Page faults:** what triggers them, hardware actions, OS handler invocation (Lecture 2.6).
- **TLB mechanics:** hit, miss, ASID, flush (Lecture 2.6).
- **Physical frame management:** free frame list, frame allocation (Lectures 2.5–2.6).
- **Disk I/O basics:** HDD/SSD latency orders of magnitude (Lecture 1.1).
- **CPU scheduling concepts:** blocking, ready, running states (Lecture 2.2).

If any of these feel uncertain, review the corresponding lectures before proceeding.

§ 1 — Motivation: Beyond Physical Memory

📎 *Extended material → Appendix A.1, A.2*

The Core Problem: Virtual Exceeds Physical

Modern systems run many processes simultaneously, each with a large virtual address space:

```
Example system:
  Physical RAM: 8 GB

  Running processes:
    Chrome browser:    2 GB virtual space (1.2 GB active)
    VS Code:           1 GB virtual space (600 MB active)
    JVM (Java app):    4 GB virtual space (2.5 GB active)
    Firefox:           2 GB virtual space (800 MB active)
    Background daemons: 500 MB total

  Sum of active pages: 1.2 + 0.6 + 2.5 + 0.8 + 0.5 = 5.6 GB
  Physical RAM: 8 GB → fits (barely).

  But if one more large process starts:
    Postgres database: 3 GB active → total active = 8.6 GB > 8 GB RAM.
    What does the OS do?
```

Without swapping: the OS must refuse to start new processes or kill existing ones. With swapping: the OS transparently moves infrequently-used pages to disk, freeing physical frames for the new process.

The OS Illusion: Every Process Believes It Has Unlimited RAM

With swapping, the OS maintains a fundamental illusion:

> ***Every process believes it has access to a large, private virtual address space that is always available — even if only a fraction of it is in physical RAM at any time.***

```
Process sees:
  Virtual address space: 0 to 2^48 (256 TB) — all accessible.
  Any access to any valid virtual address succeeds.

Reality:
  Only a few hundred MB of pages are actually in physical RAM.
  The rest are on disk (swap space), waiting to be loaded on demand.
  When the process accesses a swapped-out page:
    Hardware raises a page fault → OS loads the page → process continues.
  The process never knows the page was absent.
```

This is **demand paging** — pages are loaded into RAM only when they are actually needed.

Swap Space: A Reserved Area on Disk

Swap space is a dedicated region of disk storage that the OS uses to hold evicted pages:

```
Disk layout (Linux example):

Partition table:
  /dev/sda1:  /boot   (EXT4)  500 MB
  /dev/sda2:  /       (EXT4)  200 GB
  /dev/sda3:  [swap]          16 GB  ← dedicated swap partition

OR: a swap file within a filesystem:
  /swapfile   16 GB  ← file-backed swap (Linux fallocate + mkswap)

The OS formats swap space into fixed-size slots matching the page size (4 KB).
Each slot can hold exactly one evicted page.
The OS maintains a swap map: for each slot, which page it holds (or free).
```

The Swapping Contract

Swapping works through an implicit contract between hardware and OS:

```
Hardware responsibilities:
  1. Every memory access checks the PTE.
  2. If PTE.present = 0: raise Page Fault exception (do NOT access RAM).
  3. Record the faulting VA in a special register (e.g., CR2 on x86).
  4. Switch to kernel mode, jump to OS page fault handler.

OS responsibilities:
  1. On page fault: determine if the page is valid (in swap) or illegal.
  2. If valid: find a free physical frame (evicting another page if needed).
  3. Read the swapped-out page from disk into the free frame.
  4. Update PTE: set present=1, set PFN to the new frame.
  5. Return to the faulting instruction: hardware re-executes it → success.

Process view:
  The memory access appears to succeed (after a brief pause).
  The process is unaware that any swapping occurred.
```

✅ Checkpoint 1 — Motivation: Beyond Physical Memory

Test your understanding:

1. A system has 8 GB of physical RAM and is running processes whose total active memory is 10 GB. What two options does the OS have? What are the trade-offs of each?
2. What were "overlays" in early programming? What problem did they solve, and why were they unsatisfactory?
3. Explain the "OS illusion" provided by demand paging. What does the process believe? What is the reality?
4. NVMe SSD swap has a latency of ~100 µs; RAM has a latency of ~100 ns. A process accesses a swapped-out page. How much slower is this access compared to a RAM-resident page?
5. Explain one modern context (other than desktop OS) where swapping or demand paging is critical today.

Write your answers before moving on.

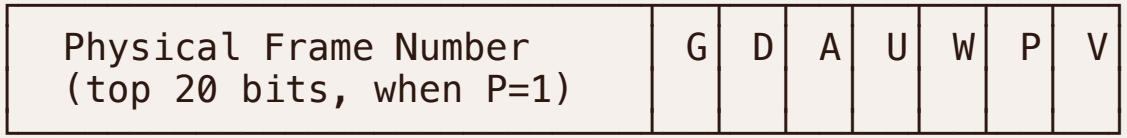
§ 2 — Swap Space: The Mechanics

📎 *Extended material → Appendix A.3*

The Present Bit in the PTE

To support swapping, the PTE needs one additional field beyond what we saw in Lecture 2.5:

```
Extended PTE (32-bit example):
```




```
  V = Valid     (1 = page was ever allocated; 0 = never mapped → SIGSEGV)
  P = Present   (1 = page is in physical RAM; 0 = page is on disk → swap fault)
  W = Writable
  U = User-accessible
  A = Accessed  (set by hardware on any access)
  D = Dirty     (set by hardware on any write)
  G = Global

When P=0 but V=1:  the page exists but is currently on disk.
  The PFN field is repurposed to hold the SWAP ENTRY (disk location).
```

Page-In: Bringing a Page From Disk Into RAM

Page-in (also called a **major page fault**) is the process of loading a page from swap into RAM:

```
Trigger: process accesses VA → PTE.present = 0 → Page Fault

OS page fault handler (major fault path):
  1. Read swap entry from PTE: (device=0, slot=1234)
  2. Find a free physical frame F (may require eviction — see § 4).
  3. Issue disk read: read 4 KB from swap device 0, slot 1234 → frame F.
  4. Block the process: move it to WAITING state while disk I/O completes.
  5. OS runs other processes while disk I/O is in flight.
  6. Disk I/O completes → OS wakes the process.
  7. Update PTE: PFN = F, present = 1, clear swap entry.
  8. Invalidate TLB entry for this VA (if any stale entry exists).
  9. Return to user mode → CPU re-executes the faulting instruction → hit!

Latency: 100 µs (NVMe SSD) to 10 ms (HDD) — visible pause to the user.
```

The Dirty Bit: Avoiding Unnecessary Disk Writes

The **dirty bit** (D bit) in the PTE is one of the most important optimizations in swapping:

```
Scenario 1: Code page (read-only)
  Process loads code from the ELF binary into frame F.
  Code page is never written → dirty bit = 0.
  When evicted: NO disk write needed.
    The original copy is still on disk (in the ELF file).
    On next access: just re-read from the ELF file (or swap).
  Eviction cost: 0 (no I/O needed).

Scenario 2: Heap page (read-write)
  Process writes data to a heap page in frame F.
  dirty bit = 1 (set automatically by hardware on first write).
  When evicted: MUST write to swap (dirty copy only exists in RAM).
  Eviction cost: one 4 KB disk write.

Optimization: always prefer evicting clean pages over dirty pages.
  Evicting a clean page is free; evicting a dirty page costs a disk write.
```

✅ Checkpoint 2 — Swap Space: The Mechanics

Test your understanding:

1. What is the **present bit** in a PTE? How does it differ from the **valid bit**? Give a scenario where valid=1 but present=0.
2. When a page is evicted, the PFN field of its PTE is repurposed. What does it store instead? What information does a **swap entry** encode?
3. A code page (loaded from the ELF binary, never written) is selected for eviction. Does the OS need to write it to swap? Explain using the dirty bit.
4. Describe the role of **kswapd** in Linux. What are the three watermarks it monitors? What happens when free pages fall below `pages_min`?
5. A system has 16 GB of RAM and uses an NVMe SSD for swap. The OS needs to evict 100 dirty pages (4 KB each). Estimate the total eviction time at 500 MB/s SSD write speed.

§ 3 — The Page Fault Handler: Full Mechanics

📎 *Extended material → Appendix A.4*

Recap: What Triggers a Page Fault?

A page fault is a CPU exception raised by the MMU when a memory access cannot be completed normally:

```
Causes of page faults:

1. Page not present (swapped out):
   PTE.present = 0, PTE.valid = 1
   → Major fault: page must be loaded from swap (disk I/O required).

2. Page never allocated (illegal access):
   PTE.valid = 0  (or address not in any VMA)
   → OS sends SIGSEGV → process killed.

3. Protection violation:
   Access type not permitted by PTE flags (write to read-only, execute data page)
   → OS sends SIGSEGV.

4. Copy-on-write fault:
   Write to a read-only page that was shared via fork() COW
   → OS allocates a new frame, copies the page, updates PTE → retry.

5. Minor fault (page in RAM but not mapped in this process's PTE):
   Anonymous page first access, mmap'd file page already in page cache
   → Just update PTE, no disk I/O needed.
```

Minor vs. Major Page Faults

An important distinction with very different performance implications:

	Minor Fault	Major Fault
Page location	Already in physical RAM	On disk (swap or file)
Disk I/O required	No	Yes
Latency	~1–10 µs (just PTE update)	100 µs – 10 ms
Cause	First access to anonymous page; shared page already in page cache	Page was swapped out; first access to memory-mapped file not yet loaded
Example	First write to a new `malloc`'d page	Accessing a page evicted by kswapd

Minor faults are cheap; major faults are expensive. The OS tries to minimize major faults through good replacement policies.

OS Page Fault Handler: Step-by-Step Flow

```
Page Fault Handler (simplified):

Input: faulting_VA (from CR2), error_code (R/W, user/kernel, P/NP)

Step 1: Validate the address
  Find the VMA (Virtual Memory Area) containing faulting_VA.
  If no VMA contains faulting_VA: → SIGSEGV (invalid address).

Step 2: Check permissions
  If error_code says WRITE but VMA is read-only: → SIGSEGV (protection fault).
  If error_code says EXECUTE but VMA is non-executable: → SIGSEGV.

Step 3: Determine fault type
  If PTE.present=0 and PTE is a swap entry: → Major fault (page on disk).
  If PTE.present=0 and page never allocated: → Minor fault (allocate new frame).
  If PTE.present=1 but PTE.write=0 and write attempt: → COW fault.

Step 4: Handle based on type (next slides).

Step 5: Return from exception → CPU re-executes faulting instruction.
```

Performance Impact: Major Page Fault Latency

The latency of a major page fault dominates all other costs:

```
Breakdown of major page fault handling time:

  Hardware trap + mode switch:      ~0.5 µs
  OS VMA lookup + validation:       ~1 µs
  Victim selection (replacement):   ~1–5 µs
  Dirty page write to disk (if any):
    NVMe SSD:   ~100 µs  (4 KB write)
    SATA SSD:   ~500 µs
    HDD:        ~5–10 ms
  Clean/swap page read from disk:
    NVMe SSD:   ~100 µs  (4 KB read)
    HDD:        ~5–10 ms
  PTE update + TLB invalidation:    ~1 µs
  Process wakeup + context switch:  ~5 µs

Total major fault cost:
  NVMe SSD:  ~200 µs (2,000× slower than a RAM access at 100 ns)
  HDD:       ~10 ms  (100,000× slower than a RAM access)
```

A program that incurs frequent major page faults can run **100,000× slower** than one that fits entirely in RAM. This is why replacement policy quality matters enormously.

✅ Checkpoint 3 — The Page Fault Handler

Test your understanding:

1. Distinguish a **minor page fault** from a **major page fault**. Give one concrete example of each.
2. What hardware register on x86-64 records the faulting virtual address? What information does the error code provide?
3. Trace the complete page fault handler execution for a major fault on a dirty victim page. List every step, including the disk I/O path.
4. A program running on an HDD-backed swap system incurs 100 major page faults per second. Each fault takes 10 ms. What fraction of CPU time is spent waiting for page faults?
5. What is a VMA (Virtual Memory Area)? How does the OS use it in the page fault handler to decide between SIGSEGV and legitimate fault handling?

§ 4 — Page Replacement: The Core Problem

📎 *Extended material → Appendix A.5*

The Replacement Decision

When physical memory is full and a new page must be brought in, the OS must answer:

> ***Which page currently in physical memory should be evicted to make room?***

This is the **page replacement problem** — one of the most studied problems in OS design. The choice directly determines how many page faults a process will experience.

```
Bad choice: evict a page that will be accessed again very soon.
  → The process immediately page-faults again.
  → Two disk I/Os wasted (evict + reload) for zero net progress.
  → In the worst case: thrashing (§ 8).

Good choice: evict a page that will not be needed for a long time.
  → The new page runs undisturbed for many accesses.
  → Disk I/O cost is amortized over many hits.

Goal: minimize the total number of page faults for a given workload.
```

The Optimal Policy (OPT / MIN)

The **Optimal replacement policy** (also called MIN or Bélády's Optimal Algorithm) is:

> ***Always evict the page that will not be used for the longest time in the future.***

```
Intuition:
  If we know the future reference string, we can make perfect decisions.
  The page we will not need for the longest time is the least valuable to keep.
  Evicting it minimizes the probability that the next access causes a fault.

Example (3 frames, reference: 7 0 1 2 0 3 0 4 2 3):

  Ref: 7  0  1  2  0  3  0  4  2  3
  F1:  7  7  7  2  2  2  2  2  2  2
  F2:  -  0  0  0  0  0  0  4  4  4
  F3:  -  -  1  1  1  3  3  3  3  3
  Flt: ✗  ✗  ✗  ✗  -  ✗  -  ✗  -  -
       (6 faults)

OPT evicts page 7 at step 4 (7 is next needed furthest in the future).
```

✅ Checkpoint 4 — Page Replacement: The Core Problem

Test your understanding:

1. What is a **reference string**? Why do we use it to evaluate replacement policies?
2. State the OPT replacement policy precisely. Why does it minimize page faults by definition?
3. Why is OPT unimplementable? What would a computer need to know to implement it?
4. Given reference string `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5` with 3 frames, compute OPT faults. Show which page is evicted at each step and why.
5. Explain how **temporal locality** provides the justification for LRU as an approximation of OPT.

Trace the OPT example fully before moving to the next section.

§ 5 — Simple Replacement Policies

📎 *Extended material → Appendix A.6*

FIFO: First-In, First-Out

The simplest replacement policy: evict the page that has been in memory the **longest**:

```
FIFO maintains a queue of pages in order of their arrival.
On a fault: evict the page at the head of the queue (oldest arrival).
           insert the new page at the tail.

Intuition: "the oldest guest should leave first."
Reality: the oldest page may still be heavily used (e.g., a shared library).
         FIFO ignores recency of use entirely.

Example (3 frames, reference: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3):

  Ref:   7    0     1       2     0     3     0     4     2     3
  Queue: [7]  [70]  [701]   [012] [012] [123] [230] [304] [042] [423]
          ↑ arrived first   ↑ evict 7
  Fault:  ✗    ✗    ✗        ✗     –     ✗     ✗    ✗      ✗     ✗
  Total:  9 faults
```

FIFO Trace: Full Reference String

Let's trace FIFO on the full reference string with 3 frames:

```
Reference: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
Frames: 3

Step  Ref  Frames (F1,F2,F3)   Queue (oldest→newest)   Fault?
  1    7   [7, –, –]           [7]                      ✗
  2    0   [7, 0, –]           [7, 0]                   ✗
  3    1   [7, 0, 1]           [7, 0, 1]                ✗
  4    2   [2, 0, 1]           [0, 1, 2]  (evict 7)     ✗
  5    0   [2, 0, 1]           [0, 1, 2]                –
  6    3   [2, 3, 1]           [1, 2, 3]  (evict 0)     ✗
  7    0   [2, 3, 0]           [2, 3, 0]  (evict 1)     ✗
  8    4   [4, 3, 0]           [3, 0, 4]  (evict 2)     ✗
  9    2   [4, 2, 0]           [0, 4, 2]  (evict 3)     ✗
 10    3   [4, 2, 3]           [4, 2, 3]  (evict 0)     ✗
 ...continuing: total = 15 faults
```

FIFO: 15 faults vs. OPT: 9 faults on the same workload.

Belady's Anomaly: More Frames → More Faults!

A counterintuitive property of FIFO — increasing the number of frames can **increase** the number of page faults:

> ***Belady's Anomaly:*** *For the FIFO replacement policy, there exist reference strings and frame counts where increasing the number of frames leads to more page faults.*

```
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

With 3 frames:  FIFO faults = 9
With 4 frames:  FIFO faults = 10  ← MORE faults with MORE frames!

Intuition: adding a frame changes which pages FIFO keeps,
           potentially evicting a page that was "protecting" a hot page.
           The queue ordering creates unexpected interactions.

This is alarming behavior for a production OS: giving a process more physical memory could make it run SLOWER.
```

LRU: The Right Intuition

Least Recently Used (LRU) replacement directly exploits temporal locality:

> ***LRU: Evict the page whose most recent access was the longest time ago.***

```
Key insight:
  "The best predictor of future memory accesses is recent past behavior."
  A page accessed at time t=99 is far more likely to be accessed at t=100
  than a page last accessed at t=1.

Relationship to OPT:
  OPT: uses future access times → evict page needed furthest in the future.
  LRU: uses past access times  → evict page unused for the longest past time.

  If the workload has strong temporal locality, the past is a good predictor
  of the future, and LRU ≈ OPT.

LRU is the most widely used replacement policy in practice because:
  1. It approximates OPT well for common workloads.
  2. It does not exhibit Belady's Anomaly (it has the "stack property").
  3. It can be approximated efficiently in hardware (accessed bit).
```

✅ Checkpoint 5 — Simple Replacement Policies

Test your understanding:

1. Trace FIFO with 3 frames on reference string `0, 1, 2, 0, 1, 3, 0, 1, 2, 3`. Count total page faults.
2. What is Belady's Anomaly? Which replacement policies can exhibit it? Which cannot?
3. Why does random replacement never exhibit Belady's Anomaly? (Hint: think about what "more frames" means probabilistically.)
4. Explain why FIFO is a poor policy for a frequently-accessed shared library page loaded at program startup.
5. State the key intuition behind LRU. How does it relate to the principle of temporal locality?

§ 6 — LRU: Mechanics, Performance, and Implementation

📎 *Extended material → Appendix A.7*

LRU Definition: Always Evict the Least Recently Used

Least Recently Used (LRU) replaces the page whose **most recent use** was the **furthest in the past**:

```
LRU maintains a total ordering of all loaded pages by time of last access.
  Most recently used (MRU) → highest value → keep.
  Least recently used (LRU) → lowest value → evict first.

On each memory access to page P:
  → Move P to the MRU position (it was just used, so it is now the newest).

On page fault (need a victim):
  → Evict the page at the LRU position (oldest last-access time).
  → Load the new page at the MRU position.

Key difference from FIFO:
  FIFO orders by ARRIVAL time.
  LRU orders by MOST RECENT ACCESS time.
  These are very different: a page loaded long ago but accessed just now
  is MRU under LRU (not evicted) but soon-to-be-evicted under FIFO.
```

LRU Trace Example: Step-by-Step

```
Reference: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3
Frames: 3  (LRU ordering shown as [MRU → LRU])

Step  Ref  Frames (set)   LRU order [newest→oldest]   Fault?
  1    7   {7}            [7]                          ✗
  2    0   {7,0}          [0,7]                        ✗
  3    1   {7,0,1}        [1,0,7]                      ✗
  4    2   {0,1,2}        [2,1,0]  (evict 7, LRU)      ✗
  5    0   {0,1,2}        [0,2,1]  (0 moved to MRU)    –
  6    3   {0,2,3}        [3,0,2]  (evict 1, LRU)      ✗
  7    0   {0,2,3}        [0,3,2]  (0 moved to MRU)    –
  8    4   {0,3,4}        [4,0,3]  (evict 2, LRU)      ✗
  9    2   {0,2,4}        [2,4,0]  (evict 3, LRU)      ✗
 10    3   {2,3,4}        [3,2,4]  (evict 0, LRU)      ✗
Total: 8 faults  (vs. OPT: 6, FIFO: 9 on this shorter string)
```

The Implementation Problem: Perfect LRU Requires Timestamps

To implement perfect LRU, we need to know **the exact time of the most recent access to every page**:

Approach 1: Hardware timestamp per frame

On every memory access: record the current clock tick in a per-frame register.
On eviction: scan all frames for the minimum timestamp → evict that frame.

Problems:

- Requires a timestamp register (64-bit counter) per physical frame.
- A 4 GB machine with 4 KB pages has 1 million frames → 8 MB of timestamp storage.
- Scanning 1 million frames on every fault → O(n) cost per replacement.
- Modern CPUs make billions of memory accesses per second → register update overhead.

Verdict: Hardware timestamp per frame is prohibitively expensive.

✅ Checkpoint 6 — LRU: Mechanics and Implementation

Test your understanding:

1. Trace LRU with 3 frames on reference string `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`. Count total page faults and compare to OPT and FIFO.
2. State the **stack property**. Which policies satisfy it? What does this guarantee about Belady's Anomaly?
3. Explain why hardware timestamps per frame are too expensive. What two costs make them impractical?
4. Why is software-maintained LRU lists infeasible? What would the OS have to do on every single memory access?
5. What is the **Accessed (A) bit** in a PTE? Who sets it and who clears it? How does it enable LRU approximation?

§ 7 — Approximating LRU: Clock and Its Variants

📎 *Extended material → Appendix A.8, A.9*

The Clock Algorithm: Overview

The **Clock algorithm** (also called the Second-Chance algorithm) is the most widely used LRU approximation in production operating systems:

```
Key idea:
  Arrange all physical frames in a circular list.
  Maintain a "clock hand" pointer that sweeps around the circle.
  Use the Accessed bit (A) as a proxy for recent use.

Replacement rule:
  When a victim is needed, sweep the clock hand:
    If A=1: clear A to 0 (give the page a "second chance") and advance.
    If A=0: evict this page (it was not accessed since last sweep) → stop.

Intuition:
  Pages accessed recently have A=1 → they get a second chance.
  Pages not accessed since the last sweep have A=0 → they are evicted.
  The clock hand naturally sweeps past recently-used pages
  and stops at unused ones — approximating LRU.
```

Clock Algorithm: Step-by-Step Mechanics

```
Clock algorithm pseudocode:

function find_victim():
  while True:
    page = frames[clock_hand]
    if page.accessed == 1:
      page.accessed = 0      // clear the bit (second chance granted)
      clock_hand = (clock_hand + 1) % num_frames
    else:  // accessed == 0
      victim = page
      clock_hand = (clock_hand + 1) % num_frames  // advance past victim
      return victim

On page access to page P:
  PTE_P.accessed = 1   // set by hardware automatically

On page fault:
  victim = find_victim()
  evict(victim)
  load_new_page_into(victim.frame)
  new_page.accessed = 1
```

Clock Algorithm: Pseudocode

```
# Clock replacement algorithm (Python-like pseudocode)

frames = [Page(id=None, accessed=0)] * num_frames
clock_hand = 0

def replace(new_page_id):
    global clock_hand
    while True:
        if frames[clock_hand].accessed == 0:
            # Evict this page
            evict(frames[clock_hand])
            frames[clock_hand] = Page(id=new_page_id, accessed=1)
            clock_hand = (clock_hand + 1) % num_frames
            return
        else:
            # Second chance: clear bit and advance
            frames[clock_hand].accessed = 0
            clock_hand = (clock_hand + 1) % num_frames

# Hardware automatically sets frames[i].accessed = 1 on any access to frame i.
```

✅ Checkpoint 7 — Approximating LRU: Clock and Variants

Test your understanding:

1. Describe the Clock algorithm. What does the clock hand do when it encounters a page with A=1? With A=0?
2. Trace the Clock algorithm on reference string `0,1,2,3,0,1,4,0,1,2,3,4` with 3 frames. Count faults.
3. NRU uses four page categories. List them and explain which is preferred for eviction and why.
4. Explain the Aging algorithm. How does shifting the register right approximate LRU? What is the trade-off with the number of bits?
5. Define the Working Set $W(t, \Delta)$. If a process's working set has size 50 but only 30 frames are allocated, what do you expect to happen?

§ 8 — Policy Considerations and Real Systems

📎 *Extended material → Appendix A.10, A.11, A.12*

Thrashing: When Combined Working Sets Exceed Physical RAM

Thrashing occurs when the OS cannot provide enough frames to satisfy all processes' working sets:

Definition:

Thrashing = a state where processes spend more time waiting for pages to be loaded than actually executing.

Cause:

Sum of all processes' working sets > physical RAM.
Every time page P is loaded for process A, it evicts a page needed by process B. Process B then faults and loads its page, evicting a page needed by process C. And so on.

Observable symptoms:

- CPU utilization drops dramatically (processes are always blocked on I/O).
- Disk I/O utilization approaches 100%.
- System feels completely unresponsive.
- Many page faults per second (visible via vmstat, iostat).

Local vs. Global Replacement Policies

A fundamental design choice: when evicting a page, can you steal frames from other processes?

Local Replacement

Each process has a FIXED frame allocation (e.g., 20 frames).
When process A needs to evict a page, it can ONLY choose from its own 20 frames. Other processes' frames are not candidates.

Pros:

- Isolation: process A cannot harm process B's memory by evicting its pages.
- Predictable: each process's performance depends only on its own behavior.

Cons:

- Inflexible: if process A has 20 frames but only needs 5,

  the 15 unused frames sit idle even if process B is desperately short.
- Requires determining the right allocation for each process (hard problem).

✅ Checkpoint 8 — Policy Considerations and Real Systems

Test your understanding:

1. Define thrashing. What is its root cause? Describe two observable symptoms.
2. A system has 32 GB of RAM and 10 running processes with working sets of 4 GB each. What happens? What should the OS do?
3. Compare **local** and **global** replacement policies. Which does Linux use and why?
4. Explain Linux's **two-list LRU** (Active + Inactive lists). Why does a page need to be accessed twice before being promoted to the Active list?
5. What is **sequential scan thrashing**? How does Linux's two-list LRU prevent working set eviction during a large file read?
6. What are the three watermarks monitored by kswapd? What action does the OS take at each threshold?

Comparative Summary Table

Results for reference string `1,2,3,4,1,2,5,1,2,3,4,5` with 3 frames:

Policy	Total Faults	Relative to OPT	Belady's Anomaly?	Implementation Cost
OPT	7	Baseline (0%)	No	Not implementable
LRU	10	+25%	No	High (timestamps)
FIFO	9	+12.5%	**Yes**	Low (queue)
Clock	~10–11	~+25–37%	No	Very low (1 bit)
NRU	~10	~+25%	No	Low (2 bits)
Random	~11	~+37%	No	Trivial

Note: actual counts depend on tie-breaking rules; these are representative values.

Summary: Chapter 21 — Swap Mechanisms

Core concepts:

- **Swap space:** a reserved disk/SSD area holding evicted pages (4 KB slots).
- **Present bit** in PTE: 1=in RAM, 0=on disk. `valid=1, present=0` → major page fault.
- **Swap entry** in PTE: encodes (device, slot) when `present=0`.
- **Dirty bit:** set by hardware on write. Clean eviction = free; dirty = requires disk write.
- **Major page fault:** PTE.present=0 → OS reads page from disk into a free frame → updates PTE → retries.
- **Minor page fault:** page in RAM but not mapped → just update PTE, no I/O.
- **Demand paging:** pages loaded on first access, not at program start.
- **kswapd:** background daemon proactively maintaining free frame pool via three watermarks.

Lecture 2.7 Complete

Recommended Reading

- **OSTEP Ch. 21:** *Beyond Physical Memory: Mechanisms*
- **OSTEP Ch. 22:** *Beyond Physical Memory: Policies*
- Available free at: https://ostep.org

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.1	§ 1	Historical context: programmer-managed overlays	1
A.2	§ 1	Two-level memory hierarchy and modern swapping relevance	2
A.3	§ 2	Swap entry in PTE, page-out eviction, Linux setup, sizing, kswapd	5
A.4	§ 3	Hardware fault actions, step-by-step handler walkthrough (minor/major/evict)	5
A.5	§ 4	Optimal policy (OPT) unimplementability, benchmark role, future prediction	4
A.6	§ 5	Simple policies: Belady's anomaly worked example, random replacement	3
A.7	§ 6	LRU stack property, workload performance, perfect LRU cost, accessed bit	6
A.8	§ 7	Clock algorithm: clock face structure, trace simulation, comparison to LRU/FIFO	5
A.9	§ 7	Advanced approximations: NRU (clock with dirty bit), aging, working set (WSClock)	6
A.10	§ 8	Thrashing CPU curve, load control, global vs. local replacement, frame allocation	5
A.11	§ 8	Linux memory management: zone allocator, kswapd watermarks, CLOCK-Pro	3
A.12	§ 8	Worked problems (OPT/FIFO/LRU/Clock traces), exam strategy, common mistakes	9

Slides are numbered `A.section.slide`*. Checkpoints, prerequisites and summaries live in the main part.*

📎 A.1 — Historical Context: Programmer-Managed Overlays (Pre-Virtual Memory)

A.1.1 · Historical Context: The Programmer's Burden (Overlays)

Before virtual memory and swapping existed, programmers managed memory manually using **overlays**:

```
Large program that doesn't fit in RAM (1960s–1970s):

Programmer manually divides code into "overlays":
  Overlay A: initialization code (needed at startup only)
  Overlay B: main computation (needed most of the time)
  Overlay C: reporting/output (needed at the end)

At runtime:
  1. Load overlay A → run initialization → done.
  2. Manually load overlay B over A's memory region → run computation.
  3. Manually load overlay C over B's memory region → run output.

The programmer writes explicit load/unload code.
If they get it wrong: crash, corrupted data, wrong output.
This is enormously complex, error-prone, and non-portable.
```

Swapping automates exactly this process — transparently, correctly, and without programmer involvement.

📎 A.2 — Motivation: Two-Level Memory Hierarchy & Modern Swapping Relevance

A.2.1 · The Two-Level Memory Hierarchy

Swapping exploits the fact that storage forms a hierarchy with vastly different speed/cost trade-offs:

```
Level 1: Physical RAM (DRAM)
  Capacity:  4 – 256 GB (per machine)
  Latency:   60 – 100 ns
  Cost:      ~$5–10 / GB
  Volatile:  Yes (lost on power-off)

Level 2: Swap Space (SSD or HDD)
  Capacity:  10 GB – several TB
  Latency:   100 µs (NVMe SSD) to 10 ms (HDD)
  Cost:      ~$0.05–0.20 / GB
  Volatile:  No (survives power-off)

Ratio of latencies:
  NVMe SSD swap: ~1,000× slower than RAM
  HDD swap:      ~100,000× slower than RAM

Implication: swapping is a last resort, not a routine operation.
The OS must minimize swap I/O by making good eviction decisions.
```

A.2.2 · Modern Relevance: Swapping in Today's Systems

Swapping is not a relic of the past — it is critical in many modern contexts:

Smartphones (iOS, Android)

iOS does not swap but uses **compression** (compresses pages in RAM). Android swaps to a compressed RAM device (`zram`).

Containers and Cloud VMs

Cloud providers **overcommit** memory — sell more total RAM to VMs than is physically installed. Swapping is the mechanism that makes this financially viable.

Memory Overcommit in Linux

Linux by default allows `malloc()` to succeed even when physical RAM is nearly exhausted (`vm.overcommit_memory`). Pages are allocated lazily; swap handles the overflow.

Hibernation

Entire system state (all RAM pages) written to swap → power off → restore on next boot. Swapping infrastructure makes this possible.

📎 A.3 — Swap Space: Page-Out Eviction, Linux Swap Setup & Sizing Guidelines

A.3.1 · The Swap Entry in the PTE

When a page is evicted to disk (`present=0`), the OS repurposes the PFN field to encode the **swap location**:

```
PTE when page is present (P=1):

┌──────────────────────────────┬──────────────┐
│  Physical Frame Number       │  Flags       │
│  (where in RAM)              │  P=1, ...    │
└──────────────────────────────┴──────────────┘

PTE when page is swapped out (P=0, V=1):

┌──────────────────────────────┬──────────────┐
│  Swap Entry                  │  Flags       │
│  (where on disk)             │  P=0, V=1    │
└──────────────────────────────┴──────────────┘

Swap entry encodes:
  - Swap device index (which swap partition or file)
  - Slot number within that device (which 4 KB slot)

Linux swap entry: 64-bit PTE with P=0 encodes
  type (swap device) + offset (slot number) in the remaining bits.
```

A.3.2 · Page-Out (Eviction): Writing a Page From RAM to Disk

Page-out (eviction) frees a physical frame by moving its contents to swap:

```
Trigger: no free frames available; must evict a victim page V

OS eviction procedure:
  1. Select victim page V using a replacement policy (§ 4–7).
  2. Check PTE_V.dirty bit:
       Dirty=1: page has been modified → MUST write to disk.
       Dirty=0: page is identical to disk copy → skip write (free!).
  3. If dirty: write 4 KB from victim frame to swap slot on disk.
     Record (device, slot) as swap entry in PTE_V.
  4. Update PTE_V: present=0, PFN field = swap entry.
  5. Invalidate TLB entry for victim's VA (TLB shootdown if multi-core).
  6. Mark frame F as free in the physical frame free list.
  7. Frame F is now available for a new page-in.
```

A.3.3 · Swap Space on Linux: Partition vs. File

Linux supports two swap configurations:

Swap Partition

```
# Create and enable a swap partition:
mkswap /dev/sda3      # format the partition as swap
swapon /dev/sda3      # activate it
cat /proc/swaps       # view active swap areas
```

Slightly faster (no filesystem overhead). Standard on desktop Linux installs.

Swap File

```
# Create a 4 GB swap file:
fallocate -l 4G /swapfile
chmod 600 /swapfile
mkswap /swapfile
swapon /swapfile
```

More flexible (can be resized, created/deleted at runtime). Preferred on cloud instances (e.g., AWS EC2) where a dedicated partition is not available.

A.3.4 · Swap Space Sizing Guidelines

How large should swap space be?

```
Traditional rule of thumb (1990s–2000s):
  swap = 2 × RAM
  Reason: RAM was small (64 MB–1 GB); swap was heavily used.

Modern guidelines (large RAM systems):
  RAM < 2 GB:   swap = 2 × RAM
  RAM 2–8 GB:   swap = RAM
  RAM > 8 GB:   swap = 4–8 GB (enough for hibernation + moderate spikes)

Hibernation requirement:
  swap ≥ RAM (must hold entire RAM contents for hibernation to work)

Key considerations:
  More swap ≠ more performance (swapping is slow regardless of size).
  Swap is a safety net, not a performance feature.
  Better solution: add more RAM or reduce the number of processes.

Linux OOM (Out-Of-Memory) killer:
  If both RAM and swap are exhausted, Linux kills a process
  (the one deemed most dispensable by the OOM score heuristic).
```

A.3.5 · The Swap Daemon: kswapd in Linux

Linux does not wait until memory is completely exhausted before evicting pages. A background daemon, **kswapd**, proactively maintains a pool of free frames:

```
Linux memory watermarks (per memory zone):
  pages_high: target free pages  (e.g., 6% of zone)
  pages_low:  kswapd wakes up    (e.g., 4% of zone)
  pages_min:  emergency threshold (e.g., 2% of zone)

kswapd behavior:
  Free pages > pages_high:  kswapd sleeps (no pressure).
  Free pages < pages_low:   kswapd wakes up, starts evicting.
  Free pages < pages_min:   kswapd runs aggressively;
                             direct reclaim also activated
                             (allocation blocks until a frame is freed).

kswapd evicts pages in the background, using the LRU-based replacement
policy, preferring clean pages, and writing dirty pages to swap in batches
(write clustering) to improve disk throughput.
```

📎 A.4 — Page Fault Handler: Hardware Actions & Substep-by-Substep Walkthrough

A.4.1 · Hardware Actions on a Page Fault

When the MMU detects `PTE.present = 0`, the CPU performs these hardware actions automatically:

```
1. Complete the current instruction (or abort it cleanly — architecture-dependent).

2. Save processor state onto the kernel stack:
   - Instruction Pointer (RIP) — address of the FAULTING instruction (to re-execute).
   - Stack Pointer (RSP).
   - Processor flags (RFLAGS).
   - Error code (encodes: write/read, user/kernel, present/not-present).

3. Record the faulting virtual address in a special register:
   x86-64: CR2 = faulting virtual address.
   RISC-V: stval = faulting VA.

4. Switch to kernel mode (Mode Bit = Ring 0).

5. Jump to the page fault handler (IDT entry 14 on x86 = #PF).
```

The OS now runs with full knowledge of: who faulted, where they faulted, and what type of access caused the fault.

A.4.2 · Step 1: Is the Faulting Address Valid?

The OS uses **Virtual Memory Areas (VMAs)** — a per-process data structure describing all valid virtual address ranges:

```
Process VMA list (Linux struct vm_area_struct):
  VMA 1: 0x400000 – 0x401000  prot=R-X  (code segment)
  VMA 2: 0x601000 – 0x602000  prot=RW-  (data segment)
  VMA 3: 0x1000000– 0x2000000 prot=RW-  (heap, grows up)
  VMA 4: 0x7fff000– 0x8000000 prot=RW-  (stack, grows down)

Page fault at VA = 0x0000500:
  No VMA covers 0x0000500 → INVALID → SIGSEGV.

Page fault at VA = 0x1500000 (within heap VMA):
  Valid → proceed to permission check.
```

The VMA check is a software-level guard that precedes any page table lookup.

A.4.3 · Step 2: Minor Fault — Page in RAM but Not Mapped

A minor fault means the page content exists in physical RAM (e.g., in the page cache) but the current process's PTE does not yet point to it:

```
Scenario: Process B mmap()'s a file.
  Another process A already has the same file's page in the page cache (frame 42).
  Process B's PTE for this VA has present=0 (not yet mapped for B).

Page fault handler (minor fault):
  1. Look up the page in the page cache → found at frame 42.
  2. Set PTE_B.PFN = 42, PTE_B.present = 1, protection = R (file mapping).
  3. Return from fault → re-execute → success.

No disk I/O: the page was already in RAM — just needed a PTE update.
This is why minor faults are so much cheaper than major faults.
```

A.4.4 · Step 3: Major Fault — Page Is on Disk

This is the expensive case — the page must be read from swap or a file:

```
Page fault handler (major fault):

1. Read the swap entry from PTE: (device=0, slot=8192).

2. Find a free physical frame F:
   a. Check free frame list → if non-empty, take frame F.
   b. If empty: invoke page replacement policy → select victim page V.
      - If PTE_V.dirty = 1: write victim to disk first (evict dirty page).
      - Update PTE_V: present=0, PFN field = new swap entry for V.
      - TLB invalidation for victim VA.
   c. Now frame F is free.

3. Issue I/O: read 4 KB from (device=0, slot=8192) into frame F.

4. BLOCK the process: move to WAITING state.
   OS scheduler runs other processes while I/O is in flight.

5. I/O completion interrupt fires:
   OS wakes the process, sets PTE.PFN = F, PTE.present = 1.
   Returns from fault → re-execute → success.
```

A.4.5 · Step 4: Free a Frame — Invoking the Replacement Policy

When no free frame is available, the OS must choose a **victim** page to evict:

```
Frame selection (replacement policy decision):

  Which page to evict? The wrong choice causes:
    - Evict a frequently-used page → it immediately faults again (thrashing).
    - Evict a dirty page when a clean one was available → unnecessary disk write.
    - Evict a page that will be needed very soon → chain of page faults.

  The replacement policy answers: "which frame is the least valuable to keep?"
  We study these policies in §§ 4–7.

  After selecting victim frame F (containing victim page V):
    1. If PTE_V.dirty: write V to disk.
    2. PTE_V.present = 0; PTE_V stores swap entry for V.
    3. TLB entry for V's VA invalidated.
    4. Frame F is now free.
```

📎 A.5 — Optimal Policy: Unimplementability & Its Role as a Benchmark

A.5.1 · The Goal: Minimize Page Faults (Maximize Hit Rate)

We evaluate replacement policies using a **reference string** — the sequence of page numbers accessed by a program:

```
Example reference string:
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Physical memory: 3 frames (can hold 3 pages simultaneously).
Initially: all frames empty (cold start → first 3 accesses always fault).

We trace which pages are in the 3 frames after each access,
and count how many accesses cause a page fault (page not in any frame).

Metric: total number of page faults for the entire reference string.
Lower is better.
```

A.5.2 · Why OPT Is Unimplementable

OPT requires knowledge of **future memory accesses** — information that does not exist at runtime:

```
To implement OPT, the OS would need to:
  1. Know every future memory access the program will make.
  2. Compute, for each currently-loaded page, when it will next be accessed.
  3. Select the page with the maximum "time until next access" as victim.

This requires:
  - Solving the halting problem (knowing if/when each instruction executes).
  - Perfect clairvoyance of all future branches, loops, and data-dependent accesses.

Conclusion: OPT is provably impossible to implement in practice.
            It can only be computed in simulation, given a known trace.

Role of OPT: it is the theoretical lower bound on page faults.
  No algorithm can do better than OPT.
  All real algorithms are evaluated by how close they get to OPT.
```

A.5.3 · OPT as a Benchmark

```
Comparison framework:

  Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  Frames: 3

  OPT  faults: 9  ← theoretical minimum (lower bound)
  LRU  faults: 12 ← typically close to OPT
  FIFO faults: 15 ← typically further from OPT
  Rand faults: ~13 ← varies; sometimes close to LRU

  "Closeness to OPT" measures the quality of a policy.
  A policy achieving 9 faults = perfect (same as OPT).
  A policy achieving 18 faults = 2× worse than OPT.
```

OPT is the gold standard we use to assess all practical algorithms.

A.5.4 · The Crux: Using the Past to Predict the Future

Since we cannot know the future, all practical replacement policies use **past access history** as a proxy:

> ***The Principle of Temporal Locality**: if a page was accessed recently, it is likely to be accessed again soon. Therefore, the page that was accessed least recently is a reasonable approximation of the page that will be needed least soon.*

```
OPT:  evict the page needed furthest in the FUTURE. (impossible)
  ↕ approximated by ↕
LRU:  evict the page not used for the longest time in the PAST. (feasible)

The quality of this approximation depends on how well
the program's past access pattern predicts its future.
For programs with strong temporal locality (most programs):
  LRU ≈ OPT in practice.
```

📎 A.6 — Simple Policies: Random Replacement & Belady's Anomaly Worked Example

A.6.1 · Belady's Anomaly: Worked Example

```
Reference:  1   2   3   4   1   2   5   1   2   3   4   5

3 frames:
            1   2   3   4   1   2   5   1   2   3   4   5
Queue:     [1] [12][123][234][341][412][125][125][125][253][534][534]
Fault:      ✗   ✗   ✗   ✗   ✗   ✗   ✗   –   –   ✗   ✗   –
Total faults: 9

4 frames:
            1   2   3   4   1   2   5   1   2   3   4   5
Queue:     [1] [12][123][1234][1234][1234][2345][3451][4512][5123][1234][2345]
Fault:      ✗   ✗   ✗    ✗   –   –   ✗    ✗    ✗    ✗    ✗    ✗
Total faults: 10

4 frames → 10 faults > 9 faults with 3 frames.

**Belady's Anomaly confirmed!**
```

A.6.2 · Random Replacement

Random replacement selects a victim uniformly at random from all loaded pages:

```
Algorithm:
  1. Choose a random integer r in [0, num_frames).
  2. Evict the page in frame r.
  3. Load the new page into frame r.

Properties:
  + No Belady's Anomaly (more frames always helps or is neutral).
  + Extremely simple to implement (one random number per fault).
  + No bookkeeping required (no queue, no timestamps).
  - Unpredictable: sometimes evicts a heavily-used page.
  - Performance varies widely across workloads.
  - On average, similar to FIFO but without the anomaly.

Practical use:
  CPU TLB replacement often uses random (or pseudo-random) eviction
  to avoid adversarial access patterns that exploit deterministic policies.
```

A.6.3 · Why FIFO and Random Are Suboptimal

Both FIFO and Random share a fundamental flaw:

```
FIFO problem:
  Evicts the OLDEST page, regardless of how frequently it was accessed recently.
  A page loaded at time t=0 and accessed at every time step is evicted
  before a page loaded at t=100 that was never accessed again.

  Example: kernel shared library loaded at startup.
  FIFO may evict it simply because it was loaded first — even though
  it is called thousands of times per second.

Random problem:
  Evicts a uniformly random page, ignoring all access history.
  Has a 1/n probability of evicting the most-recently-used page.
  Cannot systematically exploit temporal locality.

The missing ingredient in both: RECENCY OF USE.

Better approach: use recency information.
  Pages that were accessed recently → high value → keep them.
  Pages not accessed for a long time → low value → evict them first.
```

📎 A.7 — LRU: Stack Property, Performance by Workload & Perfect LRU Cost Analysis

A.7.1 · LRU vs. OPT vs. FIFO: Comparative Fault Count

Using the full reference string `7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1` with 3 frames:

Policy	Total Faults	Faults vs. OPT
OPT	**9**	0% overhead (theoretical minimum)
LRU	**12**	+33% vs. OPT
FIFO	**15**	+67% vs. OPT
Random	~13	~+44% vs. OPT (varies)

LRU consistently outperforms FIFO and Random, and approaches OPT for workloads with good temporal locality. In practice, real programs exhibit strong locality, so LRU is often within 5–15% of OPT.

A.7.2 · LRU and the Stack Property

LRU belongs to a special class of replacement algorithms called **stack algorithms**:

> ***Stack Property:*** *For a stack algorithm, the set of pages in memory with n frames is always a subset of the set of pages in memory with $n + 1$ frames.*

```
Formal definition:
  Let S(n, t) = set of pages in memory at time t with n frames.
  Stack property: S(n, t) ⊆ S(n+1, t) for all n, t.

Implication: with more frames, you only GAIN pages — you never lose a page that would have been kept with fewer frames.
    → Adding frames never increases the fault count.
    → Belady's Anomaly is IMPOSSIBLE for stack algorithms.

LRU satisfies the stack property: LRU keeps the n most recently used pages.
  With n+1 frames: keeps the n+1 most recently used pages.
  The extra page is always one that would have been evicted with n frames.
  → The n pages kept with n frames are always a subset of n+1 pages.
```

A.7.3 · LRU Performance on Different Workload Types

LRU performs differently depending on the access pattern:

Strong Temporal Locality (loops, small working sets)

LRU ≈ OPT. Pages reused frequently stay in memory. Very few faults.

Sequential Scan (streaming through a large array once)

```
for (int i = 0; i < 10000000; i++) sum += arr[i]; // one pass, no reuse
```

LRU performs poorly: each page is used once and never reused. LRU evicts "old" pages that will never be reused anyway — no advantage over FIFO. Both are equally bad for pure sequential scans.

Random Access

LRU has no advantage over Random — all pages have equal probability of future access, so past recency is useless.

Mixed Workloads (most real programs)

LRU performs well: the frequently-reused pages (hot set) stay in memory; cold pages are gradually evicted.

A.7.4 · Software Cost of Perfect LRU

Alternatively, maintain an LRU list in software:

Approach 2: Software-maintained doubly linked list

On every memory access to page P:

1. Remove P from its current position in the list.
2. Insert P at the MRU (head) of the list.

On eviction: remove the LRU (tail) of the list.

Problem: This requires OS intervention on EVERY memory access.

- A user program makes billions of memory accesses per second.
- Each access would require a system call to update the list.
- **Overhead:** catastrophic — 1000× or more slowdown.

Verdict: Software per-access tracking is completely infeasible.

A.7.5 · The Practical Conclusion: Perfect LRU Is Not Scalable

Both hardware and software implementations of perfect LRU are too expensive for production use:

Summary of perfect LRU costs:

- Hardware timestamps:
    - Storage: 8 MB for 1M frames (non-trivial on-chip area).
    - Update: write on EVERY memory access (billions per second).
    - Scan: O(n) per eviction to find minimum timestamp.
- Software list maintenance:
    - OS intervention on EVERY access → billions of syscalls/second.
    - Completely impractical.

Conclusion:

- Perfect LRU is the theoretically ideal policy but is not implementable at the performance levels required by modern systems.

A.7.6 · The Accessed (Reference) Bit in the PTE

The key hardware feature that enables LRU approximation:

```
PTE with Accessed bit:

┌──────────────────────────┬───┬───┬───┬───┬───┬───┐
│  Physical Frame Number   │ G │ D │ A │ U │ W │ P │
└──────────────────────────┴───┴───┴───┴───┴───┴───┘
                                  ↑
                          Accessed (A) bit

Hardware behavior:
  The CPU sets A=1 automatically whenever the page is accessed
  (either read or write).
  The OS can clear the A bit (set to 0) at any time.

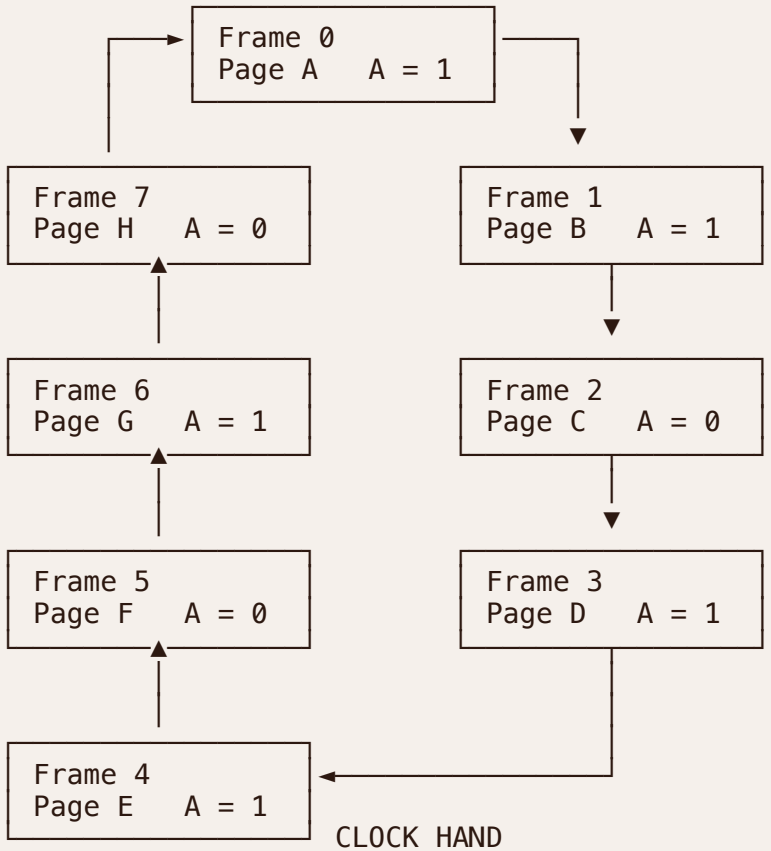
This gives the OS a 1-bit history: "has this page been accessed
since the last time I cleared the bit?"

Using just this 1-bit approximation, we can build efficient
LRU approximation algorithms — most importantly, the Clock algorithm.
```

📎 A.8 — Clock: Trace Example & Clock vs LRU/FIFO Comparison

A.8.1 · Physical Structure: The Clock Face

Clock algorithm data structure:



When replacement is needed:
  Start at clock hand position.
  Sweep clockwise: A=1 → clear A, advance; A=0 → evict, advance, stop.

A.8.2 · Clock Trace Example: Step-by-Step Eviction Simulation

```
4 frames, reference string: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
Clock hand starts at frame 0.

Step  Ref  Frames [F0,F1,F2,F3]  A bits  Hand  Action
  1    0   [0,–,–,–]             [1,0,0,0]  0→1  load 0 (cold miss)
  2    1   [0,1,–,–]             [1,1,0,0]  1→2  load 1
  3    2   [0,1,2,–]             [1,1,1,0]  2→3  load 2
  4    3   [0,1,2,3]             [1,1,1,1]  3→0  load 3
  5    0   hit                   [1,1,1,1]       A[0] already 1
  6    1   hit                   [1,1,1,1]       A[1] already 1
  7    4   FAULT → sweep:
            F0: A=1 → clear → A=0, advance to F1
            F1: A=1 → clear → A=0, advance to F2
            F2: A=1 → clear → A=0, advance to F3
            F3: A=1 → clear → A=0, advance to F0
            F0: A=0 → EVICT 0, load 4  [4,1,2,3] [1,0,0,0] 0→1
  8    0   FAULT → sweep: F1:A=0 → evict 1, load 0   [4,0,2,3] [1,1,0,0] 1→2
```

A.8.3 · Clock vs. LRU: How Close Is Clock to True LRU?

Clock is an **APPROXIMATION** of LRU, not true LRU.

Differences:

- True LRU: tracks exact time of last access for every page.
- Clock: tracks only a 1-bit "accessed recently" flag.
- True LRU: always evicts the globally least recently used page.
- Clock: evicts the first page with A=0 encountered on the sweep.

  This may not be the globally LRU page.

A.8.4 · Clock vs. LRU: How Close Is Clock to True LRU?

Clock is an **APPROXIMATION** of LRU, not true LRU.

How good is the approximation?

- In practice, Clock achieves 80–90% of LRU's quality.
- For most real workloads: Clock fault count ≈ LRU fault count.
- **Advantage:** Clock runs in O(1) amortized time per replacement. No per-access overhead (A bit set by hardware automatically).

> *Clock degenerates to FIFO when all A bits are always 0 (no temporal locality) and to MRU when all A bits are always 1 (very high temporal locality — hand sweeps far before finding A=0).*

A.8.5 · Clock vs. FIFO: Why Clock Is Strictly Better

- FIFO problem: evicts the oldest-arriving page, ignoring recency.
- Clock solution: gives recently-used pages a second chance.

Example: A shared library page loaded at t=0 but accessed every millisecond.

- FIFO: the library page is evicted after being in memory the longest.

  → Immediate fault: it is reloaded, becomes the newest, evicted later, ...

  → Constant thrashing of the library page.

- Clock: the library page always has A=1 (accessed constantly).

  → Clock hand always clears its A bit and moves on.

  → Library page is NEVER evicted as long as it is actively used.

  → No thrashing.

> *Clock correctly identifies "actively used" pages via the A bit and protects them from eviction, regardless of how long they have been loaded. FIFO has no such mechanism.*

📎 A.9 — Advanced Approximations: NRU Enhanced Clock, Aging Algorithm & Working Set

A.9.1 · Enhanced Clock (NRU): Using Both Accessed and Dirty Bits

The **Not Recently Used (NRU)** or Enhanced Clock algorithm uses **two bits** per page:

```
Two bits per page:
  A = Accessed bit  (set by hardware on any access)
  D = Dirty bit     (set by hardware on any write)

Four possible categories for each page:
  Class 0: A=0, D=0  — not accessed, not dirty    (best victim)
  Class 1: A=0, D=1  — not accessed, but dirty    (needs write-back)
  Class 2: A=1, D=0  — accessed, not dirty        (clean, recently used)
  Class 3: A=1, D=1  — accessed and dirty         (worst victim)

NRU eviction preference:
  Prefer Class 0 (evict for free — clean and cold).
  Then Class 1 (need write-back, but cold).
  Then Class 2 (clean and hot — free eviction but page may be reused).
  Avoid Class 3 (write-back AND likely to be reused — most expensive).
```

A.9.2 · NRU: Four Categories and Eviction Strategy

```
NRU Clock sweep (Enhanced Clock):
  On each sweep pass, look for the lowest-class victim:

  Pass 1: look for Class 0 (A=0, D=0) — ideal victim.
          If found: evict immediately.
  Pass 2: look for Class 1 (A=0, D=1) — write-back needed.
          While sweeping: clear A bits of Class 2 and Class 3 pages.
          If found: evict (with write-back).
  Pass 3: look for Class 2 (A=0, D=0 after clearing in pass 2).
  Pass 4: look for Class 3 (A=0, D=1 after clearing).

The A bits cleared during passes mean the next sweep will see
more Class 0 and Class 1 pages → faster victim finding.

Benefit over basic Clock:
  Preferring clean victims (D=0) avoids unnecessary disk writes.
  Dirty pages are evicted only when no clean candidate exists.
  Reduces average eviction cost (fewer disk writes per page fault).
```

A.9.3 · Aging Algorithm: Software LRU with Shift Registers

The **Aging algorithm** uses a software counter per page to maintain an approximate LRU ordering with more than 1 bit of history:

```
Each page has an N-bit shift register (counter), initially 0.

Periodically (e.g., every timer interrupt):
  1. For each page:
     a. Shift the register right by 1 bit.
     b. OR the current A bit into the leftmost (MSB) position.
     c. Clear the A bit.

Example (8-bit registers, 5 periods):
  Page A accessed in periods 1, 2, 3, 4, 5:
    After period 1: 10000000
    After period 2: 11000000
    After period 3: 11100000
    After period 4: 11110000
    After period 5: 11111000  ← high value → recently used → keep

  Page B accessed in periods 1 only:
    After period 1: 10000000
    After period 2: 01000000
    After period 3: 00100000
    After period 4: 00010000
    After period 5: 00001000  ← low value → cold → evict first
```

A.9.4 · Aging: Eviction and Properties

Eviction rule: Evict the page with the LOWEST counter value.

Properties of the Aging algorithm:

- Better LRU approximation than Clock (N bits of history vs. 1 bit).
- No per-access OS intervention (A bit set by hardware, checked periodically).
- O(1) per replacement (scan counters; pick minimum).
- Does not know the exact order of accesses within one period.
- Sensitive to period length: too long → loses fine-grained recency; too short → high overhead.
- 8-bit counter: forgets accesses more than 8 periods old.

> ***Trade-off:*** *number of bits in the counter vs. accuracy of LRU approximation.*
>
> - *1 bit = Clock algorithm.*
> - *8 bits = reasonable approximation with ~8 time periods of memory.*
> - *∞ bits = perfect LRU (but too expensive — see § 6).*

A.9.5 · Working Set Model

The **Working Set** of a process is the set of pages it actively uses during a time window:

> ***Working Set*** $W(t, \Delta)$ *= the set of pages referenced by the process in the interval* $(t - \Delta, t]$.

```
Example: reference string over time:
  t: 1  2  3  4  5  6  7  8  9  10  11  12
  P: A  B  C  A  D  A  B  A  C   A   B   A

  W(12, 5) = pages accessed in [7, 12] = {A, C, B} = {A, B, C}
  W(12, 3) = pages accessed in [9, 12] = {C, A, B} = {A, B, C}
  W(12, 1) = pages accessed in [11,12] = {B, A} = {A, B}

Key insight:
  If the OS can allocate at least |W(t, Δ)| frames to a process,
  the process experiences very few page faults.
  If frames < |W(t, Δ)|: the process thrashes (many faults).
```

A.9.6 · Working Set Clock (WSClock)

WSClock is a practical algorithm that combines Clock with working set tracking:

```
Each page frame has:
  - A bit (accessed bit)
  - D bit (dirty bit)
  - Last-use time (timestamp of last access)
  - In-working-set flag

WSClock sweep (on page fault):
  For each frame encountered by the clock hand:
    age = current_time - frame.last_use_time

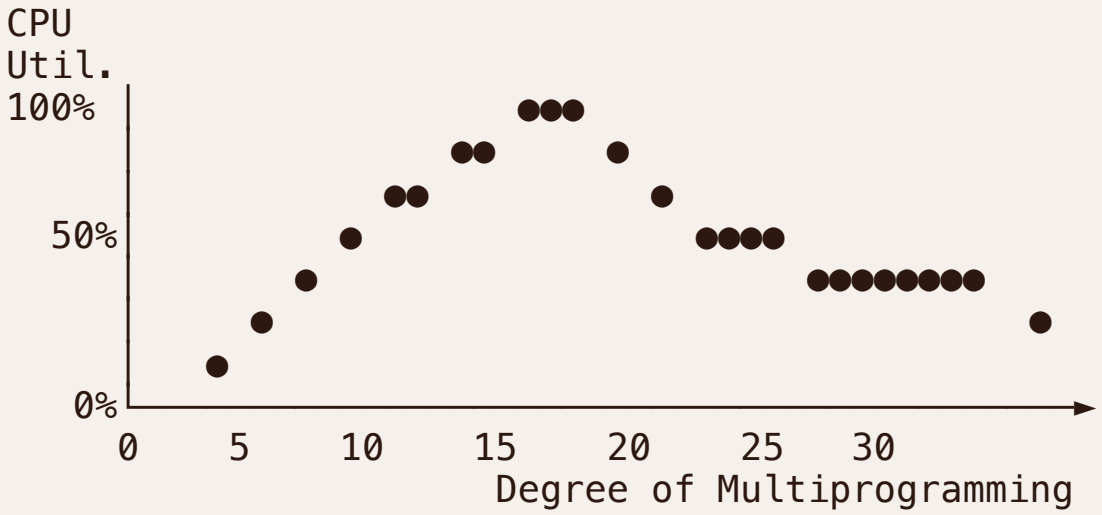
    if frame.A == 1:
      frame.last_use_time = current_time   // update last-use
      frame.A = 0                          // clear for next round
      advance hand

    else:  // A == 0
      if age > threshold:   // page has been idle for long enough
        if frame.D == 0:    // clean → evict for free
          evict(frame)
          return
        else:               // dirty → schedule write-back, continue
          schedule_writeback(frame)
          advance hand
      else:
        advance hand        // in working set despite A=0 → skip
```

📎 A.10 — Policy Details: Thrashing Curve, OS Responses, Global Policy & Frame Allocation

A.10.1 · Thrashing Illustrated: CPU Utilization Curve

CPU Utilization vs. Degree of Multiprogramming:



Up to ~10 processes: adding more increases CPU utilization (more work done).
At ~15 processes: combined working sets approach physical RAM.
Beyond ~15: thrashing → CPU utilization COLLAPSES.
The OS is busy swapping pages, not executing user instructions.

A.10.2 · OS Responses to Thrashing

When the OS detects thrashing (via page fault frequency monitoring), it has several options:

1. Working Set Model Enforcement

```
OS tracks each process's working set size W_i.
If sum(W_i) > physical_frames: suspend some processes.
  Suspended processes: swap out ENTIRE address space.
  Reduces total working set demand.
  Resume processes when memory pressure decreases.
```

2. Page Fault Frequency (PFF) Control

```
Monitor page fault rate per process:
  Fault rate too high: process needs more frames → allocate more.
  Fault rate too low:  process has excess frames → reclaim some.
  If total frames exhausted: suspend a process (reduce multiprogramming).
```

3. OOM Killer (Linux last resort)

```
If all else fails: kill the process with the highest "oom_score"
(heuristic: largest Resident Set Size + lowest priority + user processes preferred).
```

A.10.3 · Load Control: Reducing the Degree of Multiprogramming

The most effective response to thrashing is to **reduce the number of competing processes**:

```
Working set model:
  Let W_i = working set size of process i.
  If sum(W_i) > physical_frames:
    Select a process to SUSPEND (swap out entirely).
    Free up W_i frames for other processes.
    Resume when memory pressure eases.

Which process to suspend?
  Lowest priority process.
  Process with the largest working set (most frames freed).
  Process that has been running the longest (gives others a turn).
  Process blocked on I/O (low cost — it wasn't using CPU anyway).

Key insight:
  It is better to run 5 processes well than 10 processes thrashing.
  Reducing multiprogramming from 10 to 5 can increase
  overall throughput by 10× or more.
```

A.10.4 · Global Replacement Policy

Global replacement

The OS treats ALL physical frames as a shared pool. When ANY process needs a victim, it can evict from ANY process. The LRU/Clock policy operates globally across all processes.

Pros:

- Efficient: memory flows to where it is needed most.
- No over-allocation or under-allocation per process.
- Adaptive: a process whose working set shrinks gives memory to growing ones.

Cons:

- One process can "steal" frames from another, harming its performance.
- Difficult to give performance guarantees to individual processes.

> ***Practical outcome:*** *Global replacement generally achieves better overall system throughput. Most real operating systems (Linux, Windows) use global replacement. Isolation is provided by memory limits (cgroups, job objects) rather than fixed per-process frame allocations.*

A.10.5 · Frame Allocation Algorithms

How does the OS decide how many frames each process gets?

Equal Allocation

```
All processes get the same number of frames: frames/n.
Simple but ignores process size and behavior.
A small process (50 pages) gets the same as a large one (10,000 pages).
```

Proportional Allocation

```
Frames proportional to process virtual address space size (or RSS):
  frames_i = (size_i / sum(size_j)) × total_frames

More reasonable: larger processes get more frames.
Still ignores priority and actual working set size.
```

Priority-Based Allocation

```
Higher-priority processes get more frames.
Low-priority background jobs starve for frames.
Used in real-time and mixed-criticality systems.
```

📎 A.11 — Linux Memory Management: Zone Allocator, kswapd Watermarks & CLOCK-Pro

A.11.1 · Linux Memory Management: Zone Allocator and LRU Lists

Linux uses a sophisticated multi-zone, multi-list LRU system:

```
Memory zones (NUMA-aware):
  ZONE_DMA:    low physical memory (for legacy DMA devices, < 16 MB).
  ZONE_DMA32:  physical memory < 4 GB (32-bit DMA devices).
  ZONE_NORMAL: regular physical RAM (most pages live here).
  ZONE_HIGHMEM: high physical memory (32-bit only, above 896 MB).

Per-zone LRU lists (Linux's two-list LRU):
  Active list:   recently accessed pages (A bit=1 recently).
  Inactive list: pages that have not been recently accessed.

Page lifecycle:
  New page → Inactive list.
  Page accessed again (A=1 while on inactive) → promoted to Active list.
  Active page not accessed → demoted to Inactive list (after sweep).
  Inactive page not accessed → eligible for eviction.

This two-list structure avoids a single sequential scan evicting
the entire working set (as would happen with a simple LRU list).
```

A.11.2 · Linux kswapd and the Watermark System

```
Linux maintains three watermarks per zone (adjustable via /proc/sys/vm/):

  pages_high  (e.g., 6% of zone): target for free pages.
  pages_low   (e.g., 4% of zone): kswapd wakes up.
  pages_min   (e.g., 2% of zone): emergency — direct reclaim activated.

kswapd (kernel swap daemon):
  Background thread, one per NUMA node.
  Wakes when free pages < pages_low.
  Scans inactive LRU list, evicts pages (clean first).
  Writes dirty pages to swap in batches (write clustering).
  Goes back to sleep when free pages > pages_high.

Direct reclaim:
  When an allocation fails AND free pages < pages_min:
    The allocating process itself does synchronous page reclaim.
    Blocks the allocation until enough frames are freed.
    Visible as allocation latency spikes (stall).
```

A.11.3 · Linux 2Q / CLOCK-Pro: Avoiding Sequential Scan Thrashing

A known pathology of simple LRU: **sequential scan thrashing**:

```
Problem: process reads a large file sequentially (streaming I/O).
  Each file page is accessed once and never again.
  LRU keeps the most recently read file pages → evicts working set pages.
  After the scan: the process's actual working set is all gone!
  Result: massive fault storm re-loading the working set.

Linux solution: the two-list LRU (Active + Inactive).
  New file pages go to the INACTIVE list first.
  They must be accessed TWICE before being promoted to Active.
  A sequential scan touches each page only once → they stay Inactive.
  Inactive pages are evicted first → working set stays in Active list.
  Working set survives the sequential scan intact!

CLOCK-Pro: a further refinement with three hands tracking
  hot pages, cold pages, and non-resident (evicted) pages,
  providing even better scan resistance.
```

📎 A.12 — Worked Problems: OPT/FIFO/LRU/Clock Traces, Exam Strategy & Reference

A.12.1 · Reference String Methodology

Before tracing, always clarify:

```
Given a replacement problem:

1. Note the reference string (sequence of page numbers accessed).
2. Note the number of physical frames available.
3. Assume cold start: all frames empty at time 0.
   → The first k unique pages always cause faults (k = num frames).
4. For each access:
   a. Is the page already in a frame? → HIT (no fault).
   b. Is a frame free? → Load page, no eviction needed (fault, but cheap).
   c. All frames full → Apply replacement policy to find victim, evict,
      load new page (fault, with potential eviction cost).
5. Count total faults and (if asked) total evictions.
6. For dirty-aware policies: also note whether the victim was dirty.
```

A.12.2 · Problem 1: OPT — Full Trace

Reference string: `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

Frames: 3

```
OPT: evict the page whose NEXT USE is furthest in the future.

Step  Ref  Frames        Next use of each: 1→5,2→6,3→10,4→11,5→12     Fault?
  1    1   {1}                                                        ✗
  2    2   {1,2}                                                      ✗
  3    3   {1,2,3}                                                    ✗
  4    4   {1,2,4} evict 3 (next use: step 10, furthest)             ✗
  5    1   {1,2,4} hit                                               –
  6    2   {1,2,4} hit                                               –
  7    5   {1,2,5} evict 4 (next use: step 11, furthest)             ✗
  8    1   {1,2,5} hit                                               –
  9    2   {1,2,5} hit                                               –
 10    3   {2,3,5} evict 1 or 2 (next use: never)                    ✗
 11    4   {3,4,5} evict 2 or 3 (next use: never)                    ✗
 12    5   {3,4,5} hit                                               –

Total OPT faults: 7
```

A.12.3 · Problem 2: FIFO — Full Trace

Same reference string: `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

Frames: 3

```
FIFO: evict the page that has been in memory the LONGEST.
Queue shown as [oldest → newest]:

Step  Ref  Frames       FIFO Queue          Fault?
  1    1   {1}          [1]                  ✗
  2    2   {1,2}        [1,2]                ✗
  3    3   {1,2,3}      [1,2,3]              ✗
  4    4   {2,3,4}      [2,3,4] evict 1      ✗
  5    1   {3,4,1}      [3,4,1] evict 2      ✗
  6    2   {4,1,2}      [4,1,2] evict 3      ✗
  7    5   {1,2,5}      [1,2,5] evict 4      ✗
  8    1   {1,2,5}      hit                  –
  9    2   {1,2,5}      hit                  –
 10    3   {2,5,3}      [2,5,3] evict 1      ✗
 11    4   {5,3,4}      [5,3,4] evict 2      ✗
 12    5   {5,3,4}      hit                  –

Total FIFO faults: 9
```

A.12.4 · Problem 3: LRU — Full Trace

Same reference string: `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`

Frames: 3

```
LRU: evict the page NOT USED for the longest time.
Order shown as [MRU → LRU]:

Step  Ref  Frames       LRU order [MRU → LRU]     Result
  1    1   {1}          [1]                       ✗
  2    2   {1,2}        [2,1]                     ✗
  3    3   {1,2,3}      [3,2,1]                   ✗
  4    4   {2,3,4}      [4,3,2] evict 1 (LRU)     ✗
  5    1   {3,4,1}      [1,4,3] evict 2 (LRU)     ✗
  6    2   {4,1,2}      [2,1,4] evict 3 (LRU)     ✗
  7    5   {1,2,5}      [5,2,1] evict 4 (LRU)     ✗
  8    1   {1,2,5}      [1,5,2]                   –
  9    2   {1,2,5}      [2,1,5]                   –
 10    3   {1,2,3}      [3,2,1] evict 5 (LRU)     ✗
 11    4   {2,3,4}      [4,3,2] evict 1 (LRU)     ✗
 12    5   {3,4,5}      [5,4,3] evict 2 (LRU)     ✗

Total LRU faults: 10
```

A.12.5 · Problem 4: Clock Algorithm Trace

Reference string: `0, 1, 2, 3, 0, 1, 4, 0, 1, 2`

Frames: 3. Clock hand starts at frame 0.

```
Frames [F0, F1, F2] and A bits [A0, A1, A2]. Hand = H.

Step  Ref  F0   F1   F2   A0 A1 A2  H   Action
  1    0   0    -    -    1  0  0   1   Load 0 at F0. H→1.
  2    1   0    1    -    1  1  0   2   Load 1 at F1. H→2.
  3    2   0    1    2    1  1  1   0   Load 2 at F2. H→0.
  4    3   FAULT. Sweep: F0:A=1→clear,H→1. F1:A=1→clear,H→2. F2:A=1→clear,H→0 F0:A=0→evict 0.
           3    1    2    1  1  1   1   Load 3 at F0. H→1.
  5    0   FAULT. Sweep: F1:A=1→clear,H→2. F2:A=1→clear,H→0. F0:A=1→clear,H→1. F1:A=0→evict 1.
           3    0    2    1  1  1   2   Load 0 at F1. H→2.
  6    1   FAULT. Sweep: F2:A=1→clear,H→0. F0:A=1→clear,H→1. F1:A=1→clear,H→2 F2:A=0→evict 2.
           3    0    1    1  1  1   0   Load 1 at F2. H→0.
...
Total Clock faults: 8 (complete trace left as exercise)
```

A.12.6 · Exam Strategy: Page Replacement Problems

```
Systematic approach for any replacement exam question:

Step 1: Set up a table.
  Columns: reference string steps (1 to n).
  Rows: frame contents + policy-specific state (queue, A bits, etc.).

Step 2: Process each reference.
  a. Check: is the page already in any frame? → HIT, mark "-".
  b. Is a frame free? → Load page, mark "✗" (cold miss).
  c. All frames full → Apply policy to find victim:
     OPT:  find the page whose next reference is furthest away.
     FIFO: evict the oldest in the queue.
     LRU:  evict the page not accessed for the longest time.
     Clock: sweep hand; clear A=1; evict first A=0.

Step 3: Count total faults and list evictions.

Step 4: If asked for Belady's: repeat with n+1 frames; compare fault counts.
```

A.12.7 · Common Mistakes to Avoid

Mistake	Correct Approach
Counting hits as faults	Only count misses (page not in any frame)
OPT: looking backward	OPT looks FORWARD — evict page needed furthest in the FUTURE
LRU: using arrival time instead of last access	LRU uses time of LAST ACCESS, not time of arrival
FIFO: not maintaining the queue correctly	FIFO evicts the page at the HEAD (oldest arrival)
Clock: not advancing the hand after eviction	Hand ALWAYS advances past the evicted frame
Forgetting cold misses	The first k unique accesses always fault (k = num frames)
Belady's: only checking one direction	Must show fault count actually INCREASES with more frames
Claiming LRU exhibits Belady's	LRU has the stack property → Belady's Anomaly is impossible

A.12.8 · Summary: Chapter 22 — Replacement Policies

Key algorithms:

Policy	Evicts	Belady's?	Implementable?
OPT	Page needed furthest in future	No	No (oracle needed)
FIFO	Oldest-loaded page	**Yes**	Yes (simple queue)
LRU	Page unused longest in past	No	Not perfectly (too expensive)
Clock	First page with A=0 on sweep	No	Yes (1 bit per page)
NRU	Lowest (A,D) class page	No	Yes (2 bits per page)
Aging	Page with lowest shift-register value	No	Yes (N bits per page)

Key concepts: Belady's Anomaly (FIFO), stack property (LRU), temporal locality, working set, thrashing, global vs. local replacement.

A.12.9 · Recommended Exercises

Before Lecture 13, practice:

1. **OPT:** Reference string `7,0,1,2,0,3,0,4,2,3,0,3,2`, 3 frames. Trace fully; count faults.
2. **FIFO:** Same string, 3 frames. Count faults. Then try 4 frames — is Belady's Anomaly triggered?
3. **LRU:** Same string, 3 frames. Count faults. Compare to OPT.
4. **Clock:** Reference string `1,2,3,1,4,2,5,3,1,4`, 3 frames. Trace hand movement and A bit updates.
5. **EAT:** Page fault rate = 0.001 (1 in 1000 accesses). RAM access = 100 ns. SSD page fault = 200 µs. Calculate EAT and the slowdown vs. no swapping.
6. **Thrashing:** A system has 20 GB RAM and 5 processes with working sets 3 GB, 5 GB, 4 GB, 6 GB, 4 GB. What is the total working set demand? Will the system thrash?