Chapter 9: Virtual Memory

Operating Systems

Demand paging · Page faults · Effective access time · Copy-on-write · mmap

FIFO and Belady · OPT and LRU · Clock, NRU, aging · Working set, WSClock

Allocation and thrashing · Page size, TLB reach · Buddy, slab · Linux and Windows

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_09_Virtual_Memory.html

Learning Outcomes

Demand paging

Explain the purpose of virtual memory; describe the valid–invalid bit, the steps in handling a page fault, and instruction restart.

Performance

Compute the effective access time (EAT) from the page-fault probability; explain the difference between HDD, SSD, and compressed swap.

COW and mmap

Trace copy-on-write after `fork()` and how memory-mapped files relate to the page cache.

Replacement

Run FIFO, OPT, LRU, second chance, clock, NRU, aging, working set, and WSClock by hand on a reference string.

Allocation and thrashing

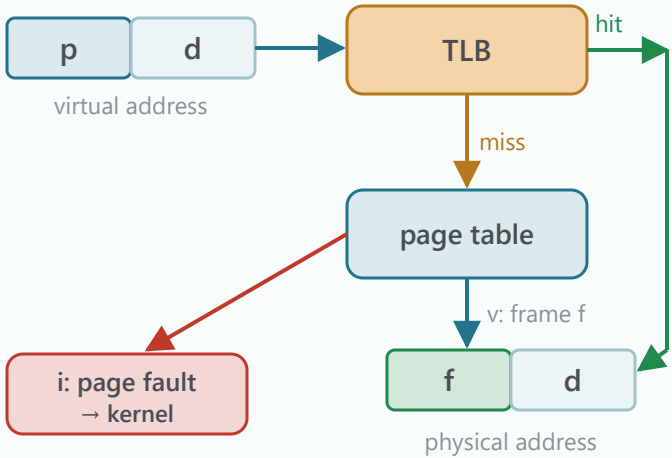
Compare equal/proportional and local/global allocation; explain thrashing with the working set and PFF.

Design and real systems

Recognize page size, TLB reach, program structure, pinning, buddy/slab, and the approaches of Linux (MGLRU, zswap) and Windows.

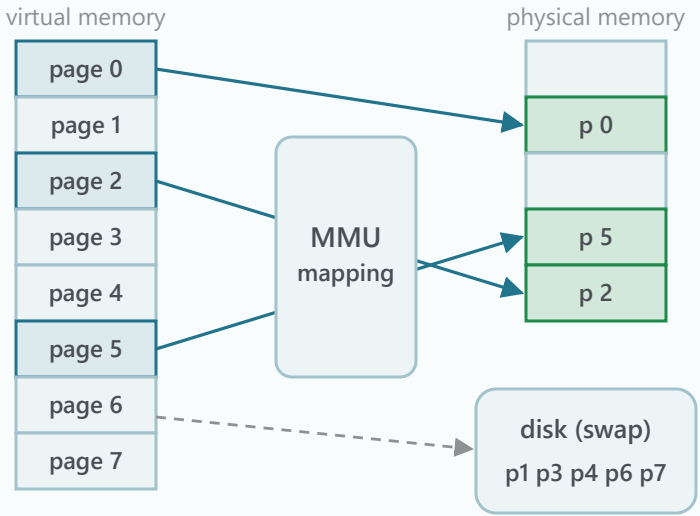
Paging and the TLB

- Virtual address = **(p, d)**; the page table maps p to a **frame**, and the offset d stays the same.
- The **TLB** caches recent translations; its hit ratio is above 99%, and on a miss the page table is walked (4 or 5 levels on x86-64).
- Each entry: frame number, **valid** (present), **R** (accessed), **M** (dirty), protection bits (R/W, U/S, NX).
- If the entry is *invalid*, a page fault occurs; if the page is **not in memory**, it is brought in from the backing store, replacing a chosen victim page if needed.



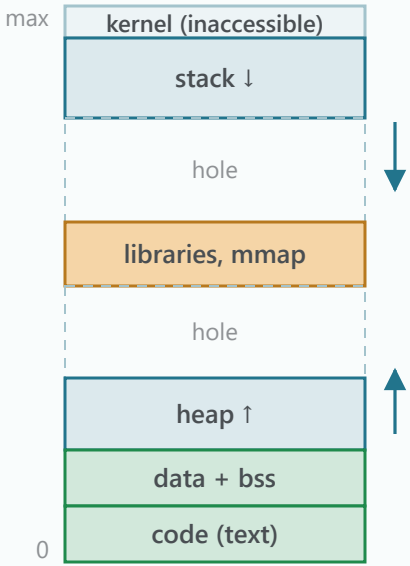
Why Virtual Memory?

- Programs rarely use all their code: error-handling routines, rarely used options, arrays allocated larger than needed.
- **Swapping** moves a process *in its entirety* to or from disk; for a large process this is expensive.
- **Virtual memory** separates logical memory from physical memory; a process runs while **only partly in memory**.
- Benefits: programs aren't limited by physical memory; fewer frames per process → **more processes**; less I/O for loading and swapping; easier sharing.
- Bloated software needn't be split by hand (the old *overlay* technique); the OS does it at page granularity.



The Virtual Address Space

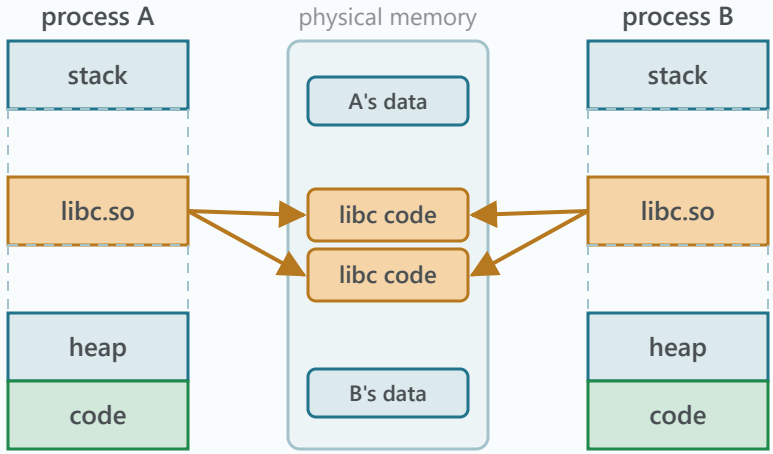
- A process's **logical view** of memory: contiguous addresses starting at 0. Physical memory is made of scattered **frames**; the MMU does the mapping.
- The stack grows **down** from the highest user address and the heap grows **up**; the unused space between is a **hole** and costs no pages or frames.
- Until the stack or heap grows into a new page, that page needs no physical memory (a *sparse* address space).
- x86-64: 48-bit virtual addresses (128 TiB for user space); 57 bits with 5-level paging. ASLR randomizes the regions' locations on every run.
- Virtual memory can be built two ways: **demand paging** (universal today) and demand segmentation (historical: Burroughs, OS/2).



A process's regions on Linux: `cat /proc/self/maps`; each line is a virtual memory area (VMA).

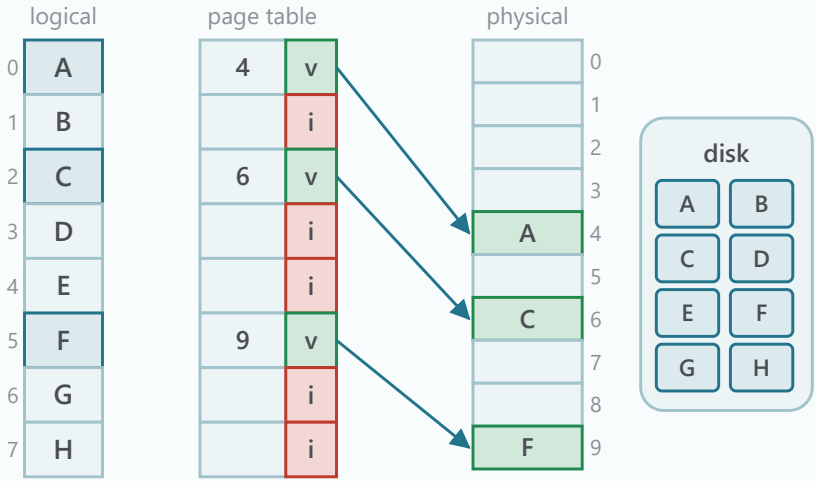
Shared Pages and Libraries

- Virtual memory makes it easy to **share** pages among processes: the same frame appears in several page tables.
- **Shared libraries** (libc, libstdc++): read-only code pages are mapped at each process's own address, with one physical copy.
- Interprocess communication through **shared memory**: two processes map the same region.
- During `fork()` pages are shared rather than copied (copy-on-write).
- Writable data pages, however, are **private** to each process.



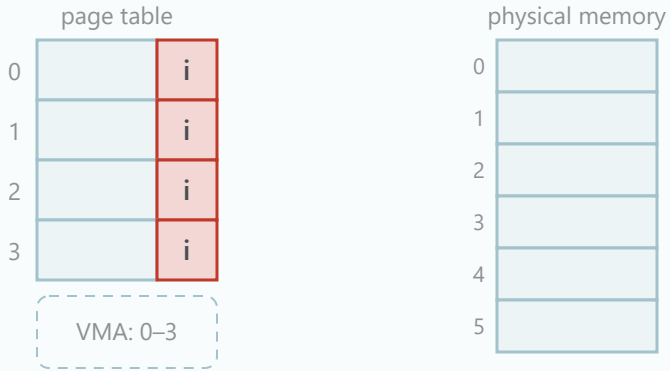
Demand Paging

- No pages (or very few) are loaded when a process starts; a page is brought in **only when it is accessed** (*lazy* loading).
- Less I/O, less memory, faster startup, more users.
- On access: if the address is **invalid** (in no region), the process is terminated; if it is valid but **not in memory**, the page is brought in.
- The component that deals with individual pages rather than whole processes is the **pager**; "lazy swapper" is an old and misleading name.
- In **pure** demand paging even the first instruction faults; thanks to locality, the fault rate soon drops.



The Valid–Invalid Bit

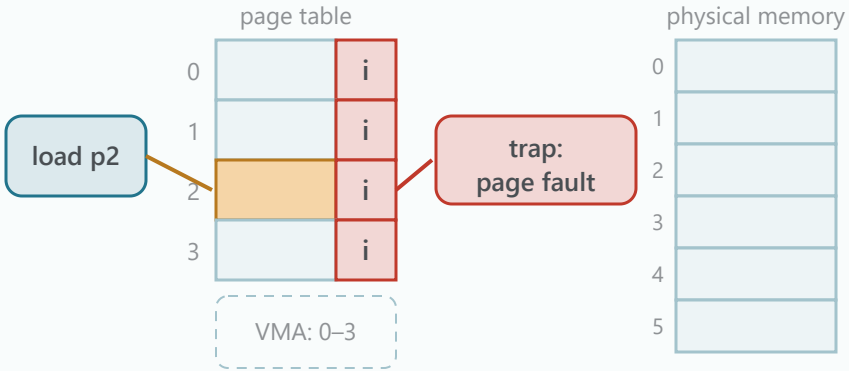
- One bit per entry: **v** = valid and in memory, **i** = not in memory *or* not in the address space. Telling the two apart is the OS's job.



Initially **every entry** of the table is set to `i` (invalid/not in memory); no frames are allocated.

The Valid–Invalid Bit

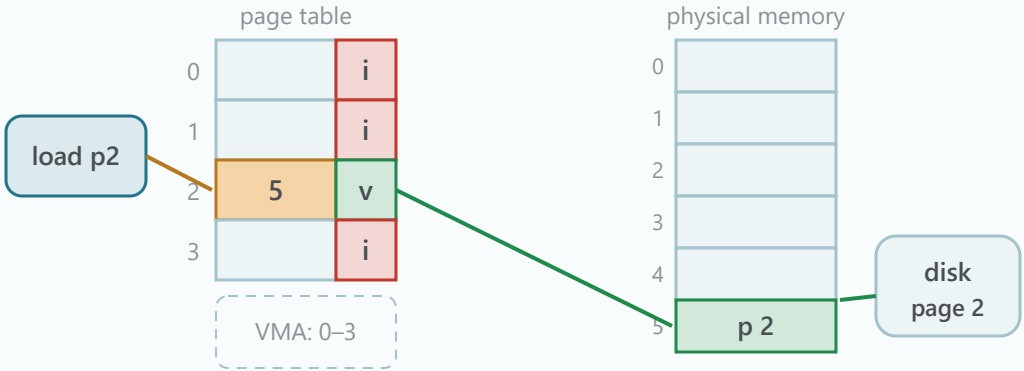
- One bit per entry: **v** = valid and in memory, **i** = not in memory *or* not in the address space. Telling the two apart is the OS's job.



The CPU accesses page 2; the MMU sees `i` in the entry and raises a **page-fault** trap.

The Valid–Invalid Bit

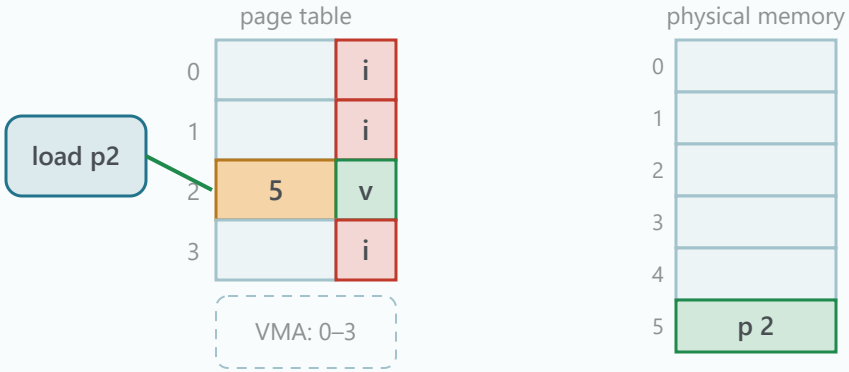
- One bit per entry: **v** = valid and in memory, **i** = not in memory *or* not in the address space. Telling the two apart is the OS's job.



The kernel verifies that the address lies in a valid region (VMA), reads the page into free frame 5, sets the entry to `5, v`; the instruction reruns.

The Valid–Invalid Bit

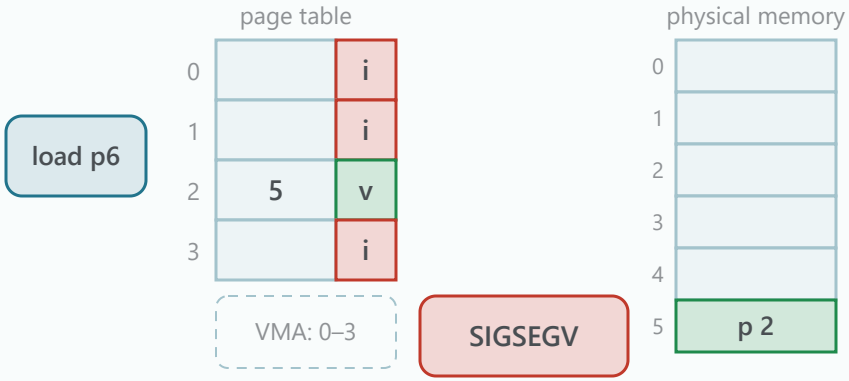
- One bit per entry: **v** = valid and in memory, **i** = not in memory *or* not in the address space. Telling the two apart is the OS's job.



Later accesses to the same page are now translated directly (and loaded into the TLB).

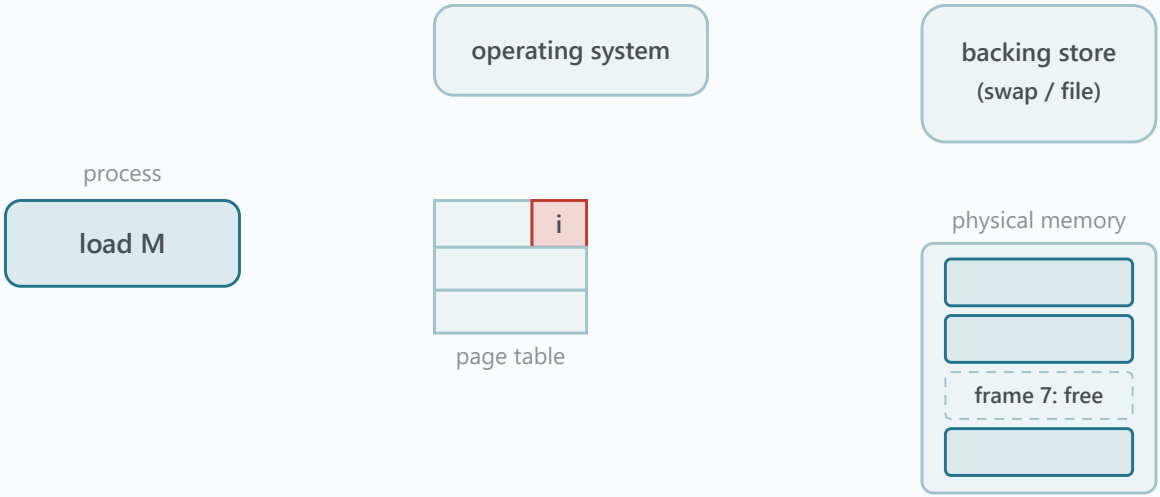
The Valid–Invalid Bit

- One bit per entry: **v** = valid and in memory, **i** = not in memory *or* not in the address space. Telling the two apart is the OS's job.



Page 6 is in no region: an **invalid reference**; the kernel terminates the process with `SIGSEGV`.

Handling a Page Fault

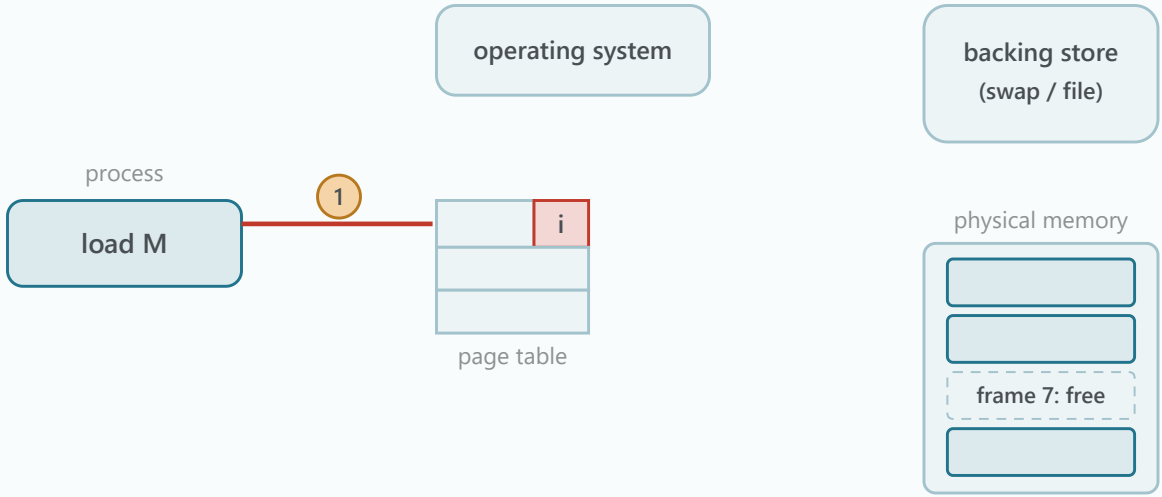


The process accesses address M; the entry's bit in the page table is `i`.

Step 1 / 7

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

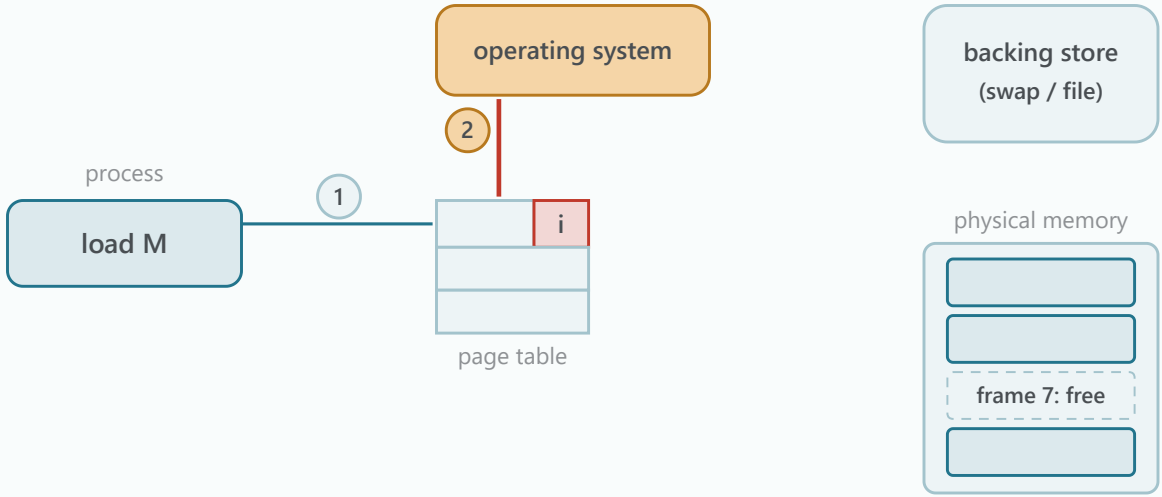
Handling a Page Fault



1. Reference: the MMU reads the entry; the page isn't in memory.

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

Handling a Page Fault

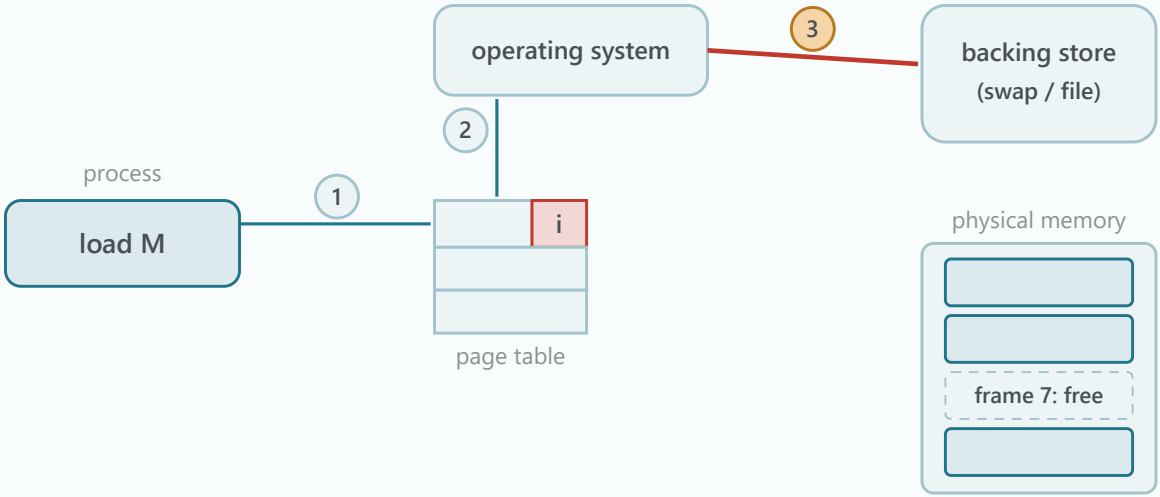


2. Trap: the hardware enters the kernel; the kernel verifies the reference is valid (otherwise it terminates the process).

Step 3 / 7

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

Handling a Page Fault

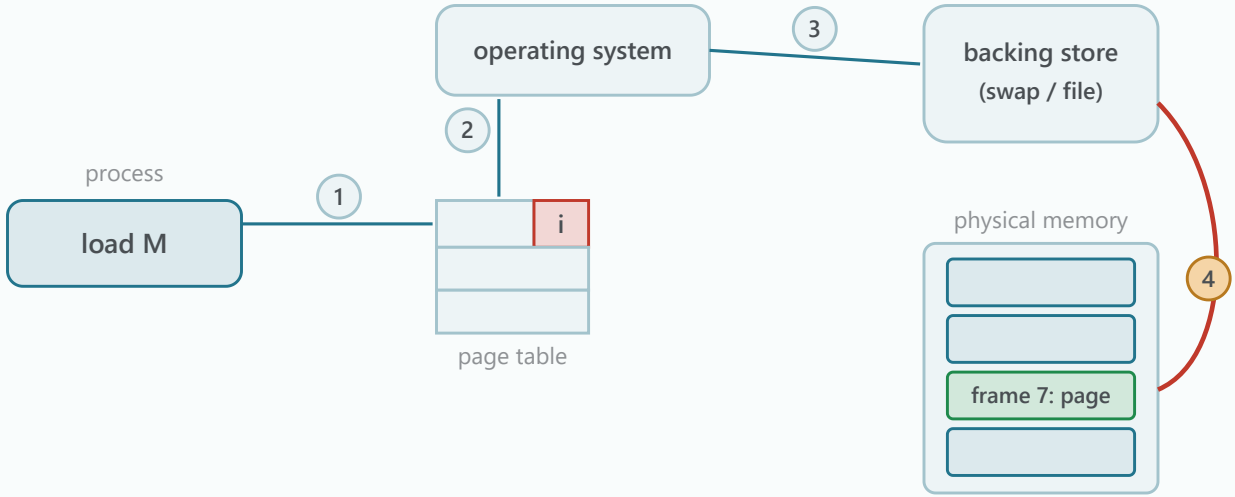


3. Page is on backing store: the page's location on disk is found (swap space or a file), and a free frame is chosen.

Step 4 / 7

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

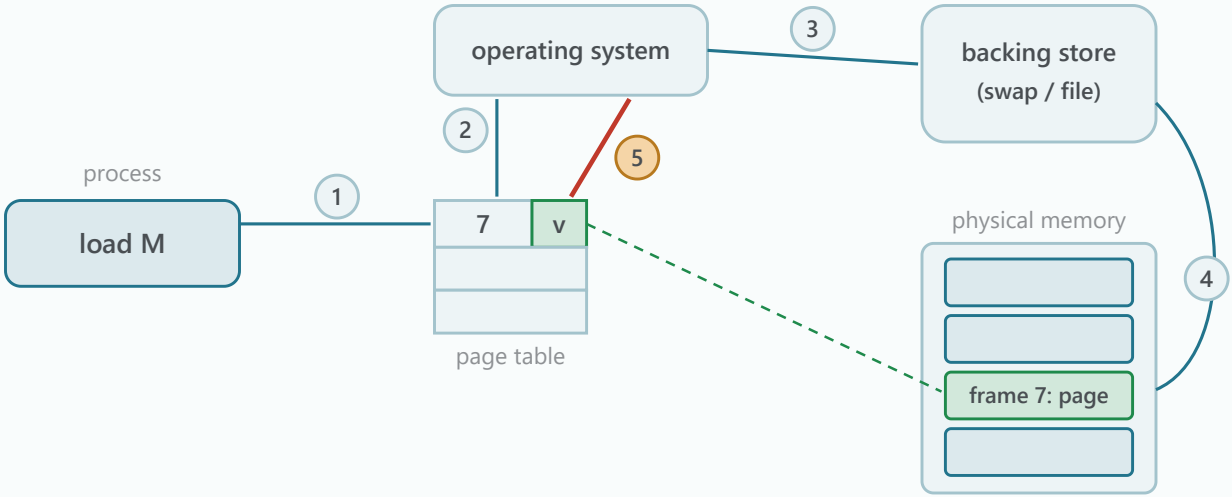
Handling a Page Fault



4. Bring it in: a disk read is scheduled; the page is read into frame 7. Meanwhile the CPU goes to another process.

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

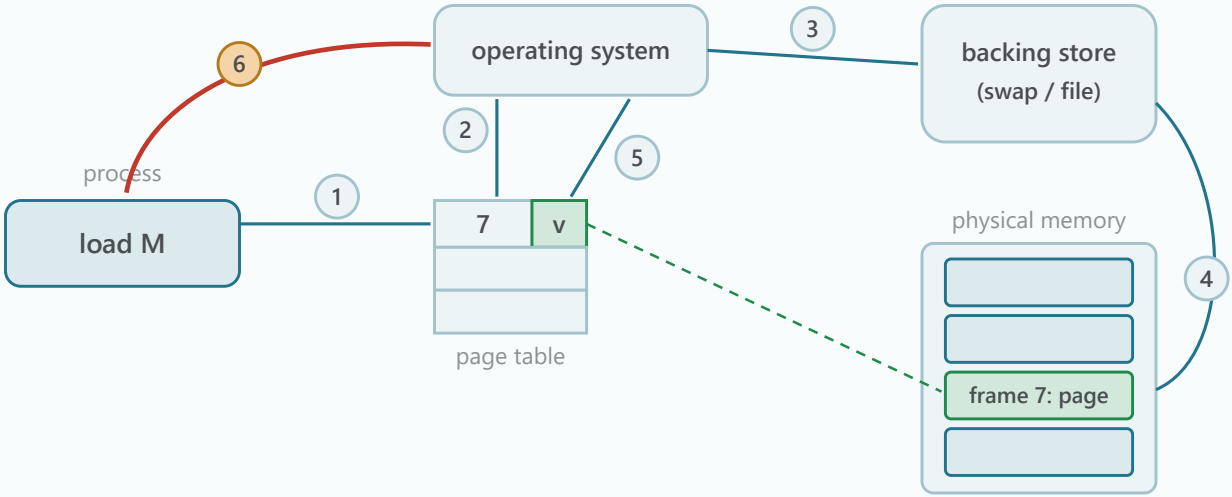
Handling a Page Fault



5. Update the table: frame 7 is written into the entry and the bit set to `v`.

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

Handling a Page Fault



6. Restart: the faulting instruction runs again; the process continues as if the page had always been there.

Step 7 / 7

A redrawing of Silberschatz Figure 10.5. A page fault is a *fault*-type trap: the saved PC points to the faulting instruction, so it can be rerun. On x86 the faulting address is written to `CR2`. A dirty victim is written to disk first, with its frame marked busy meanwhile.

Address Translation and Page Faults: Code Trace

```
▶ int translate(int va) {
2   int p = va >> 12;       /* page no */
3   int d = va & 0xFFF;     /* offset */
4   int f = tlb_find(p);    /* -1: miss */
5   if (f < 0) {
6     if (!pt[p].valid)     /* not in memory */
7       page_fault(p);      /* from disk */
8     f = pt[p].frame;
9     tlb_add(p, f);
10  }
11  pt[p].R = 1;
12  return (f << 12) | d;
13 }
```



The TLB holds translations for pages 0 and 1; page 2 is in memory (frame 5) but not in the TLB; page 3 is on disk. First call: `translate(0x2A10)`.

Step 1 / 12

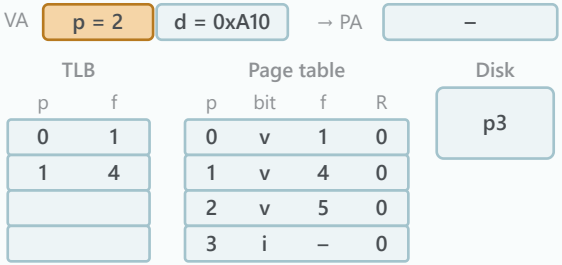
VARIABLES

va = **0x2A10** p = **–** d = **–** f = **–**

OUTPUT

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
▶    int p = va >> 12;       /* page no */
▶    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



The upper bits are the page number, the low 12 bits the offset: 0x2A10 → p = 2, d = 0xA10 (4 KB pages).

Step 2 / 12

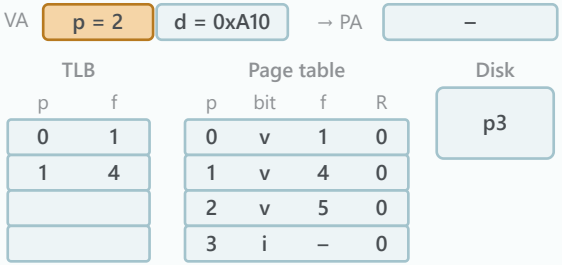
VARIABLES

va = 0x2A10 | p = 2 | d = 0xA10 | f = –

OUTPUT

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
▶    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



p = 2 isn't in the TLB: a **miss**, f = −1. Translation goes to the page table (a memory access).

Step 3 / 12

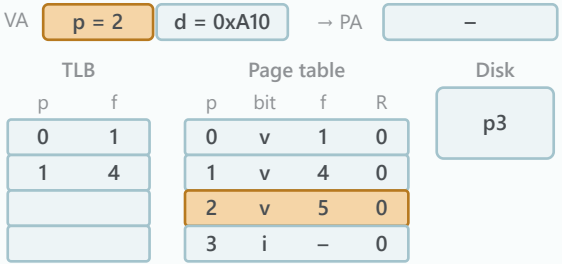
VARIABLES

va = **0x2A10** p = **2** d = **0xA10** f = **−1**

OUTPUT

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
▶    if (f < 0) {
▶      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



Entry 2's bit is `v`: the page is in memory, **no** page fault; line 7 is skipped.

Step 4 / 12

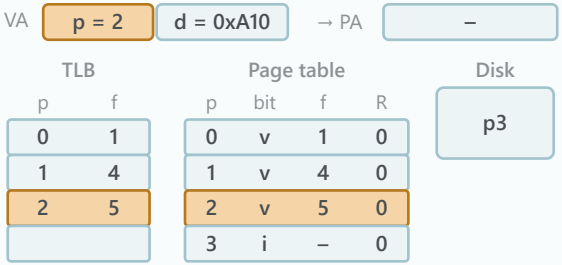
VARIABLES

`va = 0x2A10` `p = 2` `d = 0xA10` `f = -1`

OUTPUT

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
▶      f = pt[p].frame;
▶      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



f = 5 is read from the table and (2, 5) is added to the TLB: the next access to this page is translated without a table walk.

Step 5 / 12

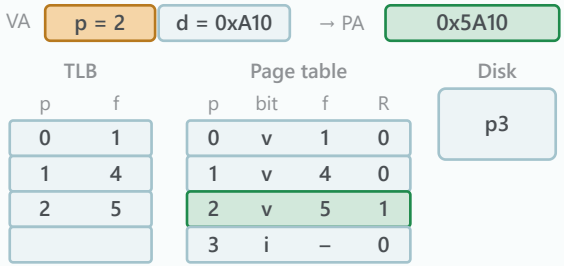
VARIABLES

va = 0x2A10 | p = 2 | d = 0xA10 | f = 5

OUTPUT

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
▶    pt[p].R = 1;
▶    return (f << 12) | d;
13 }
```



The R bit is set (replacement algorithms use it); physical address (5 << 12) | 0xA10 = **0x5A10**.

Step 6 / 12

VARIABLES

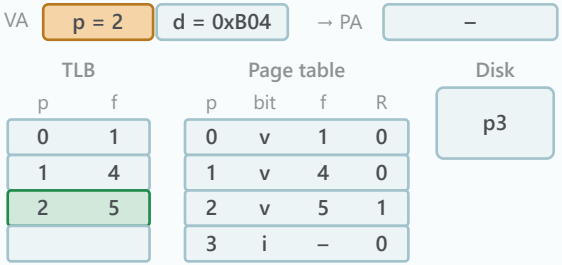
`va = 0x2A10` `p = 2` `d = 0xA10` `f = 5`

OUTPUT

```
0x2A10 -> 0x5A10
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
▶    int p = va >> 12;       /* page no */
▶    int d = va & 0xFFF;     /* offset */
▶    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



`translate(0x2B04)` : p = 2, d = 0xB04. A TLB **hit**, f = 5; the page table isn't touched.

Step 7 / 12

VARIABLES

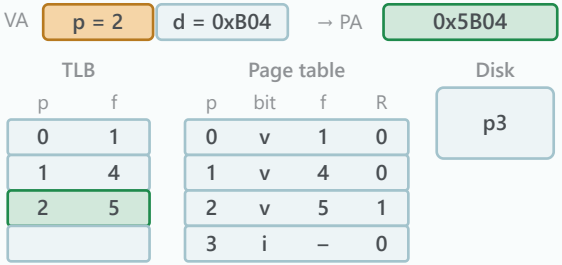
`va = 0x2B04` `p = 2` `d = 0xB04` `f = 5`

OUTPUT

```
0x2A10 -> 0x5A10
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
▶    pt[p].R = 1;
▶    return (f << 12) | d;
13 }
```



A second access to the same page: the if block is skipped, result **0x5B04**. Thanks to locality, most accesses follow this path.

Step 8 / 12

VARIABLES

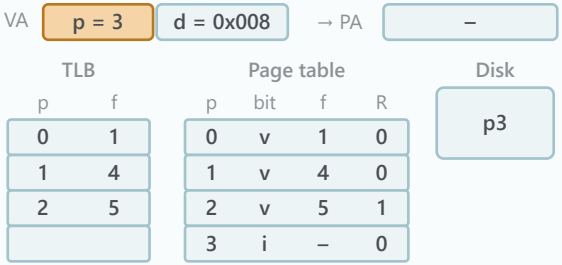
`va = 0x2B04` `p = 2` `d = 0xB04` `f = 5`

OUTPUT

```
0x2A10 -> 0x5A10
0x2B04 -> 0x5B04
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
▶    int p = va >> 12;       /* page no */
▶    int d = va & 0xFFF;     /* offset */
▶    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



`translate(0x3008)` : p = 3, d = 0x008. Not in the TLB: a miss, f = −1.

Step 9 / 12

VARIABLES

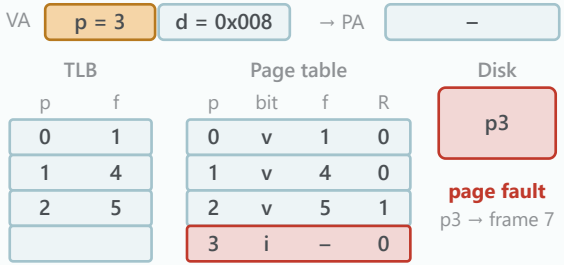
`va = 0x3008` `p = 3` `d = 0x008` `f = -1`

OUTPUT

```
0x2A10 -> 0x5A10
0x2B04 -> 0x5B04
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
▶      if (!pt[p].valid)     /* not in memory */
▶        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



Entry 3's bit is `i`: a **page fault**. The kernel reads the page from disk into free frame 7; the process waits meanwhile.

Step 10 / 12

VARIABLES

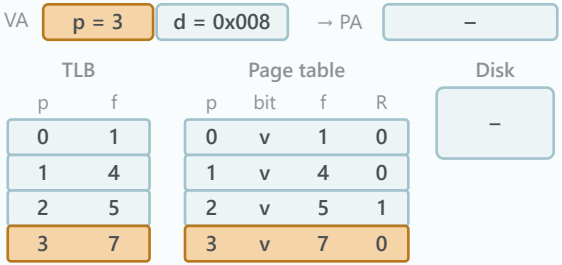
`va = 0x3008` `p = 3` `d = 0x008` `f = -1`

OUTPUT

```
0x2A10 -> 0x5A10
0x2B04 -> 0x5B04
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
▶      f = pt[p].frame;
▶      tlb_add(p, f);
10   }
11   pt[p].R = 1;
12   return (f << 12) | d;
13 }
```



The fault handler set the entry to (v, 7); when the instruction reruns, f = 7 is read and added to the TLB.

Step 11 / 12

VARIABLES

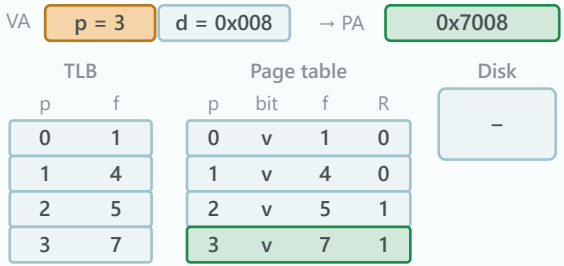
va = **0x3008** p = **3** d = **0x008** f = **7**

OUTPUT

```
0x2A10 -> 0x5A10
0x2B04 -> 0x5B04
```

Address Translation and Page Faults: Code Trace

```
1  int translate(int va) {
2    int p = va >> 12;       /* page no */
3    int d = va & 0xFFF;     /* offset */
4    int f = tlb_find(p);    /* -1: miss */
5    if (f < 0) {
6      if (!pt[p].valid)     /* not in memory */
7        page_fault(p);      /* from disk */
8      f = pt[p].frame;
9      tlb_add(p, f);
10   }
▶    pt[p].R = 1;
▶    return (f << 12) | d;
13 }
```



Result: **0x7008**. Three translations: 1 TLB hit, 2 misses, one of them a page fault (disk I/O).

Step 12 / 12

VARIABLES

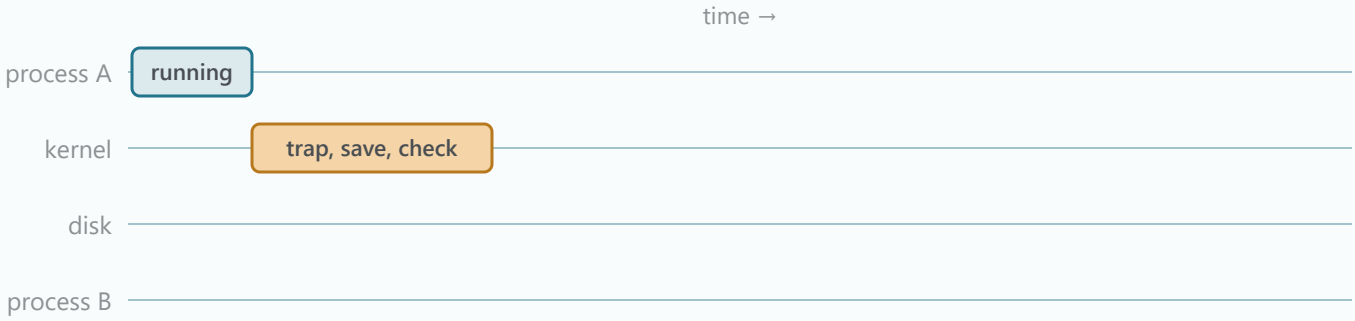
`va = 0x3008` `p = 3` `d = 0x008` `f = 7`

OUTPUT

```
0x2A10 -> 0x5A10
0x2B04 -> 0x5B04
0x3008 -> 0x7008
```

A Page Fault: Timeline

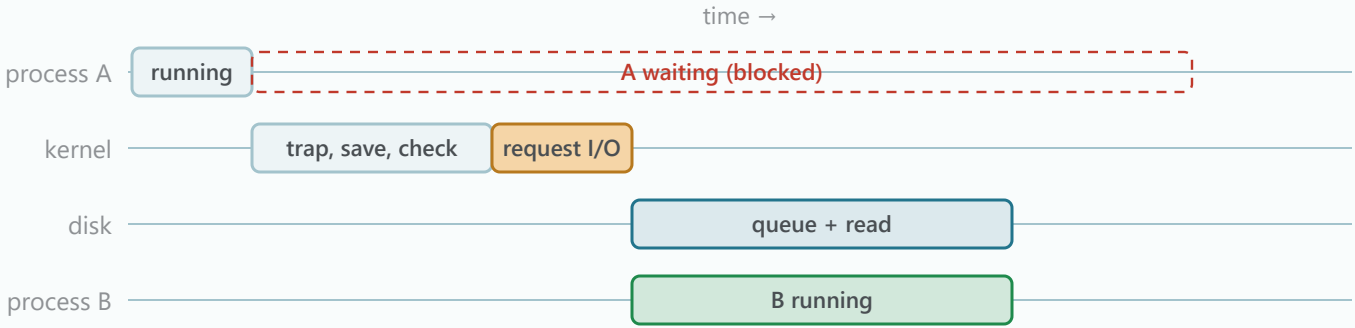
The textbook's 15 steps fall into three major costs: **servicing the trap** (µs), **reading the page** (ms on HDD, 10–100 µs on NVMe), and **restarting the process** (µs).



1–5. Trap to the kernel; registers and process state are saved; the interrupt is found to be a page fault and the reference legal; the page's location on disk is found.

A Page Fault: Timeline

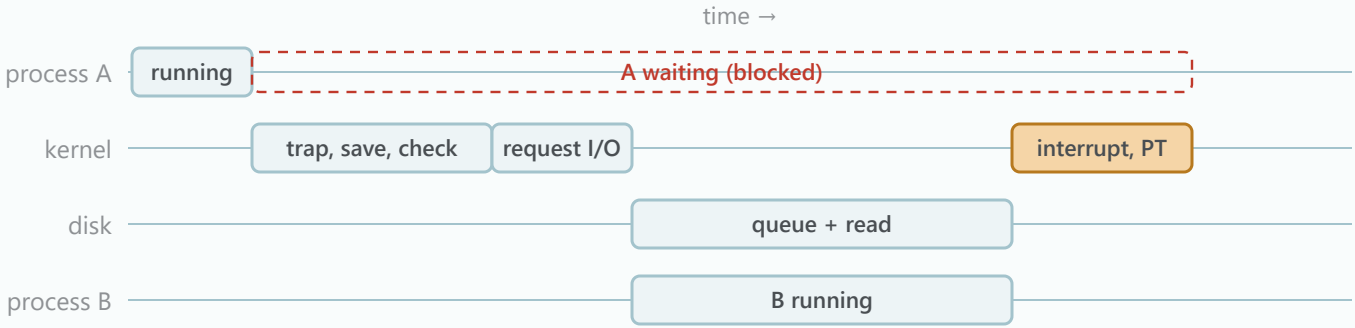
The textbook's 15 steps fall into three major costs: **servicing the trap** (µs), **reading the page** (ms on HDD, 10–100 µs on NVMe), and **restarting the process** (µs).



6–9. A read request goes on the disk/SSD queue; the device is awaited (seek + rotational latency on an HDD). Meanwhile **the CPU goes to another process** (B).

A Page Fault: Timeline

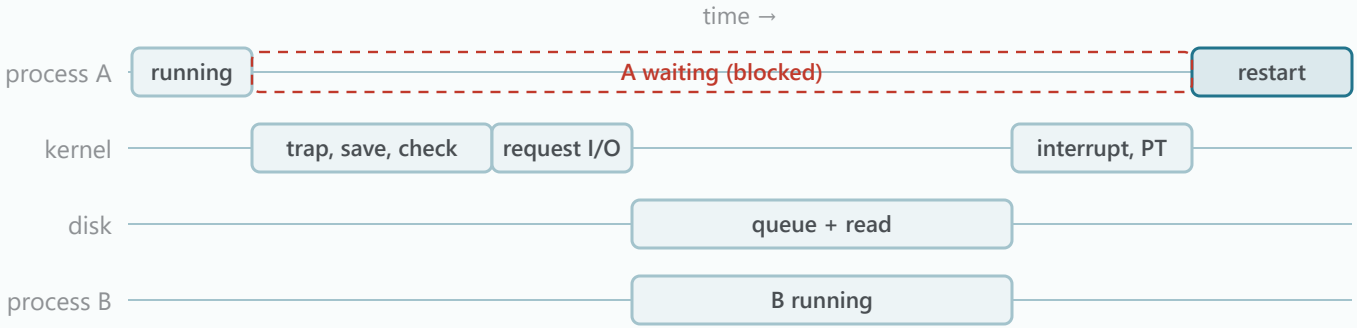
The textbook's 15 steps fall into three major costs: **servicing the trap** (µs), **reading the page** (ms on HDD, 10–100 µs on NVMe), and **restarting the process** (µs).



10–13. The I/O-complete interrupt arrives; B's state is saved, the interrupt is identified as coming from the disk, the page table is updated, and A is put on the ready queue.

A Page Fault: Timeline

The textbook's 15 steps fall into three major costs: **servicing the trap** (µs), **reading the page** (ms on HDD, 10–100 µs on NVMe), and **restarting the process** (µs).



14–15. When A gets the CPU again, its registers and page table are reloaded; the interrupted instruction runs **from the start**.

Instruction Restart and Fault Types

The restart problem

- A fault may occur while *fetching* the instruction or reading an operand; in most cases rerunning the instruction suffices.
- The hard case: the instruction faults **after modifying** some state. E.g., an IBM 360 `MVC` block move crosses a page boundary; autoincrement addressing has already changed a register.
- On the Motorola 680x0, if the three-word `MOVE.L 6(A1),2(A0)` faults while reading an operand, the OS can't tell where the instruction began; some CPUs therefore copy the PC to a hidden register before each instruction.
- Solutions: probe both ends before the instruction starts; keep old values in temporary registers and roll back. On modern CPUs **precise exceptions** guarantee this in hardware.

Fault types (Linux)

Type	What happens?
Minor	The page is already in memory (page cache, shared library, zero page, COW): only the table is updated, no I/O.
Major	The page must be read from disk/swap: I/O and waiting.
Invalid	No such region, or a permission violation: `SIGSEGV`.

```
$ /usr/bin/time -v ./prog | grep -i fault
$ ps -o min_flt,maj_flt -p PID
$ perf stat -e page-faults,major-faults ./prog
```

PERFORMANCE

Effective Access Time (EAT)

Formula

EAT = (1 − p) × *ma* + p × *page-fault time*

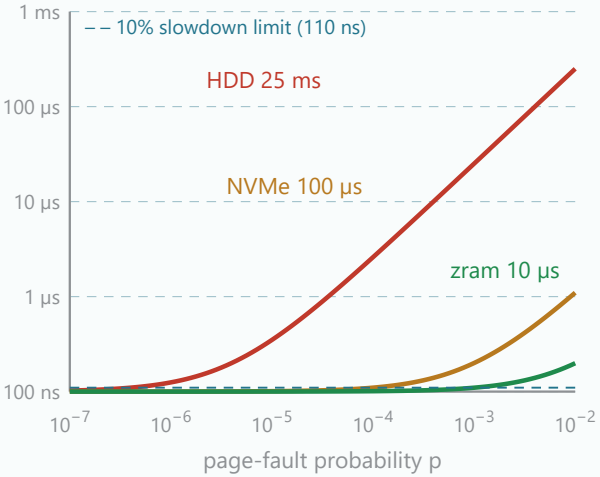
ma: memory access (100 ns); fault time: trap + read + restart; p = 0 → no faults. This covers faults that read the page from disk/swap (*major*); a *minor* fault, where the page is already in memory and only the table is updated, is far cheaper.

HDD: 25 ms = 25,000,000 ns

EAT = 100(1 − p) + 25,000,000 p = 100 + 24,999,900 p

p = 0.001 → EAT = **25,099.9 ns** ≈ 25 µs

One fault per thousand accesses slows memory down about **251×**.



Step 1 / 3

Silberschatz 10e uses 200 ns and 8 ms: p = 0.001 → 8.2 µs (41×). zram/zswap keep the page compressed in RAM; a "read" is a decompression.

Effective Access Time (EAT)

Formula

EAT = (1 − p) × *ma* + p × *page-fault time*

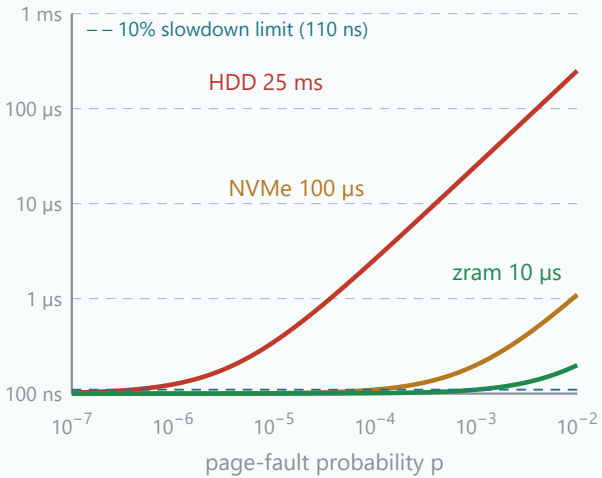
ma: memory access (100 ns); fault time: trap + read + restart; p = 0 → no faults. This covers faults that read the page from disk/swap (*major*); a *minor* fault, where the page is already in memory and only the table is updated, is far cheaper.

Goal: at most 10% slowdown

110 > 100 + 24,999,900 p → p < 4.0×10^{-7}

i.e., **fewer than one fault per 2.5 million accesses**

p < 0.0000004: faults must be extremely rare.



Silberschatz 10e uses 200 ns and 8 ms: p = 0.001 → 8.2 µs (41×). zram/zswap keep the page compressed in RAM; a "read" is a decompression.

Effective Access Time (EAT)

Formula

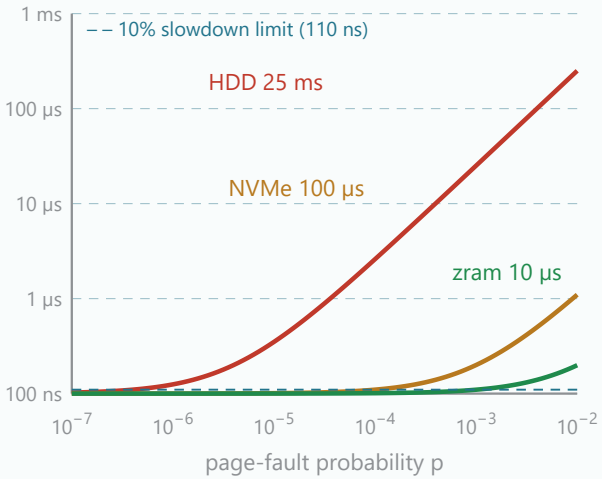
EAT = (1 − p) × *ma* + p × *page-fault time*

ma: memory access (100 ns); fault time: trap + read + restart; p = 0 → no faults. This covers faults that read the page from disk/swap (*major*); a *minor* fault, where the page is already in memory and only the table is updated, is far cheaper.

2026: NVMe 10–100 µs, zram ≈ 10 µs

NVMe (at 100 µs), p = 0.001 → EAT ≈ **199.9 ns**; for 10%, $p < 1.0 \times 10^{-4}$

zram, p = 0.001 → EAT ≈ **109.9 ns**



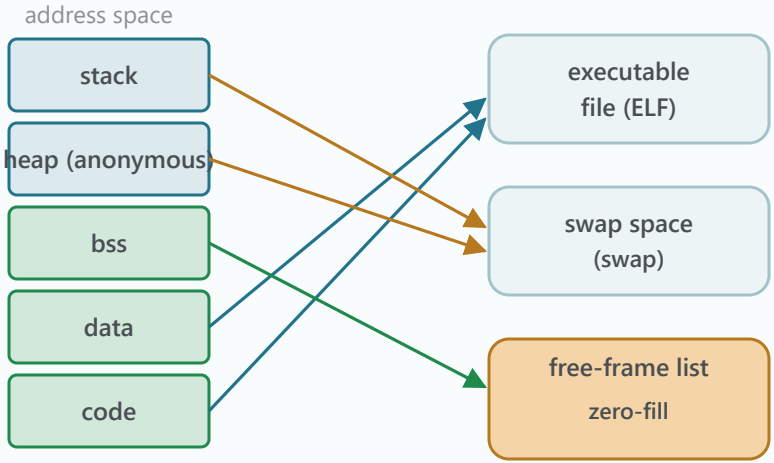
Fast storage cuts the penalty 250–2500×, but memory is still **hundreds of times** faster, so replacement algorithms still matter.

Step 3 / 3

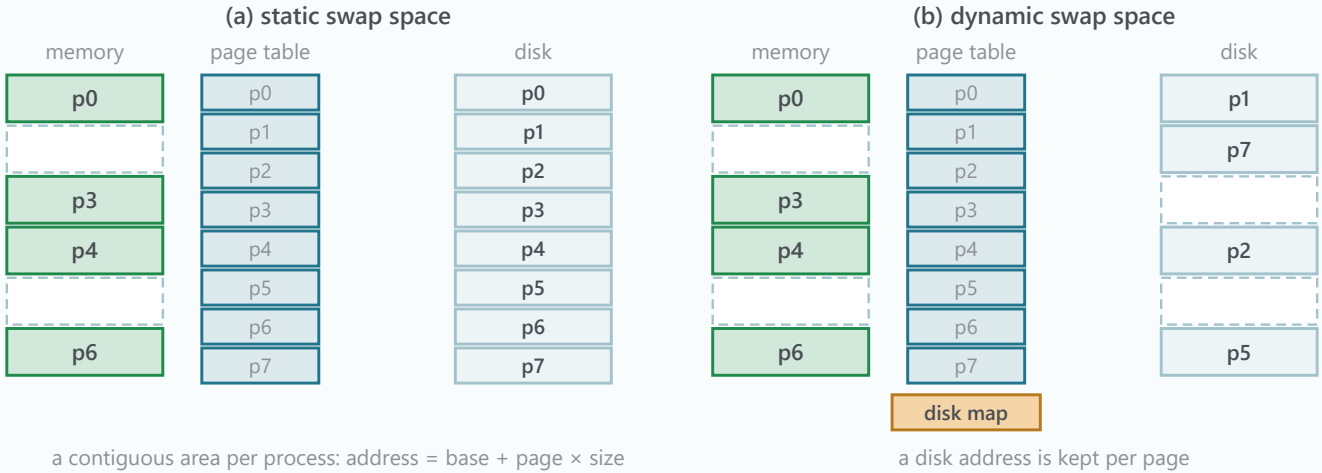
Silberschatz 10e uses 200 ns and 8 ms: p = 0.001 → 8.2 µs (41×). zram/zswap keep the page compressed in RAM; a "read" is a decompression.

Where Do Pages Come From?

- **File-backed** pages (code, read-only data, `mmap`ed files): if clean they are dropped and *reread* from the file when needed.
- **Anonymous** pages (heap, stack, modified data): they have no file and must be written to **swap space** before being dropped.
- **Zero-fill-on-demand**: bss and new heap pages are given a zeroed frame on first access; this is essential so old data doesn't leak.
- The OS keeps a **free-frame pool** so no search is needed at fault time. When the pool runs low, reclaim starts.
- Swap space can be a separate partition or a file; on Linux, `swapon --show`.



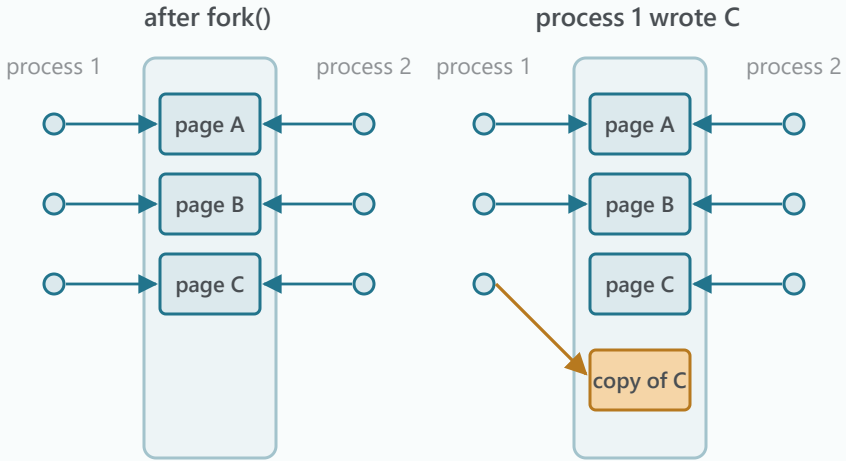
The Backing Store (Swap Space)



- **Static**: when the process starts, contiguous swap space of its full size is allocated; the start address is in the process table. Code, data, and stack may have separate areas; the stack can grow.
- **Dynamic**: nothing is allocated in advance; when a page goes out a slot is found and its address kept in a per-page **disk map**. Unmodified code pages are never swapped; they are reread from the executable.
- Location: a separate disk/partition (a Linux swap partition) or a file in the file system (`swapfile` , Windows `pagefile.sys`).

Copy-on-Write

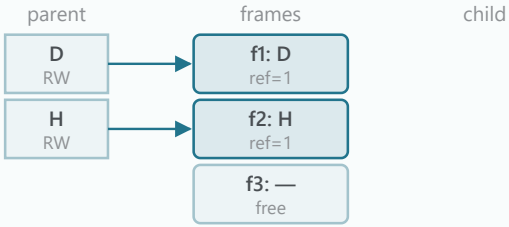
- `fork()` doesn't copy the parent's pages; parent and child **share them**, and writable pages are marked **read-only + COW** in both.
- A write by either causes a protection fault; the kernel copies **that one page** and maps the writable copy into the writer.
- Unmodified pages (code, read-only data) stay shared; `fork()` + `exec()` is almost free.
- The copy's frame comes from the zero-fill pool.
- `vfork()` doesn't even copy the page table (the parent waits); `posix_spawn()` is preferred today.



The unit of sharing is the **page**, not the file: only the written page is copied; the others stay shared.

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
▶    char *b = malloc(4096);   /* heap: H */
▶    b[0] = 'a';
5    pid_t pid = fork();
6    if (pid == 0) {           /* child */
7      printf("%d\n", x);      /* read */
8      x = 2;                  /* write */
9      _exit(0);
10   }
11   wait(NULL);
12   b[0] = 'b';               /* parent */
13   return 0;
14 }
```



The parent uses pages D (data) and H (heap) as writable.

Step 1 / 6

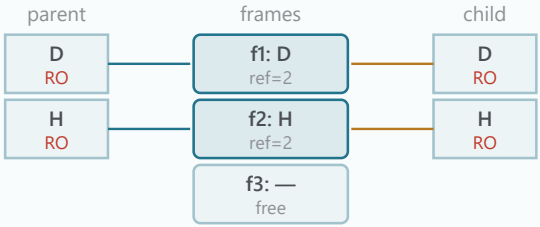
VARIABLES

`parent.pid = –` `parent.x = 1` `copies = 0`

OUTPUT

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
3    char *b = malloc(4096);   /* heap: H */
4    b[0] = 'a';
▶    pid_t pid = fork();
6    if (pid == 0) {           /* child */
7      printf("%d\n", x);      /* read */
8      x = 2;                  /* write */
9      _exit(0);
10   }
11   wait(NULL);
12   b[0] = 'b';               /* parent */
13   return 0;
14 }
```



`fork()` : only the page tables are copied; both sides are **read-only** (RO), reference counts 2.

Step 2 / 6

VARIABLES

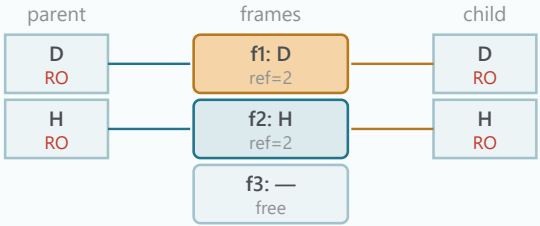
`child.pid = 0` `parent.pid = 4242`

`child.x = 1` `parent.x = 1` `copies = 0`

OUTPUT

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
3    char *b = malloc(4096);   /* heap: H */
4    b[0] = 'a';
5    pid_t pid = fork();
6    if (pid == 0) {           /* child */
▶      printf("%d\n", x);     /* read */
8      x = 2;                  /* write */
9      _exit(0);
10   }
11   wait(NULL);
12   b[0] = 'b';               /* parent */
13   return 0;
14 }
```



The child reads `x`: reading is allowed, so **no fault** and no copy.

Step 3 / 6

VARIABLES

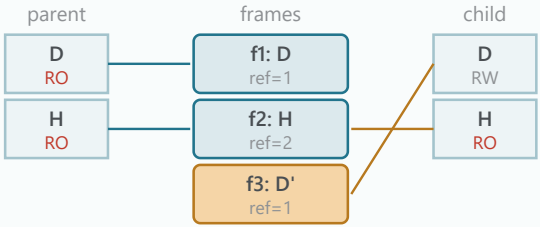
`child.pid = 0` `child.x = 1` `parent.x = 1` `copies = 0`

OUTPUT

```
1
```

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
3    char *b = malloc(4096);   /* heap: H */
4    b[0] = 'a';
5    pid_t pid = fork();
6    if (pid == 0) {           /* child */
7      printf("%d\n", x);      /* read */
▶      x = 2;                  /* write */
9      _exit(0);
10   }
11   wait(NULL);
12   b[0] = 'b';               /* parent */
13   return 0;
14 }
```



The child writes `x = 2` → protection fault. The kernel recognizes COW: it allocates f3, copies D, and maps it **RW** into the child.

Step 4 / 6

VARIABLES

`child.pid = 0` `child.x = 2`

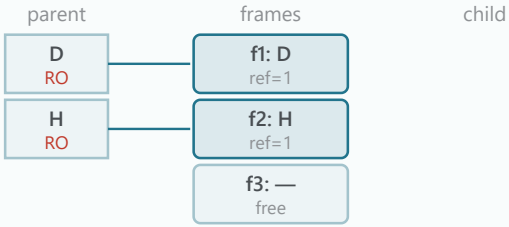
`parent.x = 1` `copies = 1`

OUTPUT

```
1
```

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
3    char *b = malloc(4096);   /* heap: H */
4    b[0] = 'a';
5    pid_t pid = fork();
6    if (pid == 0) {           /* child */
7      printf("%d\n", x);     /* read */
8      x = 2;                  /* write */
▶      _exit(0);
10   }
11   wait(NULL);
12   b[0] = 'b';               /* parent */
13   return 0;
14 }
```



The child exits: its mappings are removed, f3 is freed, and f2's reference count drops to 1.

Step 5 / 6

VARIABLES

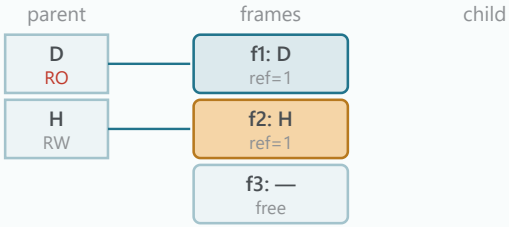
`child.pid = 0` `child.x = 2` `parent.x = 1` `copies = 1`

OUTPUT

```
1
```

fork() and COW, Step by Step

```
1  int x = 1;                  /* data: D */
2  int main(void) {
3    char *b = malloc(4096);   /* heap: H */
4    b[0] = 'a';
5    pid_t pid = fork();
6    if (pid == 0) {           /* child */
7      printf("%d\n", x);      /* read */
8      x = 2;                  /* write */
9      _exit(0);
10   }
11   wait(NULL);
▶    b[0] = 'b';               /* parent */
13   return 0;
14 }
```



The parent writes H → fault; the reference count is 1, so **no copy is needed**; the page is simply made writable.

Step 6 / 6

VARIABLES

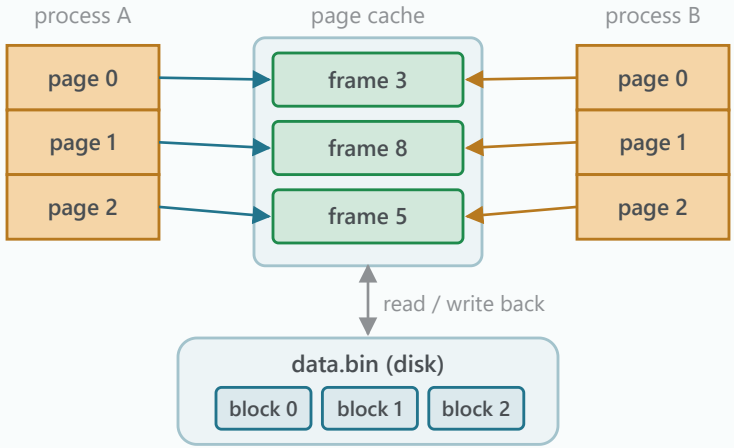
`parent.pid = 4242` `parent.x = 1` `copies = 1`

OUTPUT

```
1
```

Memory-Mapped Files

- `mmap()` maps a file into a region of the address space; the file is read and written through **memory accesses** instead of `read()/write()`.
- Pages are brought in on demand; the first access causes a (minor or major) page fault. Dirty pages are written to the file in the background or by `msync()` / `munmap()`.
- `MAP_SHARED`: changes are visible in the file and to **everyone** mapping it → IPC via shared memory.
- `MAP_PRIVATE`: writes go to a private copy via **COW**; the file is unchanged (this is how executables' data sections are loaded).
- `MAP_ANONYMOUS`: a region with no file (large `malloc` blocks). Windows: `MapViewOfFile`.



mmap, Step by Step

```
▶ int fd = open("data.bin", O_RDWR);
▶ struct stat st;
▶ fstat(fd, &st);          /* 12 KiB = 3 pages */
4 char *p = mmap(NULL, st.st_size,
5      PROT_READ | PROT_WRITE,
6      MAP_SHARED, fd, 0);
7 close(fd);               /* mapping stays */
8 char c = p[0];           /* page 0 */
9 p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



The file is opened and its size found: 12 KiB, i.e., 3 pages.

Step 1 / 7

VARIABLES

`fd = 3` `p = –` `faults = 0`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
▶  char *p = mmap(NULL, st.st_size,
▶       PROT_READ | PROT_WRITE,
▶       MAP_SHARED, fd, 0);
7  close(fd);               /* mapping stays */
8  char c = p[0];           /* page 0 */
9  p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



`mmap` only creates a **VMA**: the table entries are empty and no page has been read yet.

Step 2 / 7

VARIABLES

`fd = 3` `p = 0x7f3a…000` `faults = 0`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
4  char *p = mmap(NULL, st.st_size,
5       PROT_READ | PROT_WRITE,
6       MAP_SHARED, fd, 0);
▶  close(fd);               /* mapping stays */
8  char c = p[0];           /* page 0 */
9  p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



`close(fd)` doesn't break the mapping; the VMA keeps its own reference to the file.

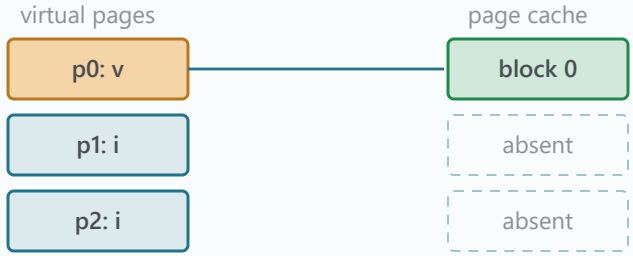
Step 3 / 7

VARIABLES

`fd = closed` `p = 0x7f3a…000` `faults = 0`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
4  char *p = mmap(NULL, st.st_size,
5       PROT_READ | PROT_WRITE,
6       MAP_SHARED, fd, 0);
7  close(fd);               /* mapping stays */
▶  char c = p[0];           /* page 0 */
9  p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



`p[0]` → page fault; block 0 is read into the page cache (a major fault), and the entry becomes valid.

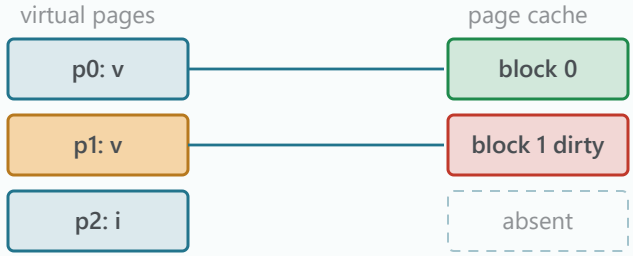
Step 4 / 7

VARIABLES

`c = '…'` `faults = 1`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
4  char *p = mmap(NULL, st.st_size,
5       PROT_READ | PROT_WRITE,
6       MAP_SHARED, fd, 0);
7  close(fd);               /* mapping stays */
8  char c = p[0];           /* page 0 */
▶  p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



`p[5000]` : 5000 / 4096 → page 1; fault, read, write → the page is **dirty** (M=1).

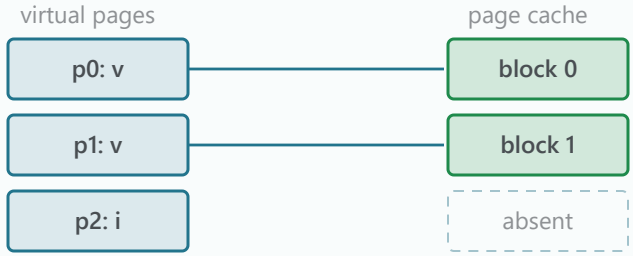
Step 5 / 7

VARIABLES

`c = '…'` `faults = 2`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
4  char *p = mmap(NULL, st.st_size,
5       PROT_READ | PROT_WRITE,
6       MAP_SHARED, fd, 0);
7  close(fd);               /* mapping stays */
8  char c = p[0];           /* page 0 */
9  p[5000] = 'x';           /* page 1 */
▶  msync(p, st.st_size, MS_SYNC);
11 munmap(p, st.st_size);
```



`msync` writes the dirty page to the file; M is cleared. Page 2 was never touched, never read.

Step 6 / 7

VARIABLES

`faults = 2` `written = 1 page`

mmap, Step by Step

```
1  int fd = open("data.bin", O_RDWR);
2  struct stat st;
3  fstat(fd, &st);          /* 12 KiB = 3 pages */
4  char *p = mmap(NULL, st.st_size,
5       PROT_READ | PROT_WRITE,
6       MAP_SHARED, fd, 0);
7  close(fd);               /* mapping stays */
8  char c = p[0];           /* page 0 */
9  p[5000] = 'x';           /* page 1 */
10 msync(p, st.st_size, MS_SYNC);
▶  munmap(p, st.st_size);
```



`munmap` removes the VMA; the pages may stay in the cache, and another process opening the file uses them with a minor fault.

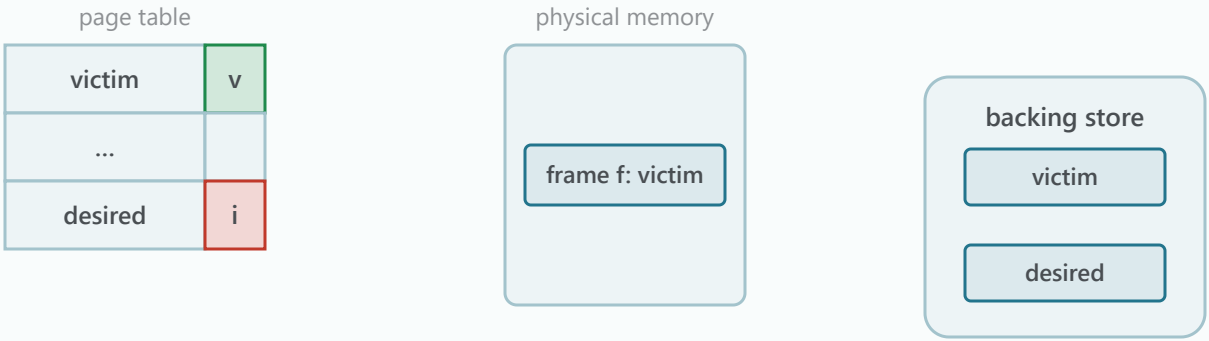
Step 7 / 7

VARIABLES

`p = –` `faults = 2`

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.



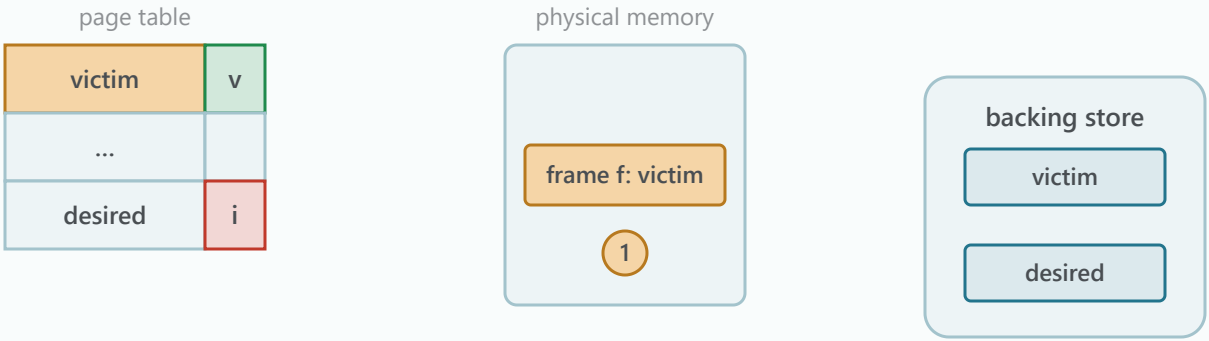
A page fault: the desired page was found on disk, but there is **no free frame**.

Step 1 / 6

Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.

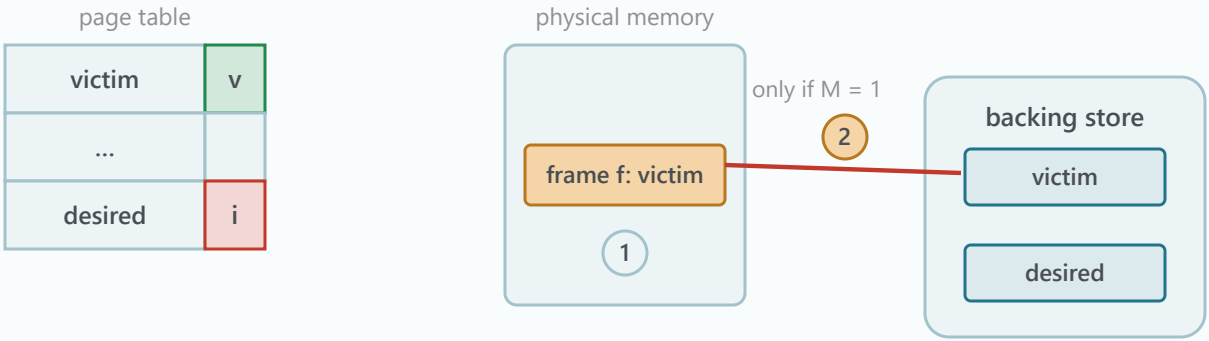


1. The replacement algorithm picks a **victim** frame.

Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.

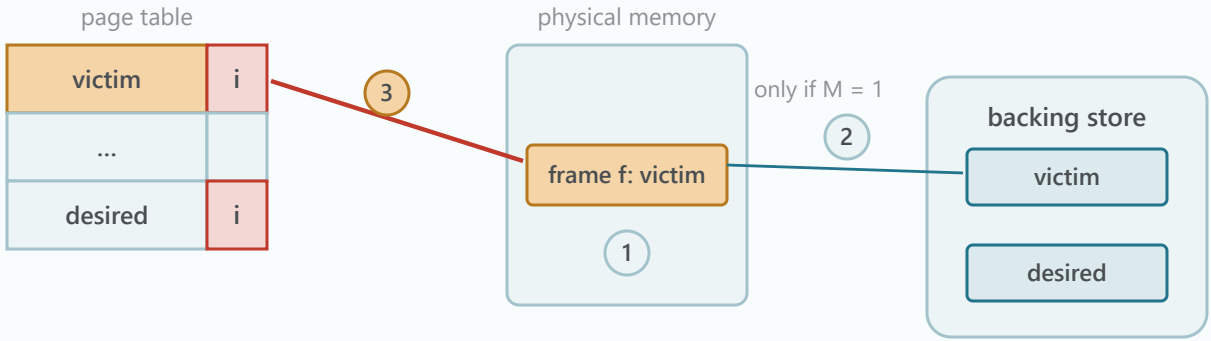


2. If the victim is **dirty** (M = 1) it is written to swap/file; if clean, no write is needed: I/O is halved.

Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.

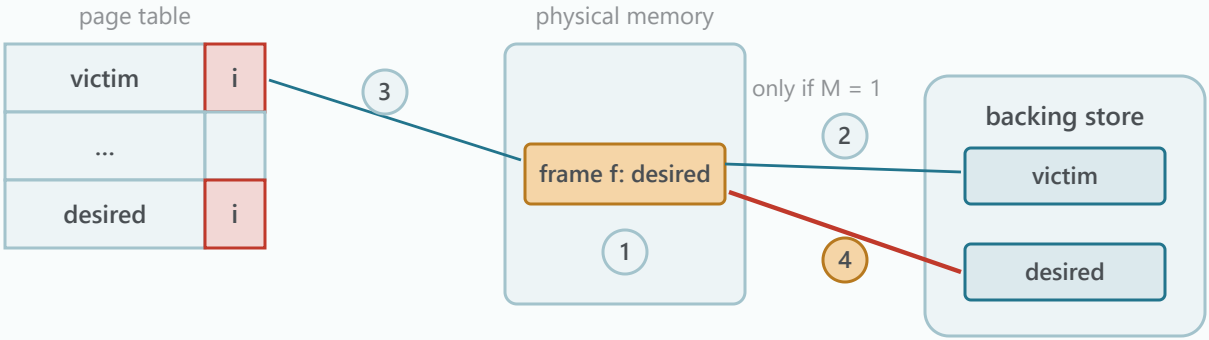


3. The victim's entry is set to `i` (and removed from the TLB: a *TLB shootdown* on multicore).

Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.

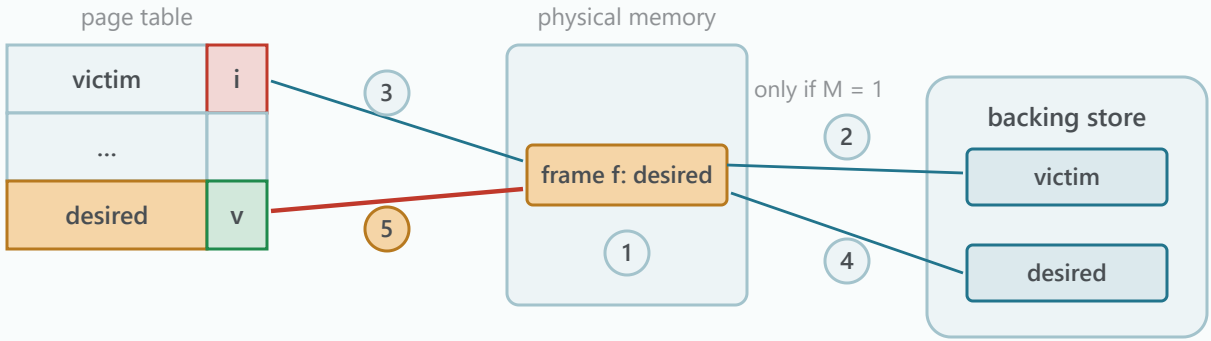


4. The desired page is read into the freed frame.

Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

No Free Frame: Page Replacement

- Memory is usually **full** (free RAM is wasted RAM). Options: kill the process (unacceptable), swap out a whole process, or replace **a single page**.



5. The new entry is set to `v`; the trapping instruction is restarted.

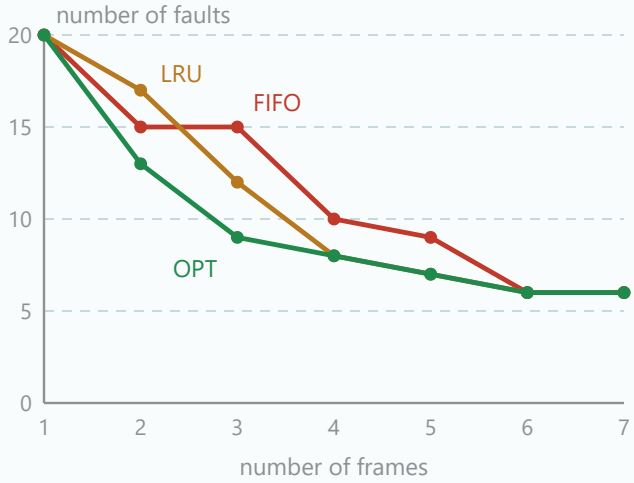
Two decisions: *frame allocation* (how many frames per process?) and *replacement* (which page goes?). Goal: fewest page faults; avoid the same page going back and forth.

Evaluating the Algorithms

- An algorithm is run on a **reference string** (page numbers only) and its **page faults** are counted.
- It is derived from an address trace; consecutive references to the same page count once. With 100-byte pages, `0100 0432 0101 0612 0102 …` → `1 4 1 6 1 6 1 6 1 6 1`.
- This chapter's **common string** (Silberschatz), 3 frames:

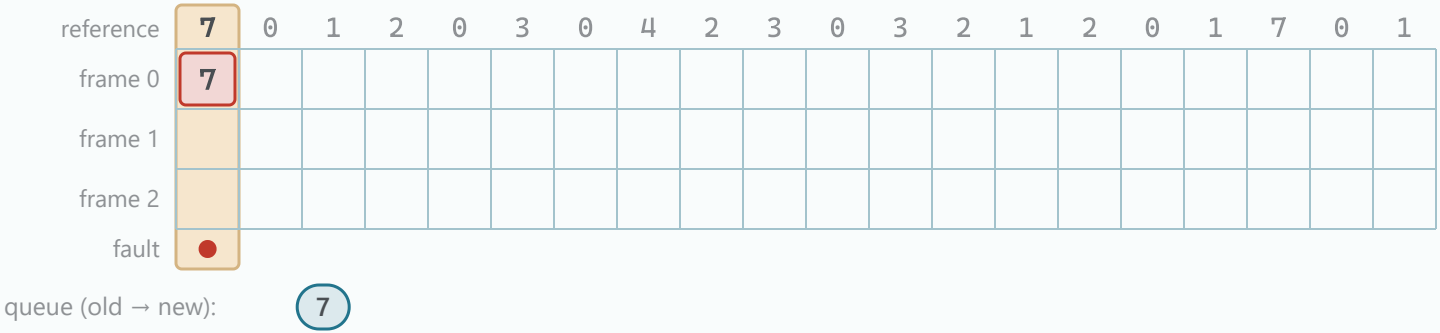
  `7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1`

- Expected: faults **decrease** as frames increase; the initial loads (here 6 distinct pages: 7, 0, 1, 2, 3, 4) are unavoidable.
- Algorithms: OPT, FIFO, LRU, second chance, clock, NRU, NFU/aging, working set, WSClock.



FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.



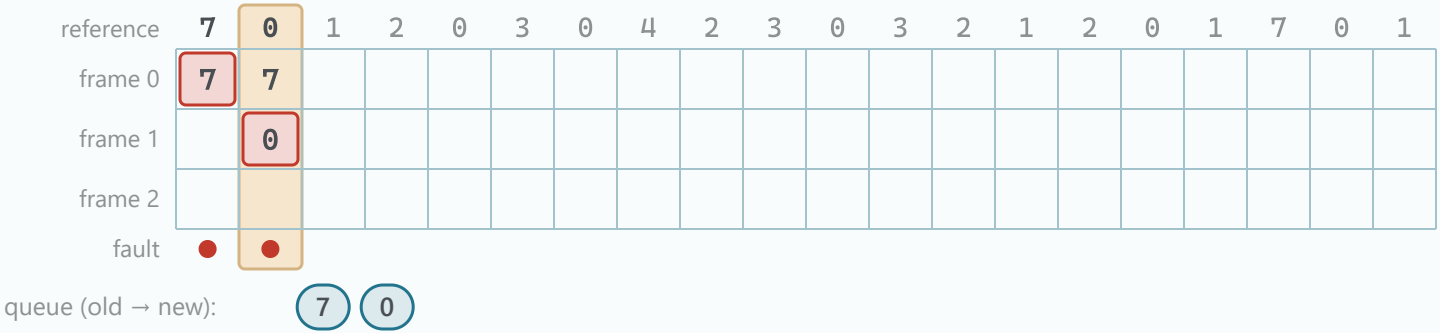
1. reference 7: fault; loaded into free frame 0. Faults: 1

Step 1 / 20

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

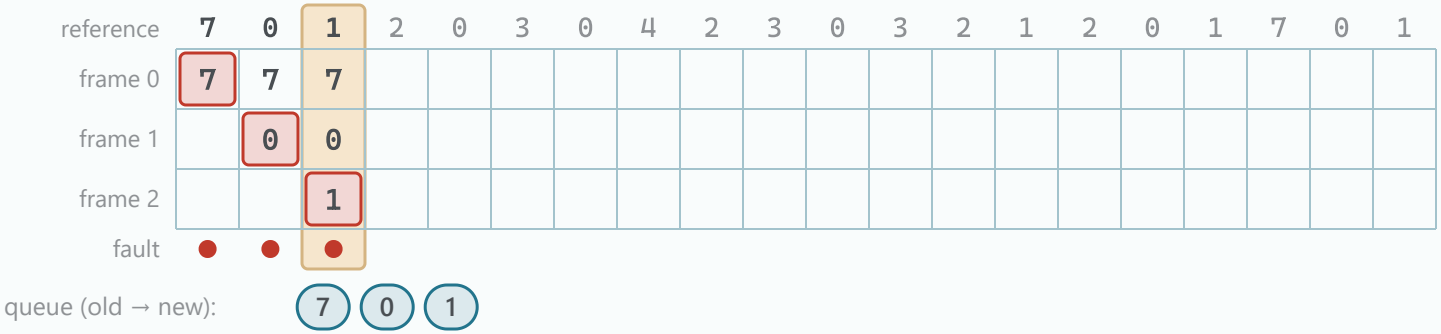


2. reference 0: fault; loaded into free frame 1. Faults: 2

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

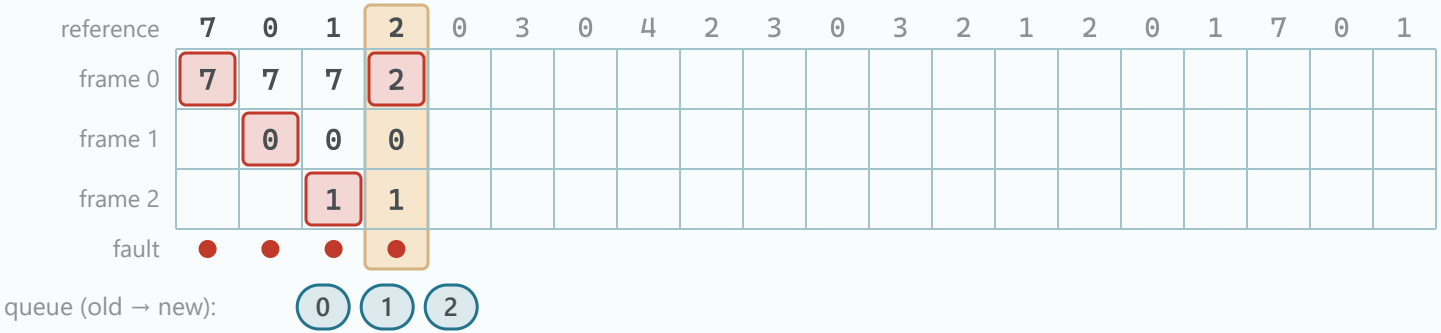


3. reference 1: fault; loaded into free frame 2. Faults: 3

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

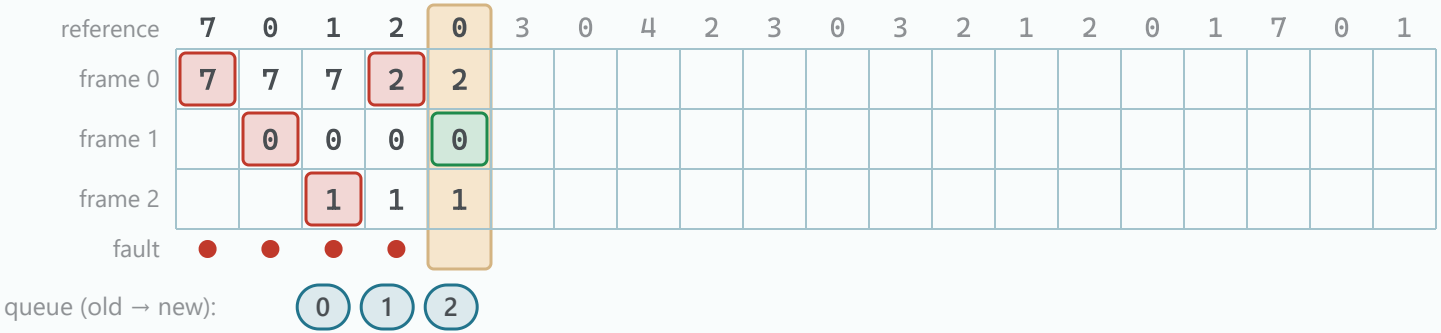


4. reference 2: fault; the oldest page, **7**, leaves the head and 2 joins the tail. Faults: 4

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

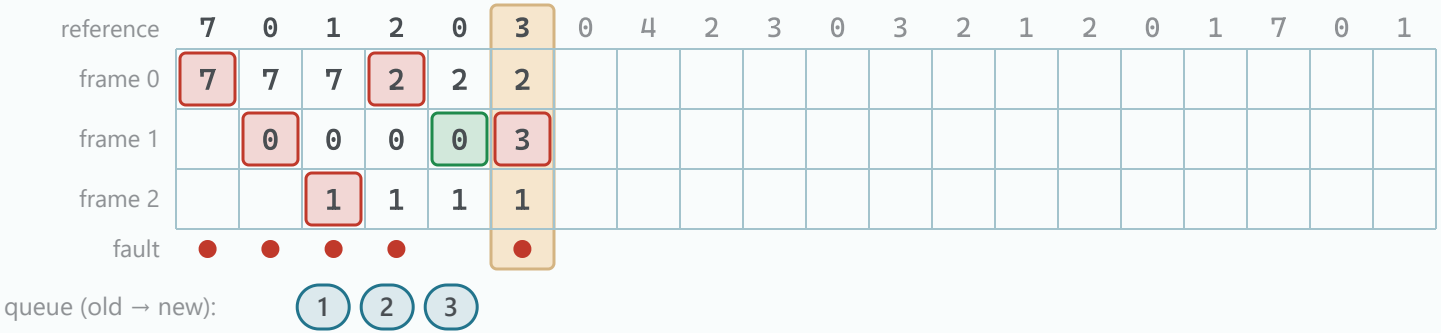


5. reference 0: in memory → hit; the FIFO order is unchanged.

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

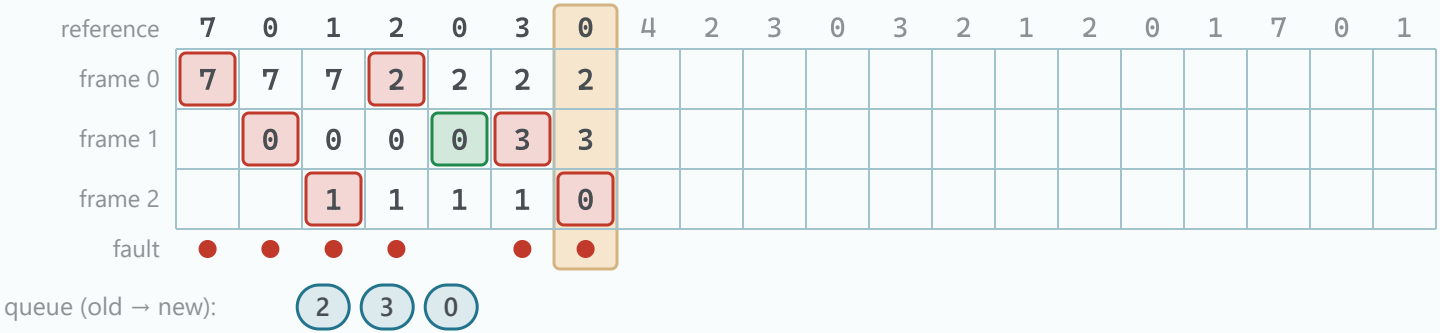


6. reference 3: fault; the oldest page, **0**, leaves the head and 3 joins the tail. Faults: 5

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

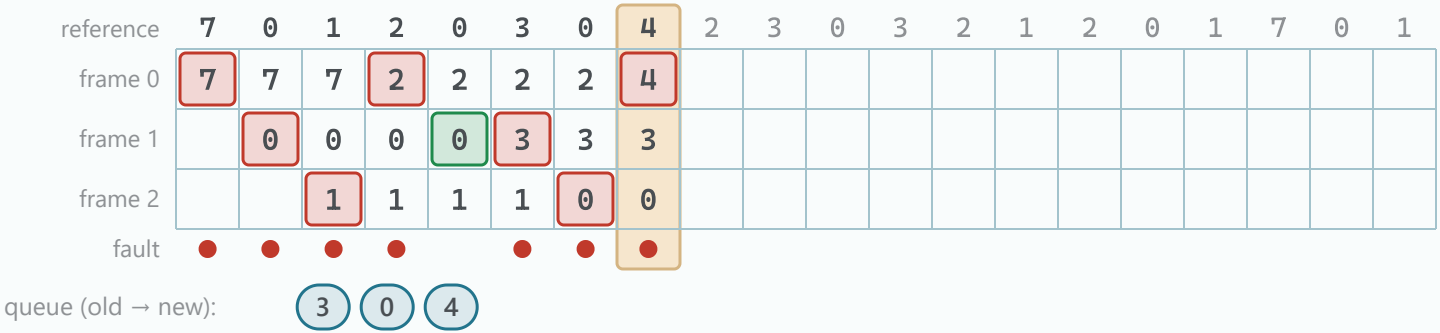


7. reference 0: fault; the oldest page, **1**, leaves the head and 0 joins the tail. Faults: 6

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

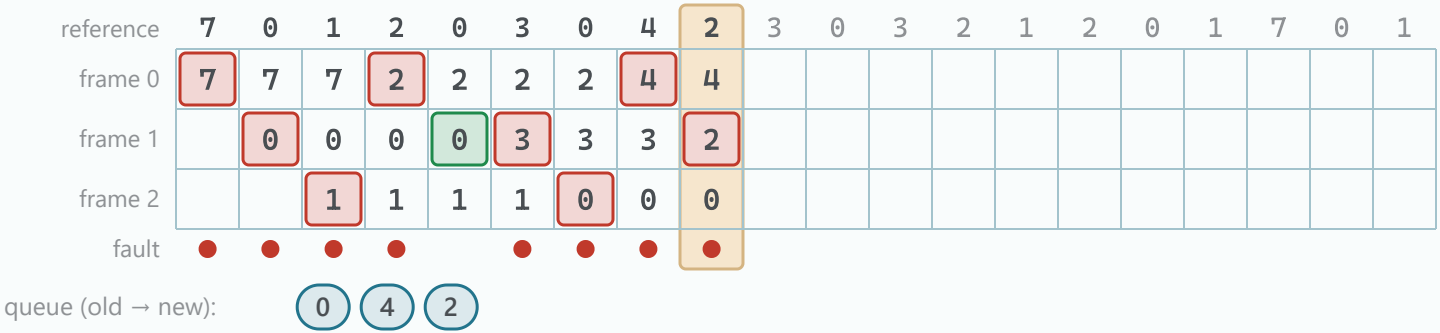


8. reference 4: fault; the oldest page, **2**, leaves the head and 4 joins the tail. Faults: 7

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

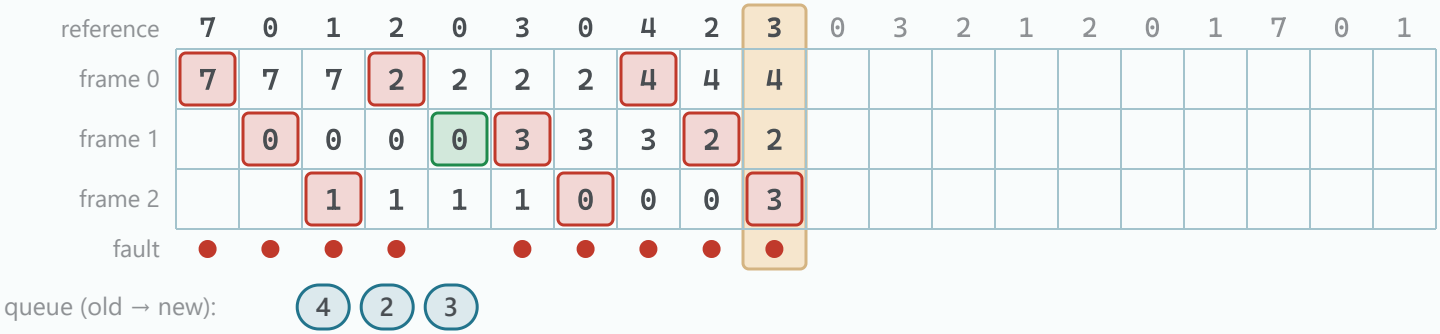


9. reference 2: fault; the oldest page, **3**, leaves the head and 2 joins the tail. Faults: 8

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

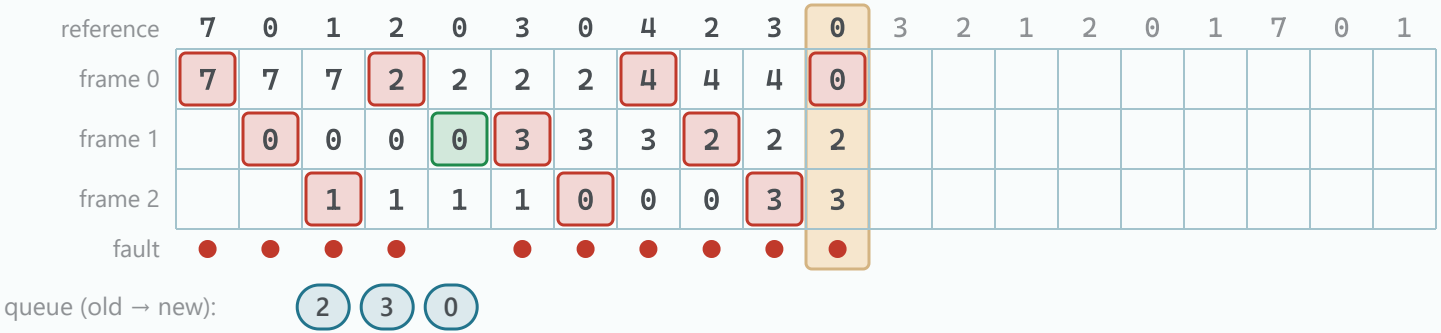


10. reference 3: fault; the oldest page, **0**, leaves the head and 3 joins the tail. Faults: 9

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

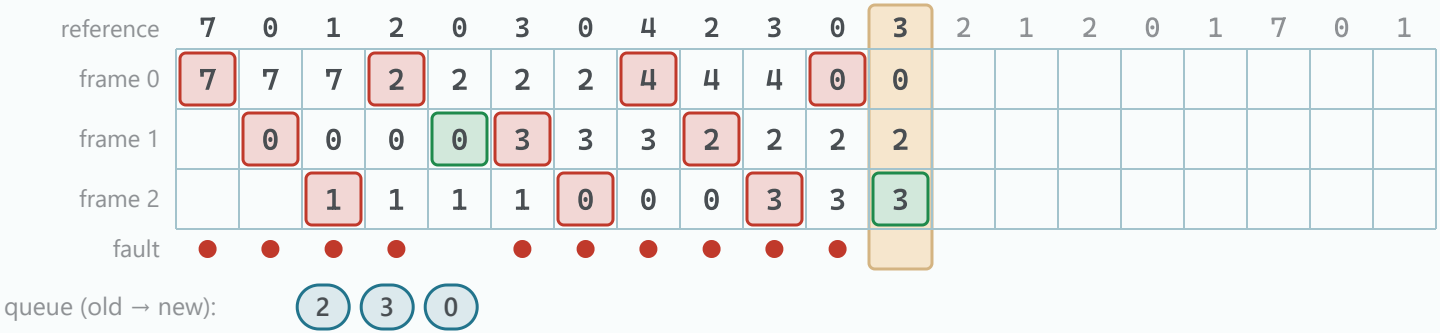


11. reference 0: fault; the oldest page, **4**, leaves the head and 0 joins the tail. Faults: 10

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

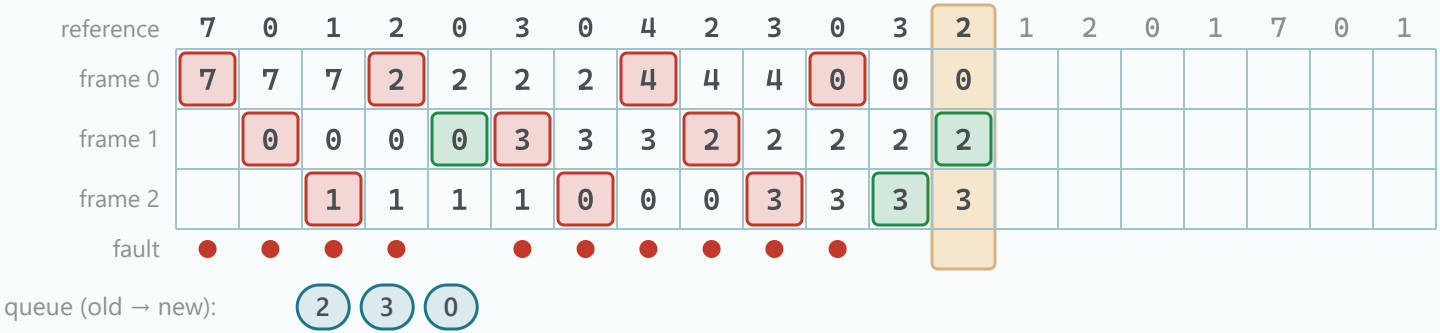


12. reference 3: in memory → hit; the FIFO order is unchanged.

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

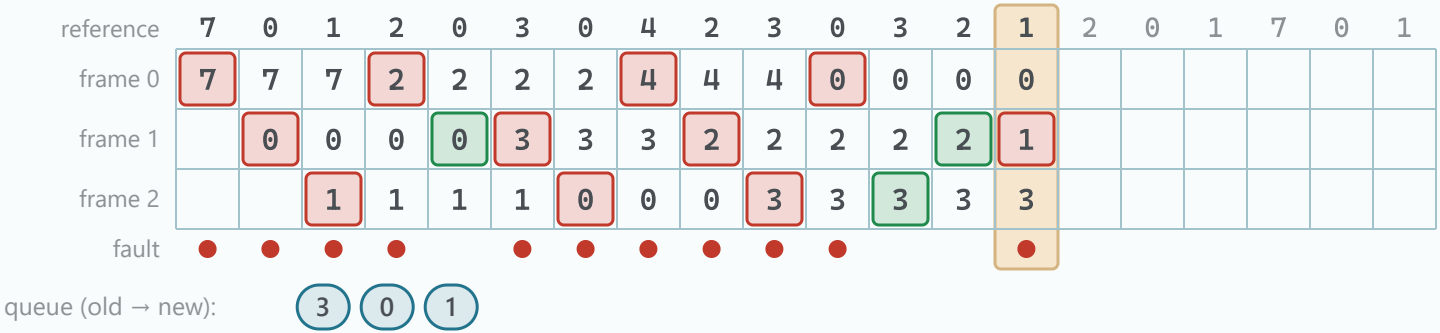


13. reference 2: in memory → hit; the FIFO order is unchanged.

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

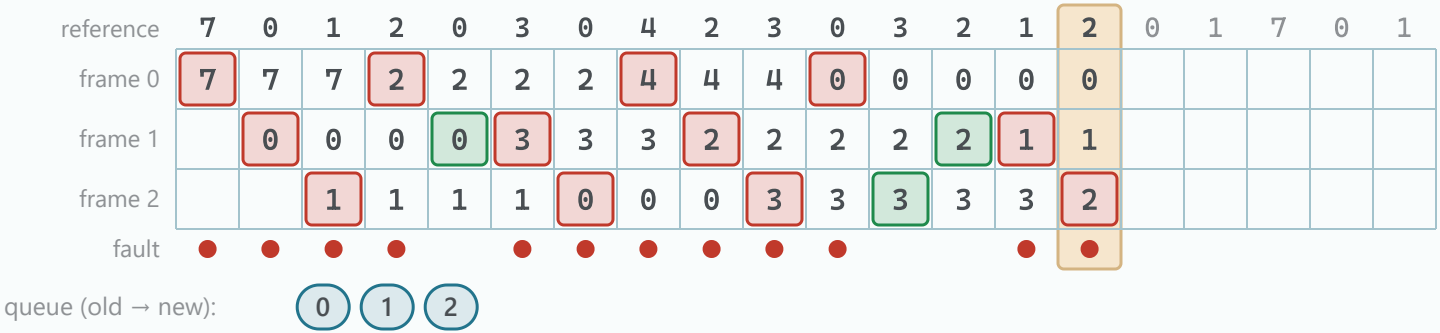


14. reference 1: fault; the oldest page, **2**, leaves the head and 1 joins the tail. Faults: 11

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

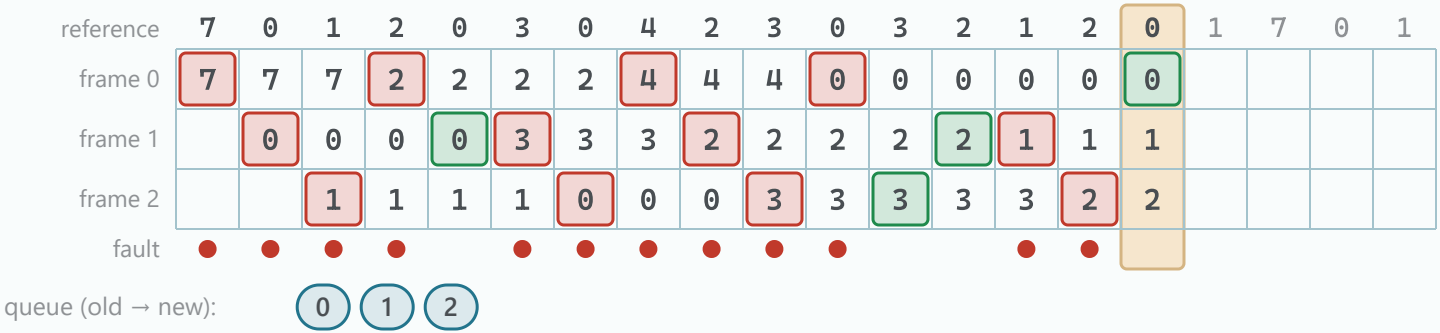


15. reference 2: fault; the oldest page, **3**, leaves the head and 2 joins the tail. Faults: 12

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

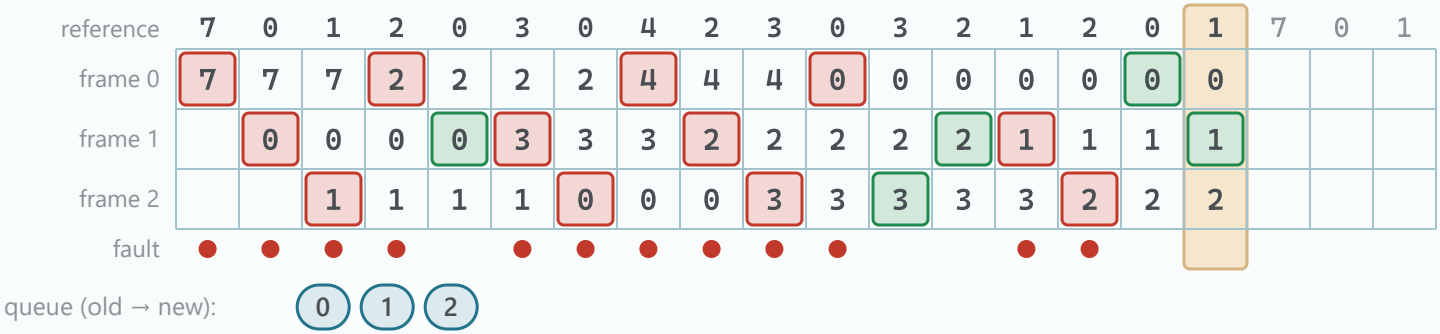


16. reference 0: in memory → hit; the FIFO order is unchanged.

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

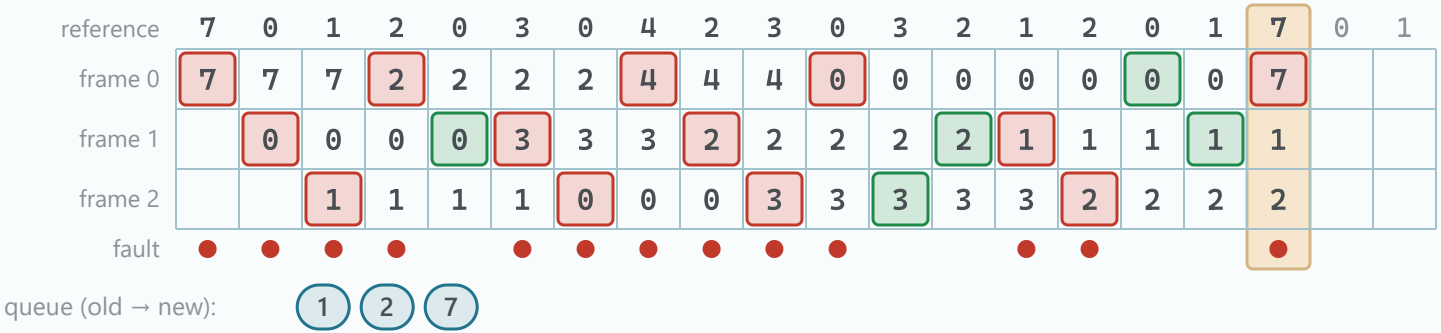


17. reference 1: in memory → hit; the FIFO order is unchanged.

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

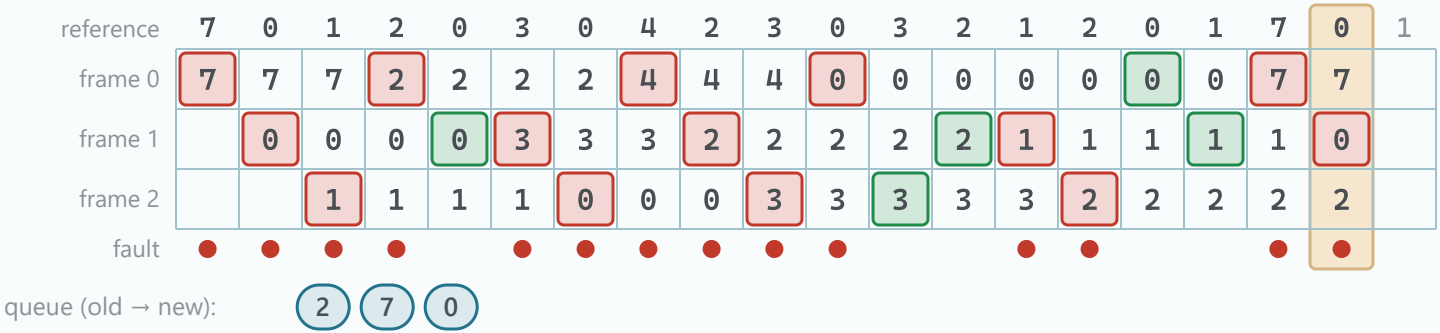


18. reference 7: fault; the oldest page, **0**, leaves the head and 7 joins the tail. Faults: 13

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.

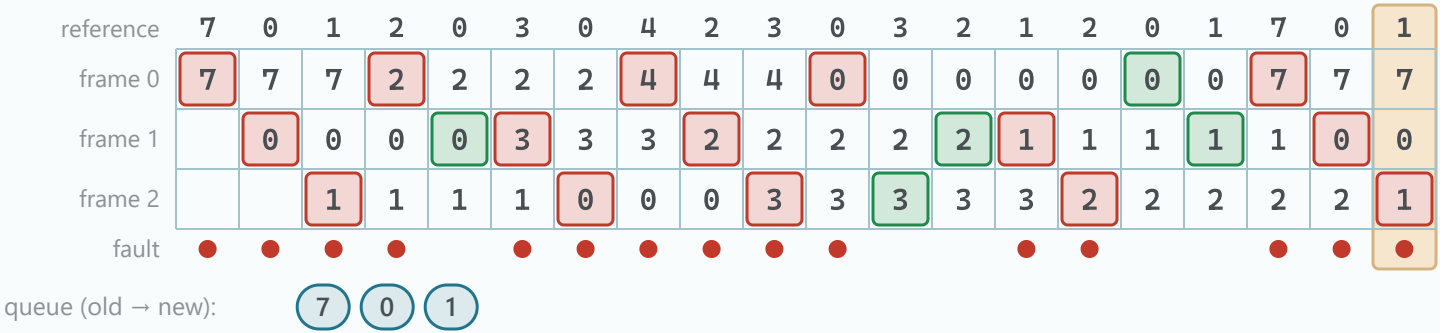


19. reference 0: fault; the oldest page, **1**, leaves the head and 0 joins the tail. Faults: 14

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

FIFO: First In, First Out

- Pages are kept in a queue in **load** order; the most recently *loaded* is at the tail.
- When space is needed, the **oldest** page, at the head of the queue, leaves.
- Easy to implement; needs no bits.
- Ignores usage entirely: the oldest page may be the most heavily used.



20. reference 1: fault; the oldest page, **2**, leaves the head and 1 joins the tail. Faults: 15 **Total: 15 faults** (3 initial loads + 12 replacements).

Step 20 / 20

Red cell: page loaded by a fault · green: hit · red dot: page fault. FIFO tracks loading, not access: the most recently *loaded* page is at the end of the list.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

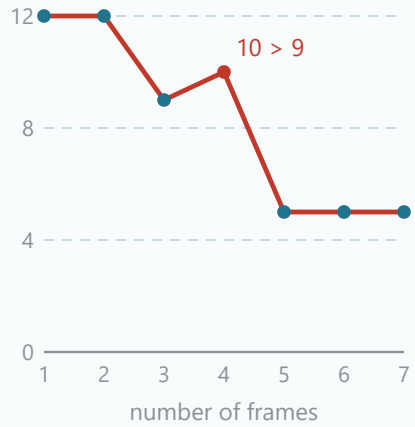
3 frames — faults: 1

	1	2	3	4	1	2	5	1	2	3	4	5
reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1											
frame 1												
frame 2												
fault	●											

4 frames — faults: 1

	1	2	3	4	1	2	5	1	2	3	4	5
reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1											
frame 1												
frame 2												
frame 3												
fault	●											

1. reference 1 — 3 frames: fault · 4 frames: fault.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

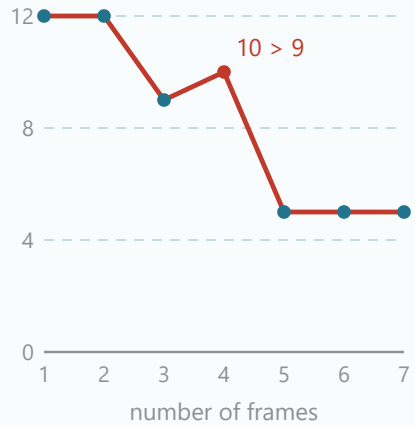
3 frames — faults: 2

	1	2	3	4	1	2	5	1	2	3	4	5
reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1										
frame 1		2										
frame 2												
fault	●	●										

4 frames — faults: 2

	1	2	3	4	1	2	5	1	2	3	4	5
reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1										
frame 1		2										
frame 2												
frame 3												
fault	●	●										

2. reference 2 — 3 frames: fault · 4 frames: fault.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

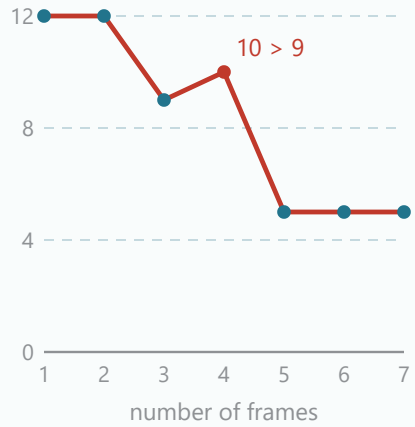
3 frames — faults: 3

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1									
frame 1		2	2									
frame 2			3									
fault	●	●	●									

4 frames — faults: 3

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1									
frame 1		2	2									
frame 2			3									
frame 3												
fault	●	●	●									

3. reference 3 — 3 frames: fault · 4 frames: fault.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

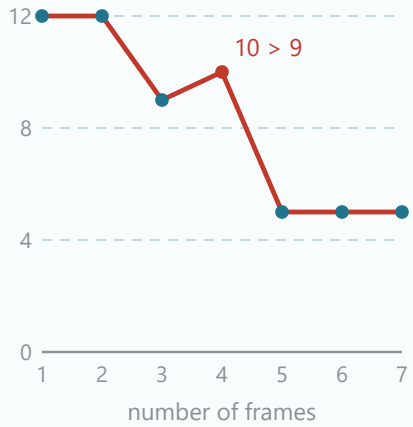
3 frames — faults: 4

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4								
frame 1		2	2	2								
frame 2			3	3								
fault	●	●	●	●								

4 frames — faults: 4

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1								
frame 1		2	2	2								
frame 2			3	3								
frame 3				4								
fault	●	●	●	●								

4. reference 4 — 3 frames: fault (1 out) · 4 frames: fault.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

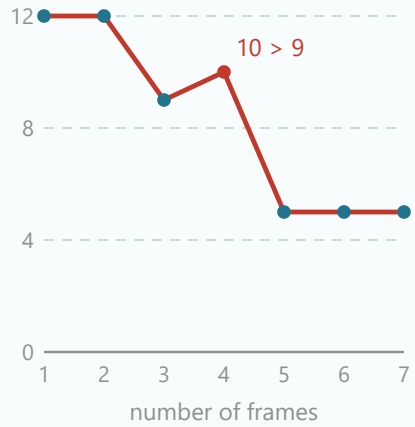
3 frames — faults: 5

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4							
frame 1		2	2	2	1							
frame 2			3	3	3							
fault	●	●	●	●	●							

4 frames — faults: 4

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1							
frame 1		2	2	2	2							
frame 2			3	3	3							
frame 3				4	4							
fault	●	●	●	●								

5. reference 1 — 3 frames: fault (2 out) · 4 frames: hit.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

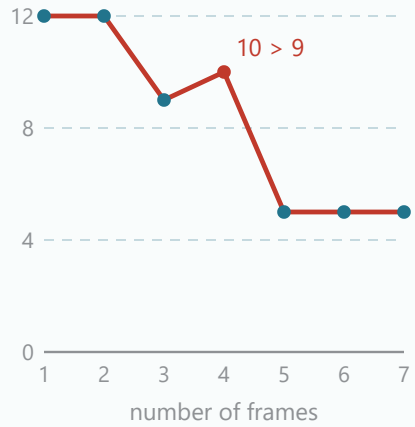
3 frames — faults: 6

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4						
frame 1		2	2	2	1	1						
frame 2			3	3	3	2						
fault	●	●	●	●	●	●						

4 frames — faults: 4

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1						
frame 1		2	2	2	2	2						
frame 2			3	3	3	3						
frame 3				4	4	4						
fault	●	●	●	●								

6. reference 2 — 3 frames: fault (3 out) · 4 frames: hit.



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

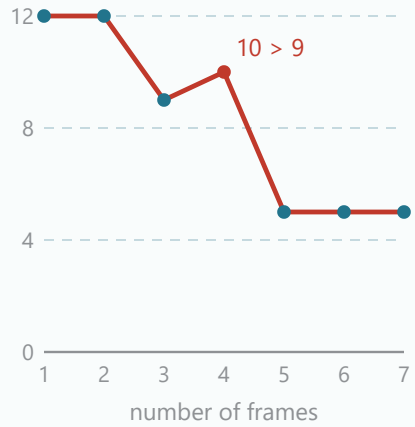
3 frames — faults: 7

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5					
frame 1		2	2	2	1	1	1					
frame 2			3	3	3	2	2					
fault	●	●	●	●	●	●	●					

4 frames — faults: 5

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5					
frame 1		2	2	2	2	2	2					
frame 2			3	3	3	3	3					
frame 3				4	4	4	4					
fault	●	●	●	●			●					

7. reference 5 — 3 frames: fault (4 out) · 4 frames: fault (1 out).



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

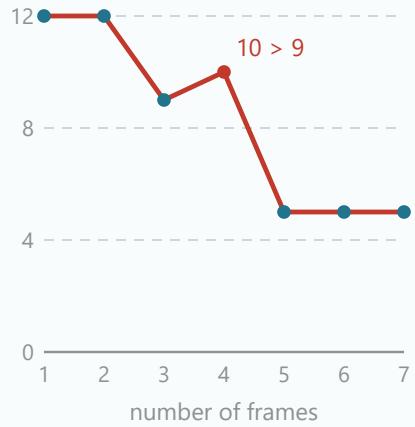
3 frames — faults: 7

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5				
frame 1		2	2	2	1	1	1	1				
frame 2			3	3	3	2	2	2				
fault	●	●	●	●	●	●	●					

4 frames — faults: 6

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5				
frame 1		2	2	2	2	2	2	1				
frame 2			3	3	3	3	3	3				
frame 3				4	4	4	4	4				
fault	●	●	●	●			●	●				

8. reference 1 — 3 frames: hit · 4 frames: fault (2 out).



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

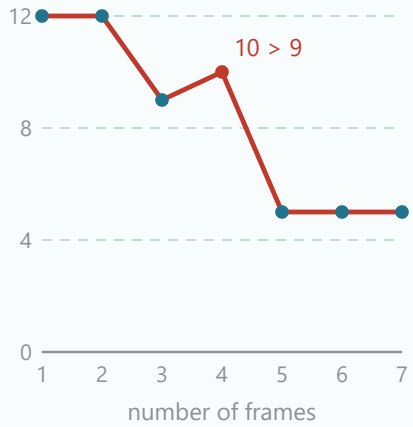
3 frames — faults: 7

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5			
frame 1		2	2	2	1	1	1	1	1			
frame 2			3	3	3	2	2	2	2			
fault	●	●	●	●	●	●	●					

4 frames — faults: 7

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5	5			
frame 1		2	2	2	2	2	2	1	1			
frame 2			3	3	3	3	3	3	2			
frame 3				4	4	4	4	4	4			
fault	●	●	●	●			●	●	●			

9. reference 2 — 3 frames: hit · 4 frames: fault (3 out).



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: 1 2 3 4 1 2 5 1 2 3 4 5

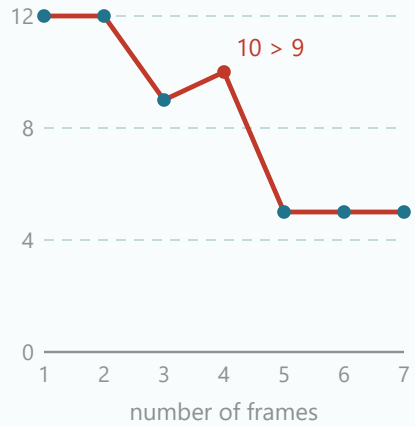
3 frames — faults: 8

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5		
frame 1		2	2	2	1	1	1	1	1	3		
frame 2			3	3	3	2	2	2	2	2		
fault	●	●	●	●	●	●	●			●		

4 frames — faults: 8

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5	5	5		
frame 1		2	2	2	2	2	2	1	1	1		
frame 2			3	3	3	3	3	3	2	2		
frame 3				4	4	4	4	4	4	3		
fault	●	●	●	●			●	●	●	●		

10. reference 3 — 3 frames: fault (1 out) · 4 frames: fault (4 out).



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

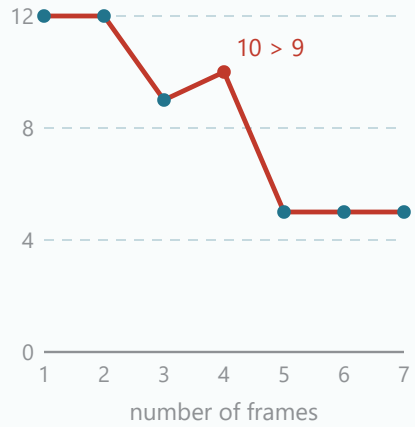
3 frames — faults: 9

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5	5	
frame 1		2	2	2	1	1	1	1	1	3	3	
frame 2			3	3	3	2	2	2	2	2	4	
fault	●	●	●	●	●	●	●			●	●	

4 frames — faults: 9

reference	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5	5	5	4	
frame 1		2	2	2	2	2	2	1	1	1	1	
frame 2			3	3	3	3	3	3	2	2	2	
frame 3				4	4	4	4	4	4	3	3	
fault	●	●	●	●			●	●	●	●	●	

11. reference 4 — 3 frames: fault (2 out) · 4 frames: fault (5 out).



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

Belady's Anomaly

More frames should mean fewer faults… not always with FIFO. String: `1 2 3 4 1 2 5 1 2 3 4 5`

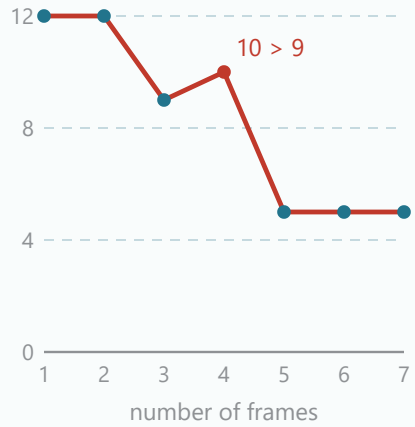
3 frames — faults: 9

	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5	5	5
frame 1		2	2	2	1	1	1	1	1	3	3	3
frame 2			3	3	3	2	2	2	2	2	4	4
fault	●	●	●	●	●	●	●			●	●	

4 frames — faults: 10

	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	1	1	1	5	5	5	5	4	4
frame 1		2	2	2	2	2	2	1	1	1	1	5
frame 2			3	3	3	3	3	3	2	2	2	2
frame 3				4	4	4	4	4	4	3	3	3
fault	●	●	●	●			●	●	●	●	●	●

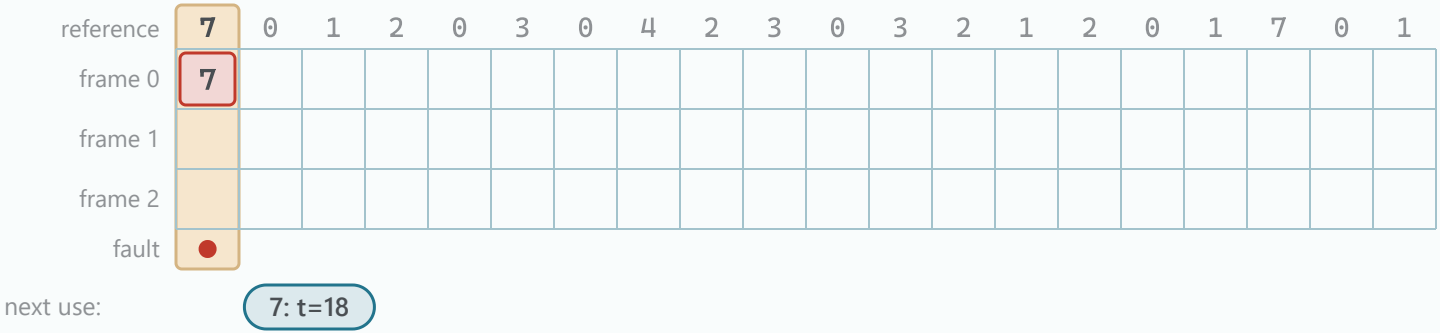
12. reference 5 — 3 frames: hit · 4 frames: fault (1 out). **Result: 9 < 10.**



Stack algorithms (OPT, LRU): the set of pages in n frames is always a subset of the set in n+1 frames → the anomaly **cannot occur**.

OPT: Replace the Page Not Used for the Longest Time

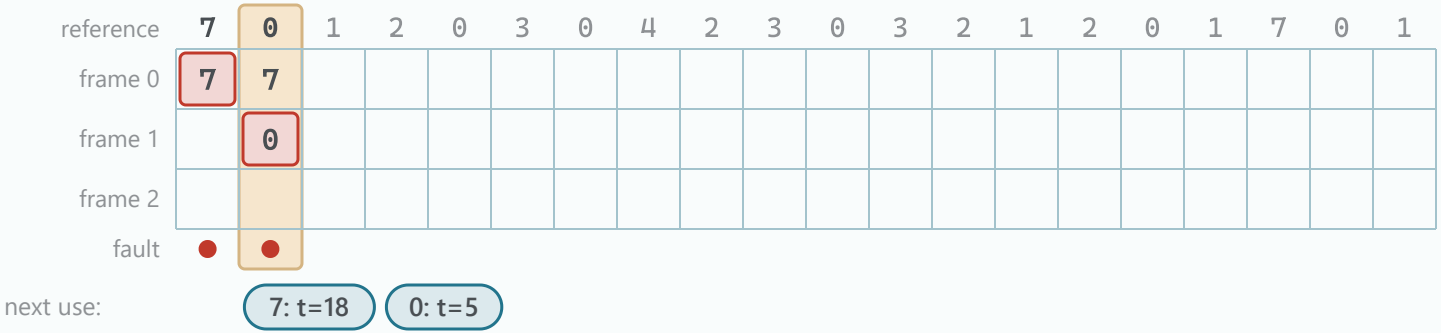
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



1. reference 7: fault; loaded into free frame 0. Faults: 1

OPT: Replace the Page Not Used for the Longest Time

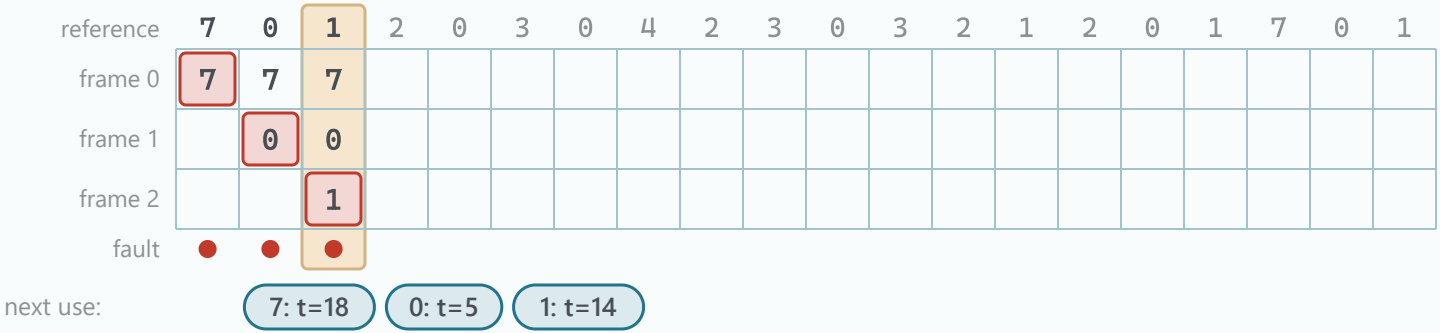
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



2. reference 0: fault; loaded into free frame 1. Faults: 2

OPT: Replace the Page Not Used for the Longest Time

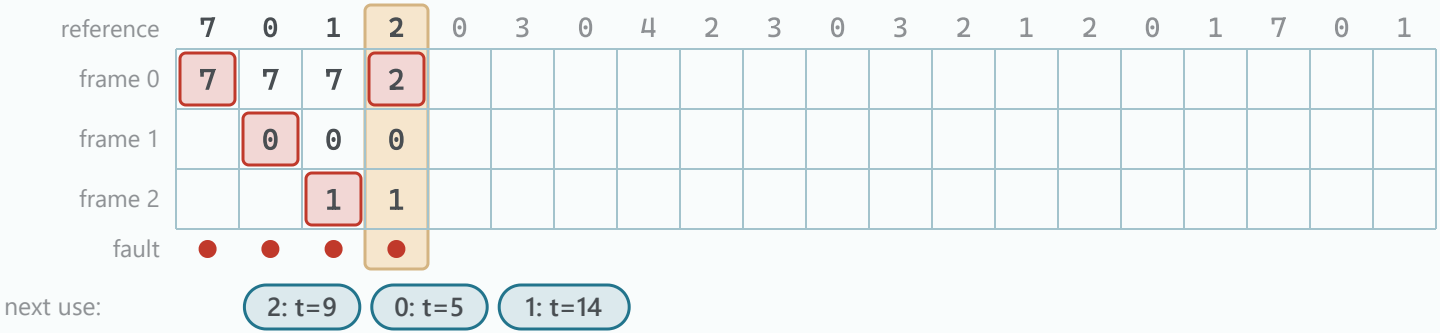
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



3. reference 1: fault; loaded into free frame 2. Faults: 3

OPT: Replace the Page Not Used for the Longest Time

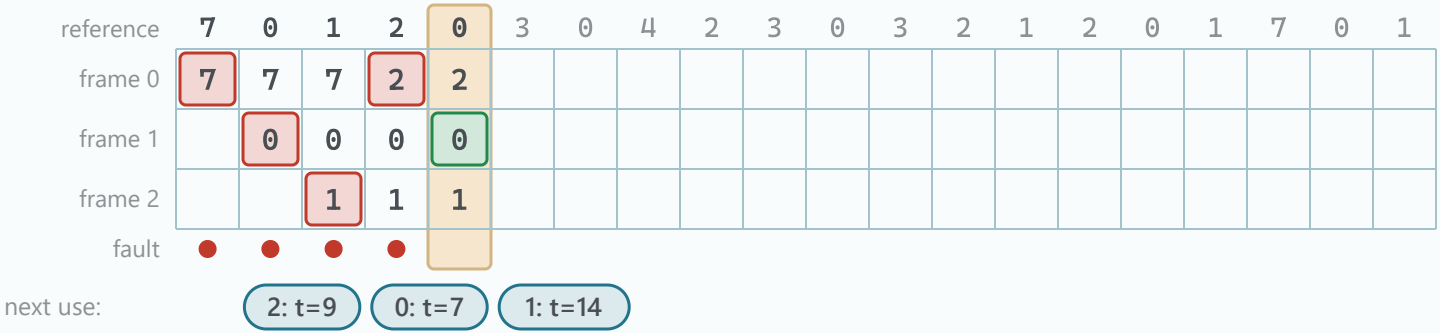
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



4. reference 2: fault; next uses (7→18, 0→5, 1→14); the farthest, **7**, leaves. Faults: 4

OPT: Replace the Page Not Used for the Longest Time

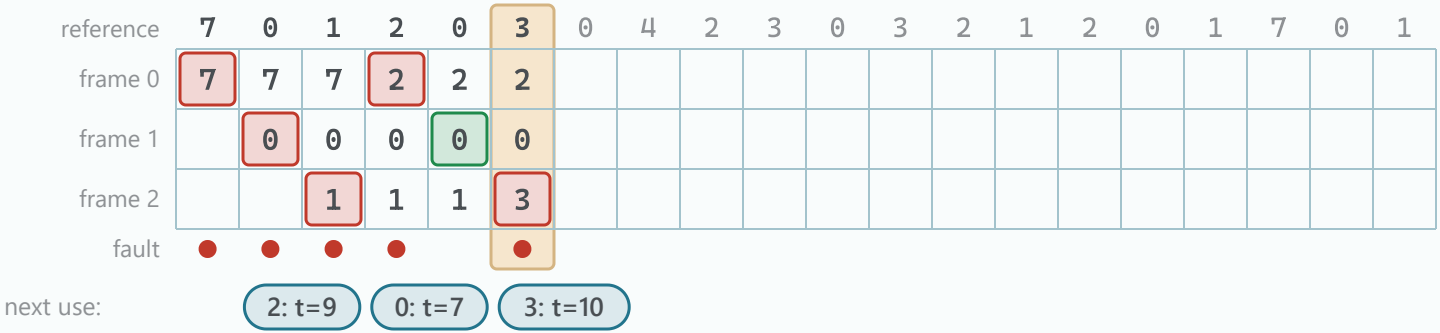
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



5. reference 0: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

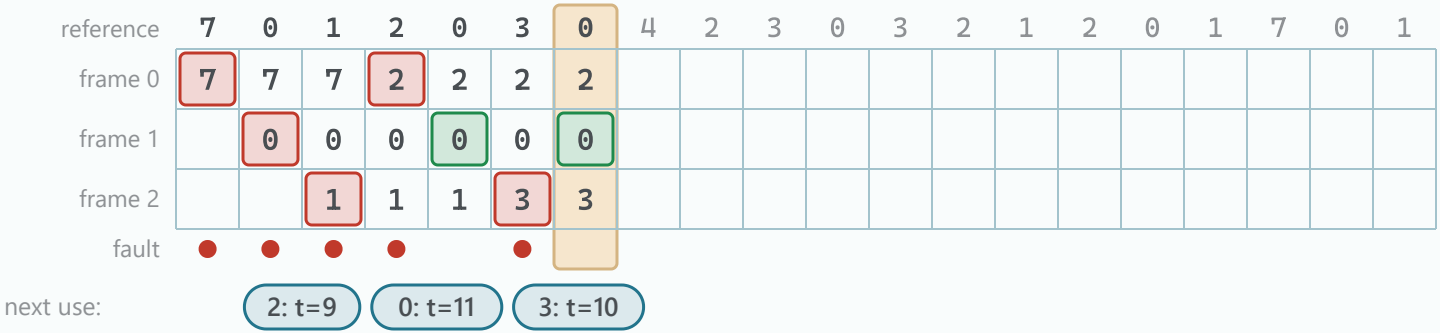
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



6. reference 3: fault; next uses (2→9, 0→7, 1→14); the farthest, **1**, leaves. Faults: 5

OPT: Replace the Page Not Used for the Longest Time

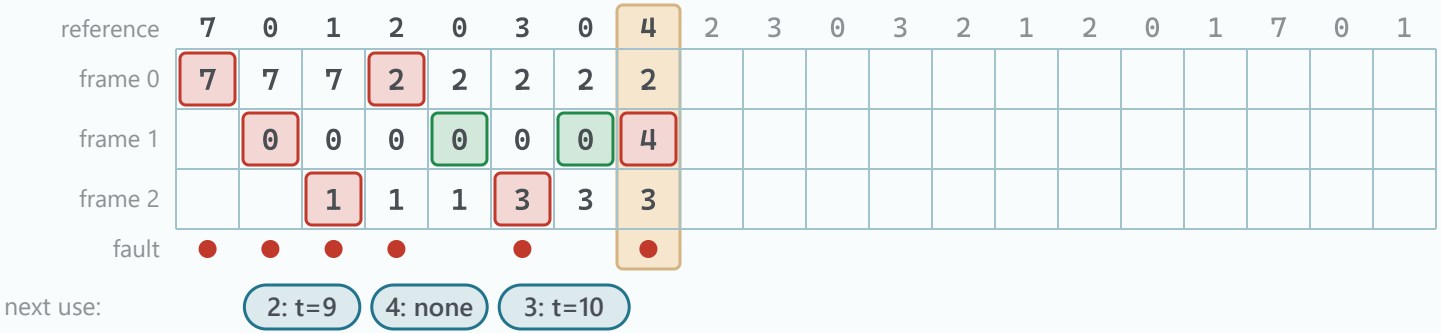
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



7. reference 0: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

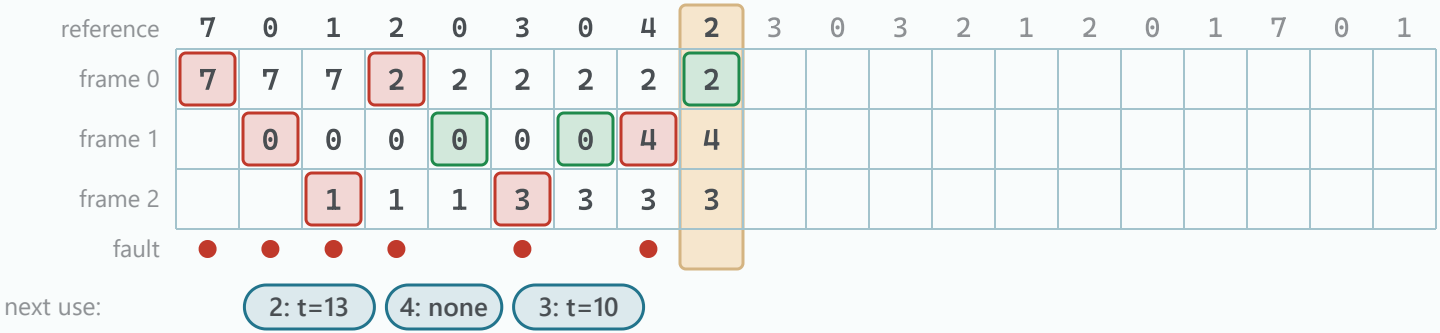
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



8. reference 4: fault; next uses (2→9, 0→11, 3→10); the farthest, **0**, leaves. Faults: 6

OPT: Replace the Page Not Used for the Longest Time

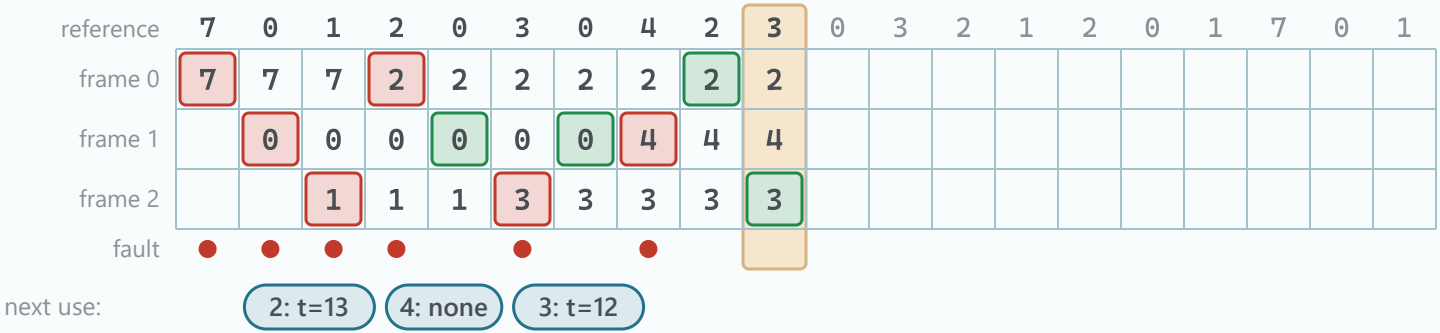
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



9. reference 2: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

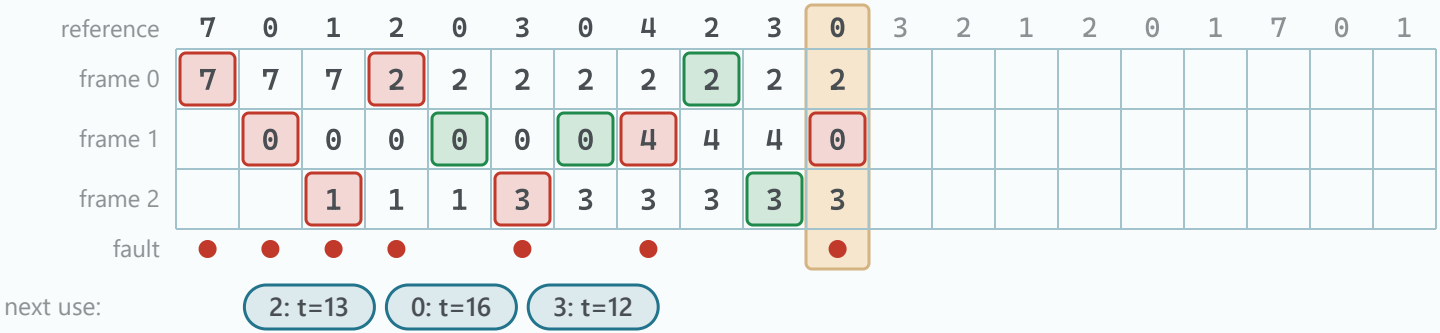
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



10. reference 3: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

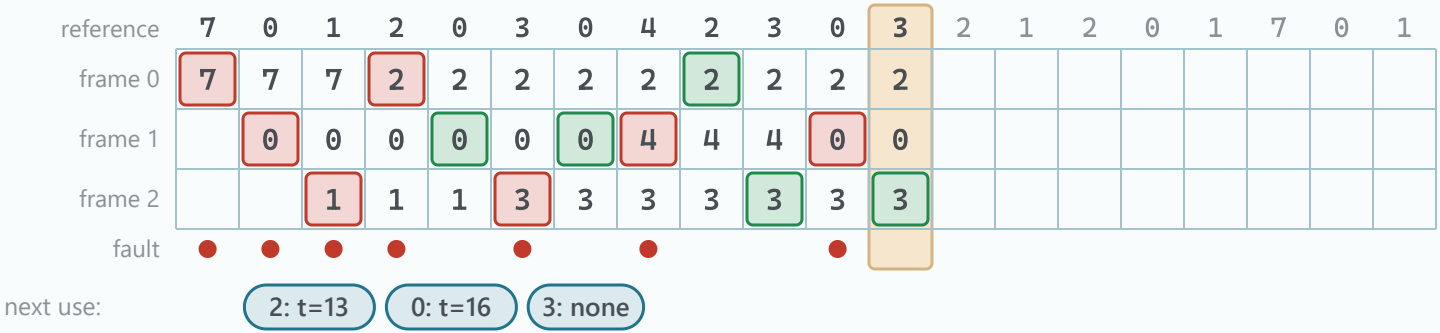
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



11. reference 0: fault; next uses (2→13, 4→none, 3→12); the farthest, **4**, leaves. Faults: 7

OPT: Replace the Page Not Used for the Longest Time

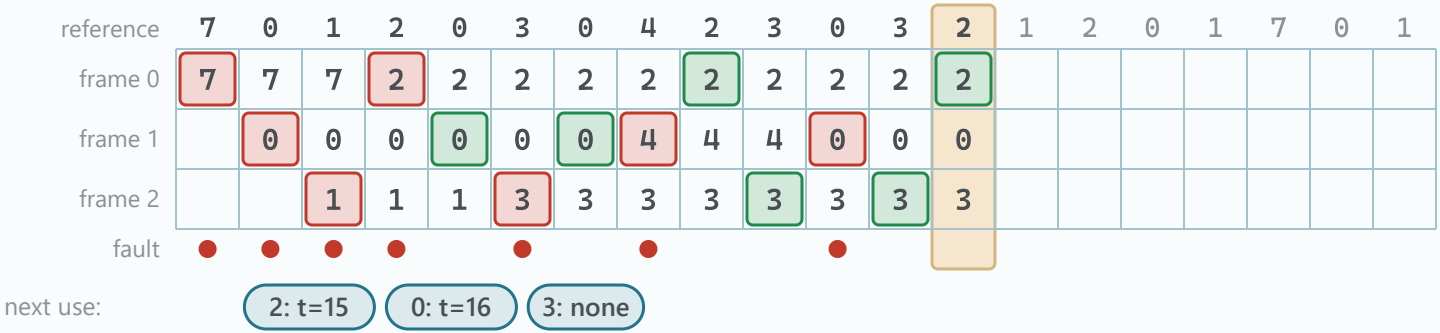
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



12. reference 3: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

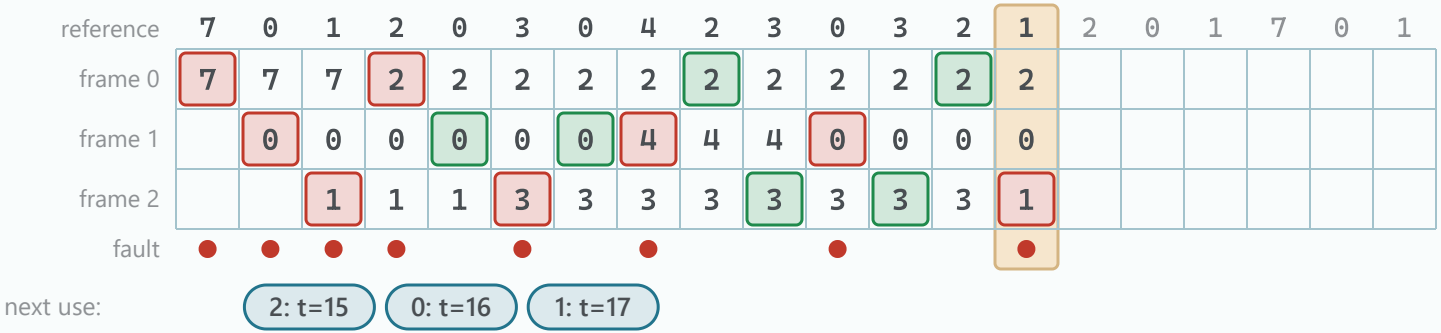
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



13. reference 2: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

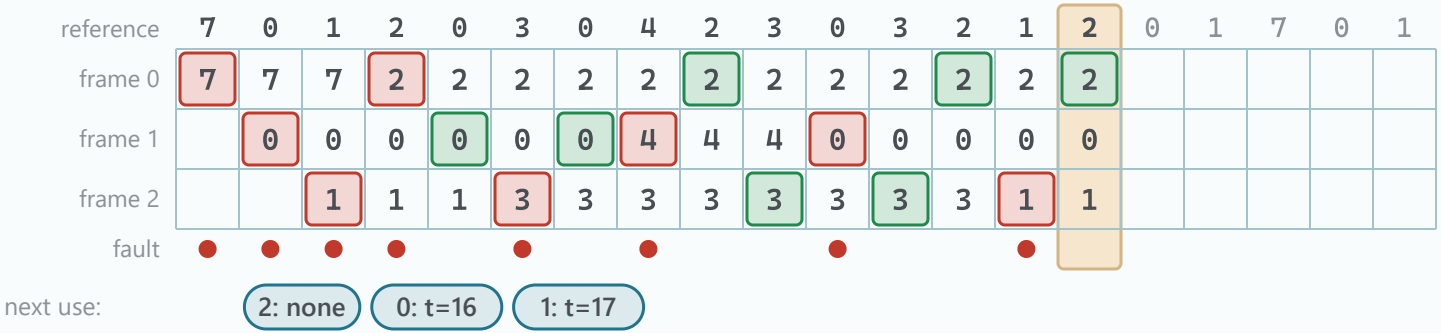
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



14. reference 1: fault; next uses (2→15, 0→16, 3→none); the farthest, **3**, leaves. Faults: 8

OPT: Replace the Page Not Used for the Longest Time

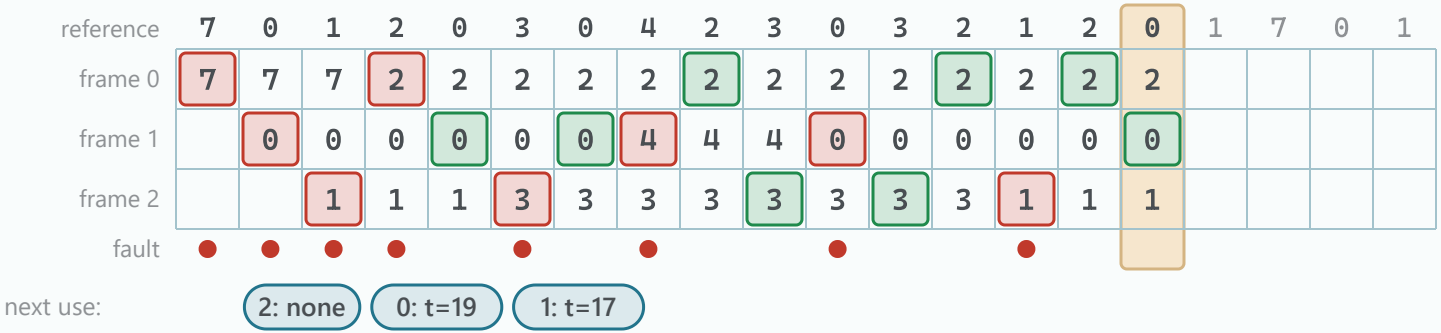
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



15. reference 2: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

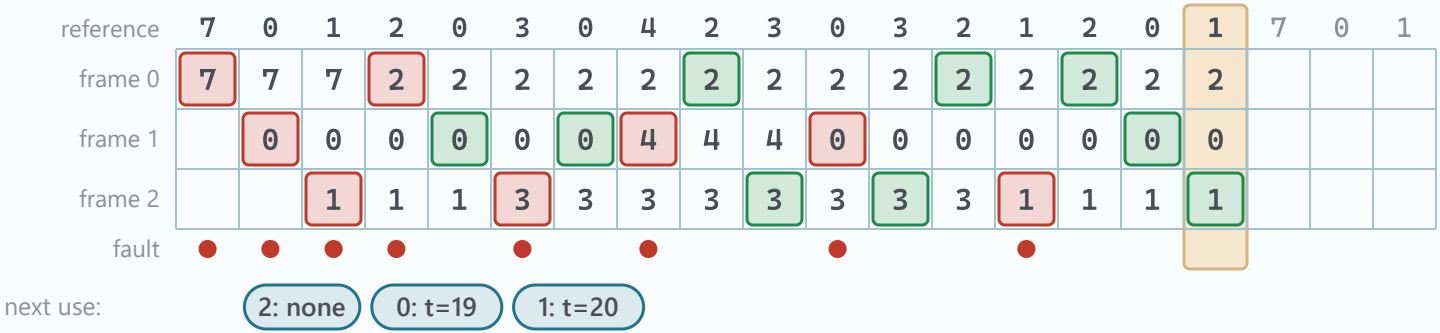
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



16. reference 0: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

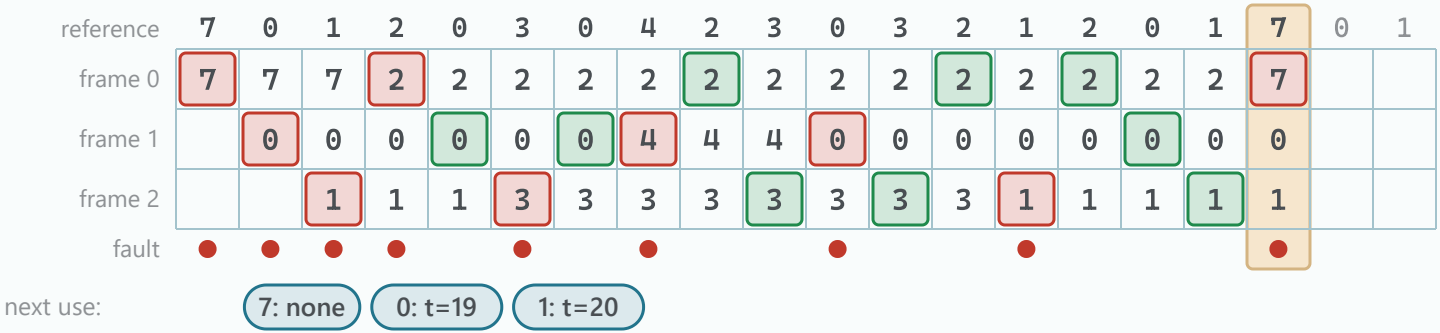
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



17. reference 1: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

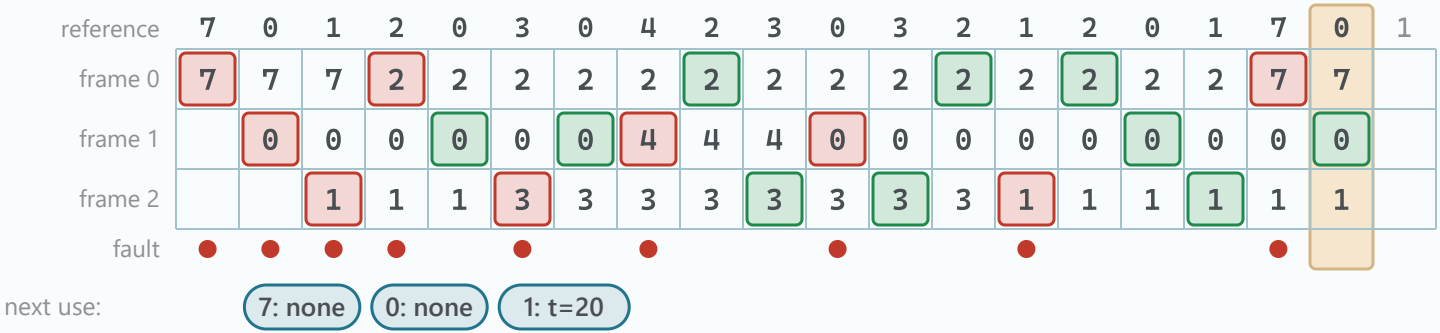
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



18. reference 7: fault; next uses (2→none, 0→19, 1→20); the farthest, **2**, leaves. Faults: 9

OPT: Replace the Page Not Used for the Longest Time

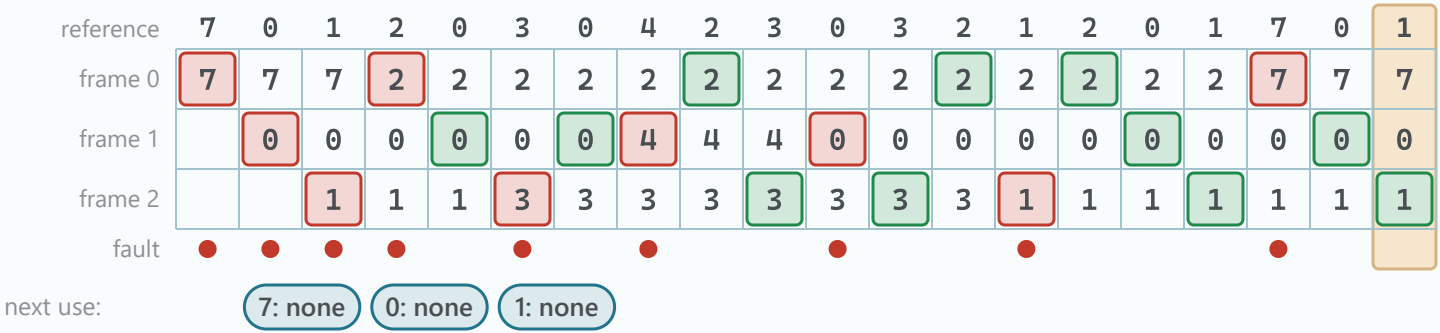
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



19. reference 0: in memory → hit.

OPT: Replace the Page Not Used for the Longest Time

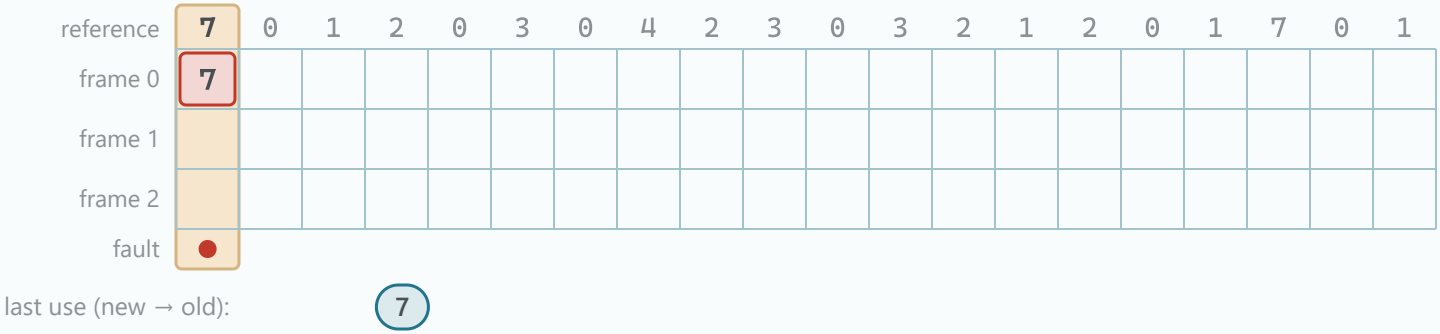
- Replace the page that will be used **furthest in the future** (or never) (Belady, 1966).
- Easy to define, impossible to implement: the OS can't know the future (a crystal ball).
- It is a **lower bound**: used to compare other algorithms.
- On a real system it can be computed only by running a trace twice.



20. reference 1: in memory → hit. **Total: 9 faults** (6 replacements): the lower bound for this string and 3 frames.

LRU: Least Recently Used

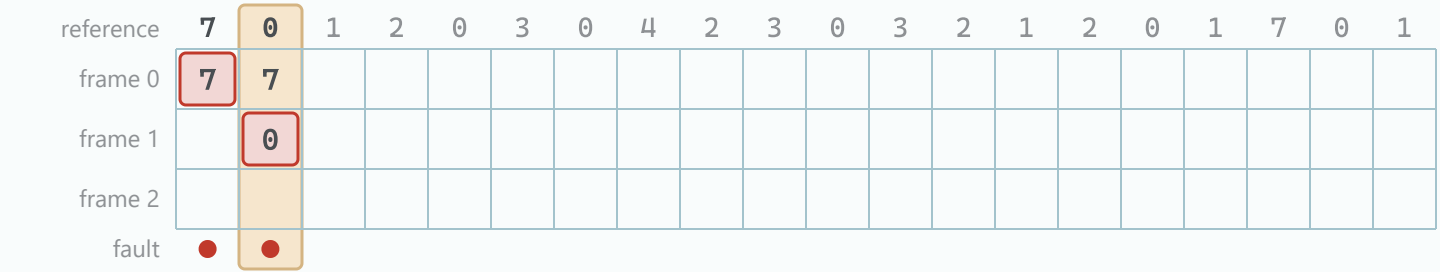
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



1. reference 7: fault; loaded into free frame 0. Faults: 1

LRU: Least Recently Used

- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.

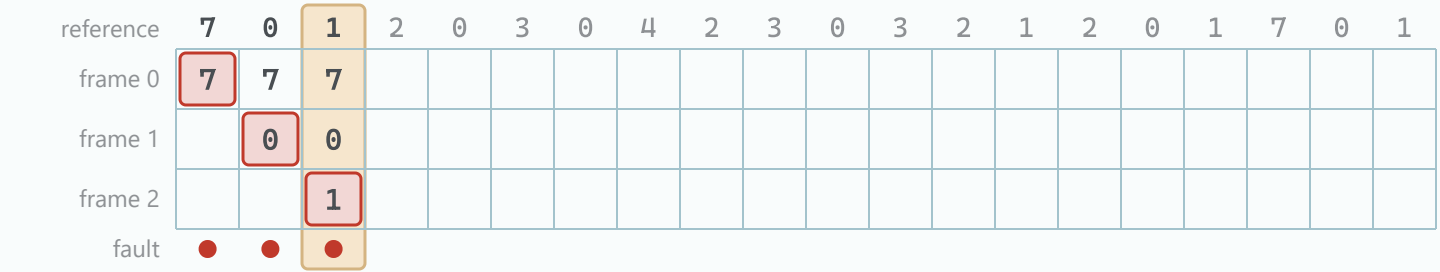


last use (new → old): 0 7

2. reference 0: fault; loaded into free frame 1. Faults: 2

LRU: Least Recently Used

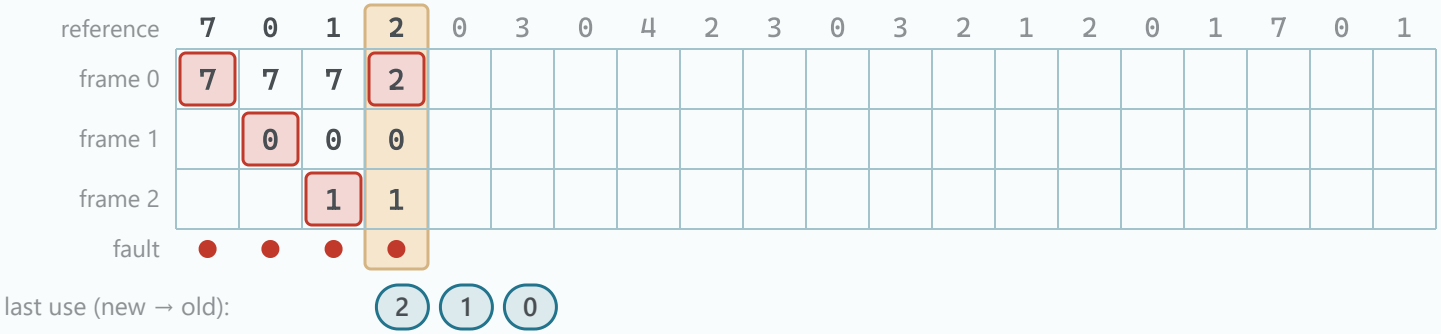
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



3. reference 1: fault; loaded into free frame 2. Faults: 3

LRU: Least Recently Used

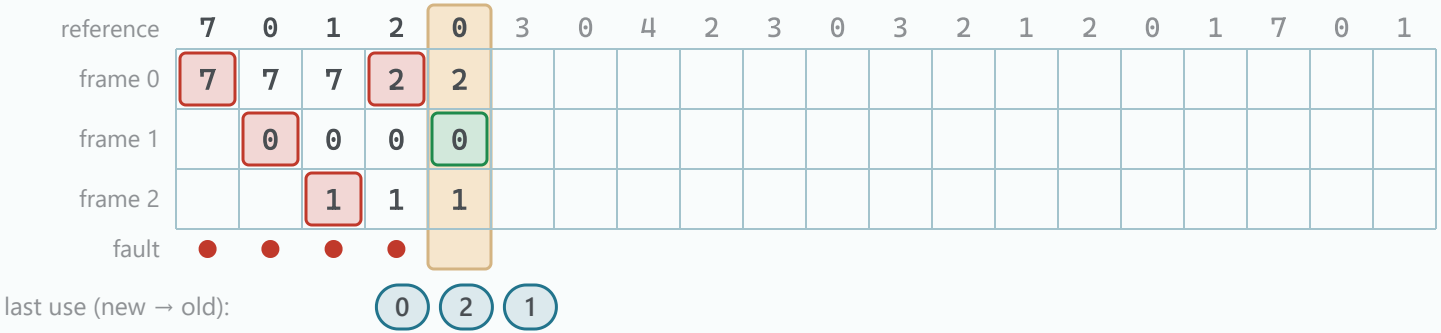
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



4. reference 2: fault; the least recently used, **7**, leaves. Faults: 4

LRU: Least Recently Used

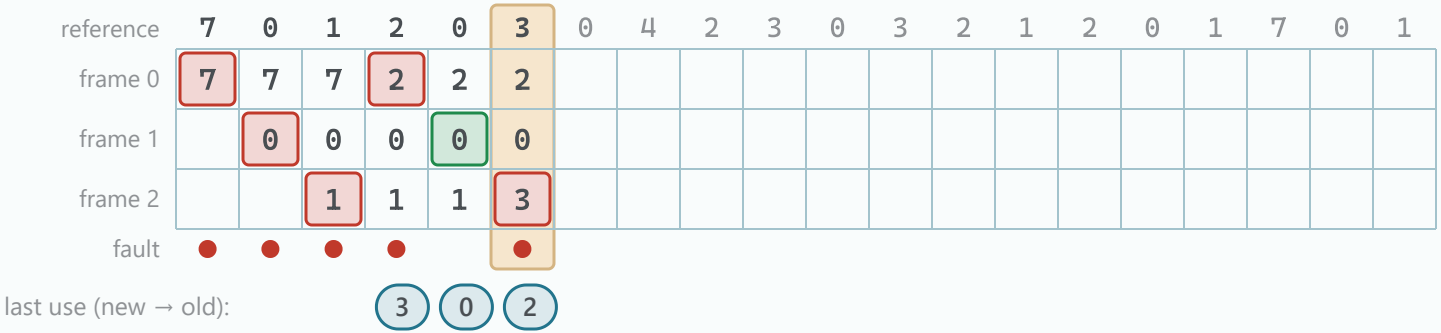
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



5. reference 0: in memory → hit; 0 becomes the most recent.

LRU: Least Recently Used

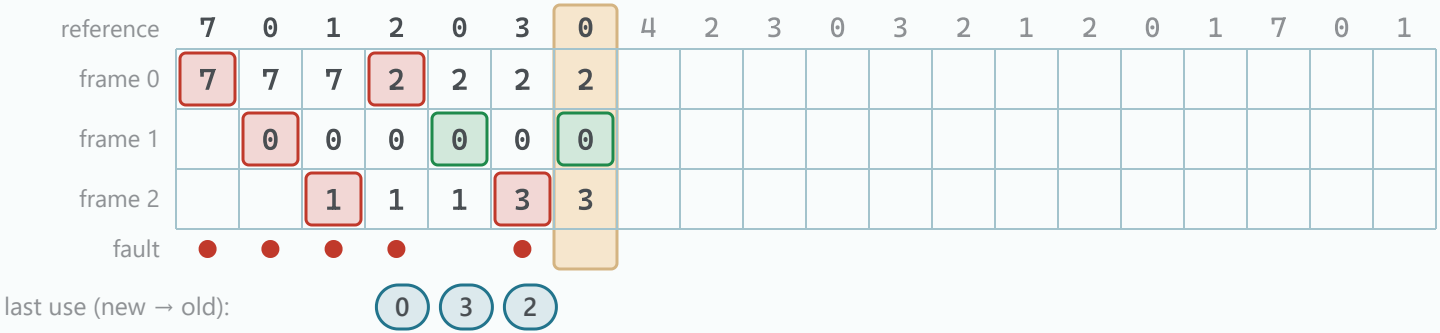
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



6. reference 3: fault; the least recently used, **1**, leaves. Faults: 5

LRU: Least Recently Used

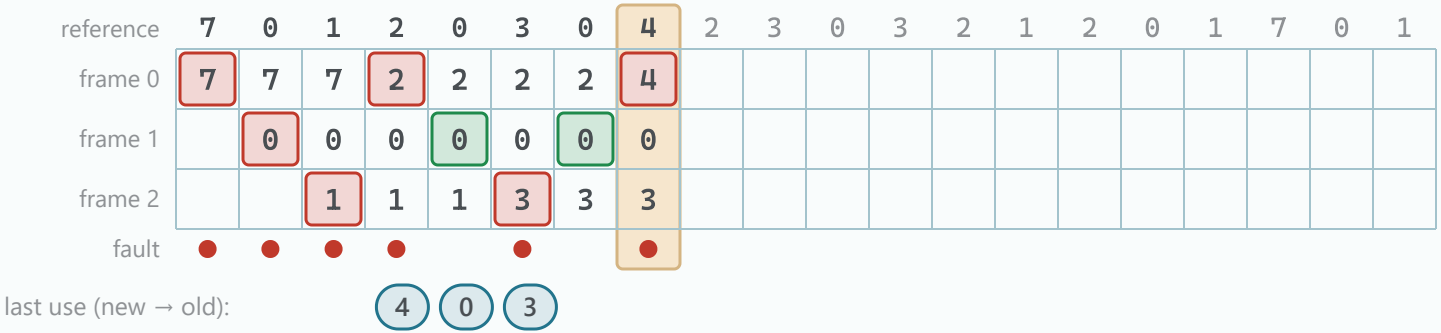
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



7. reference 0: in memory → hit; 0 becomes the most recent.

LRU: Least Recently Used

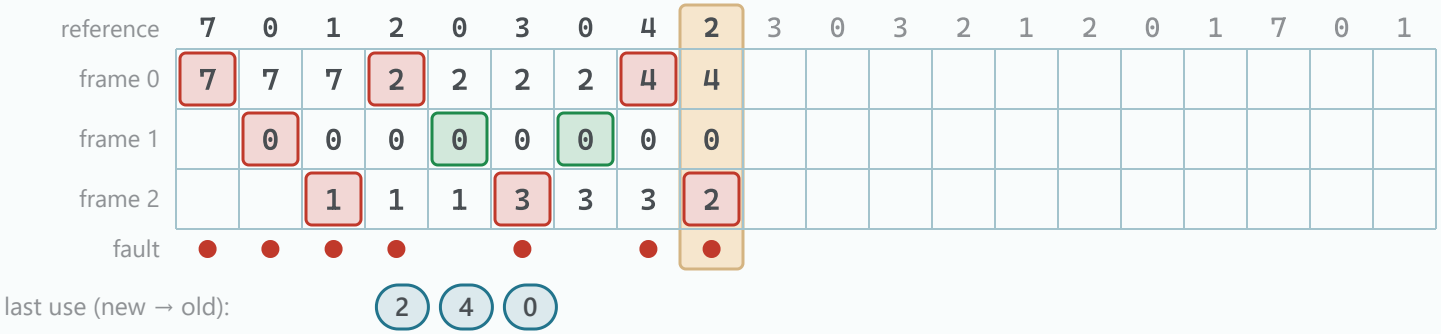
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



8. reference 4: fault; the least recently used, **2**, leaves. Faults: 6

LRU: Least Recently Used

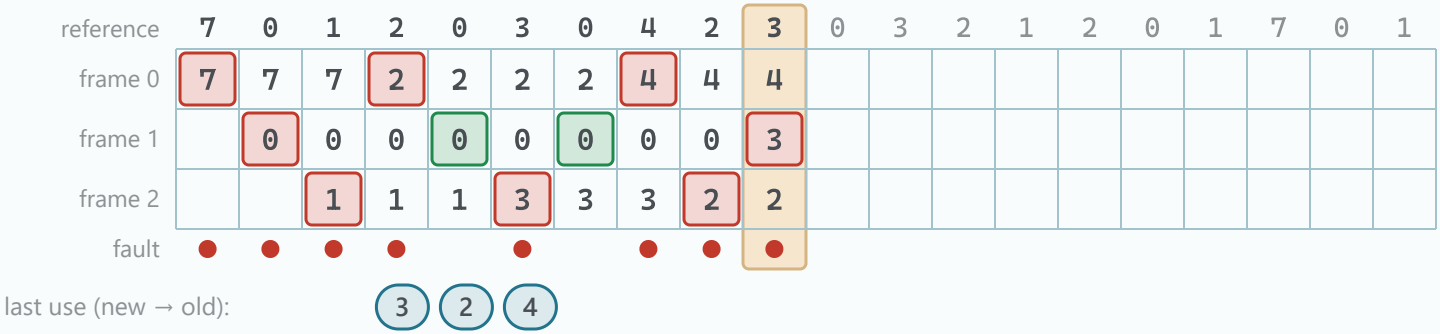
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



9. reference 2: fault; the least recently used, **3**, leaves. Faults: 7

LRU: Least Recently Used

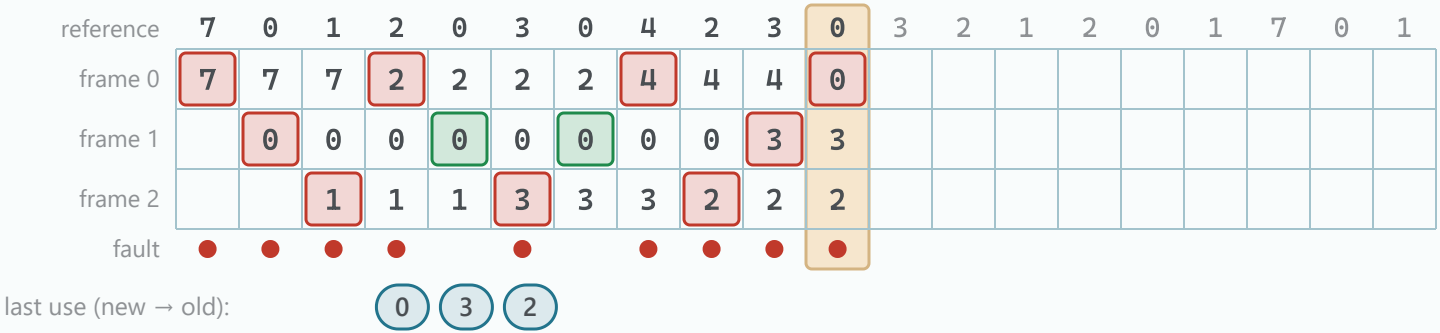
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



10. reference 3: fault; the least recently used, **0**, leaves. Faults: 8

LRU: Least Recently Used

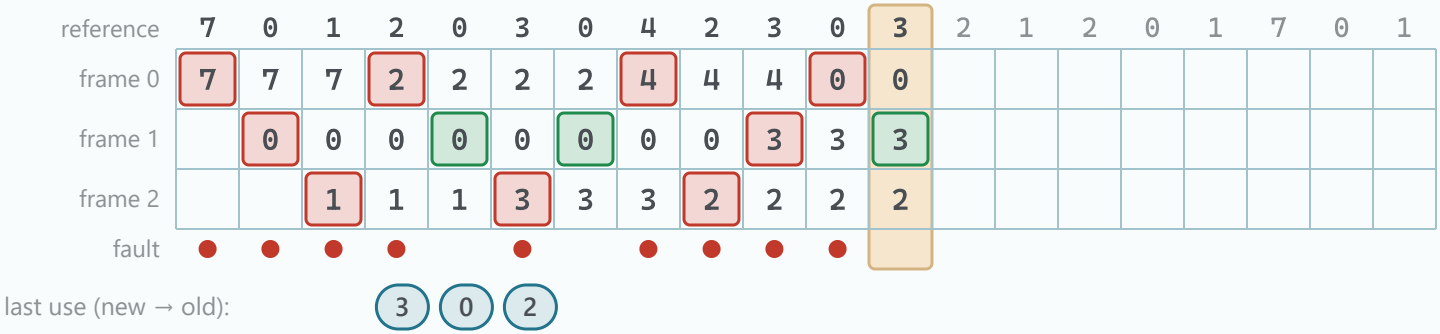
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



11. reference 0: fault; the least recently used, **4**, leaves. Faults: 9

LRU: Least Recently Used

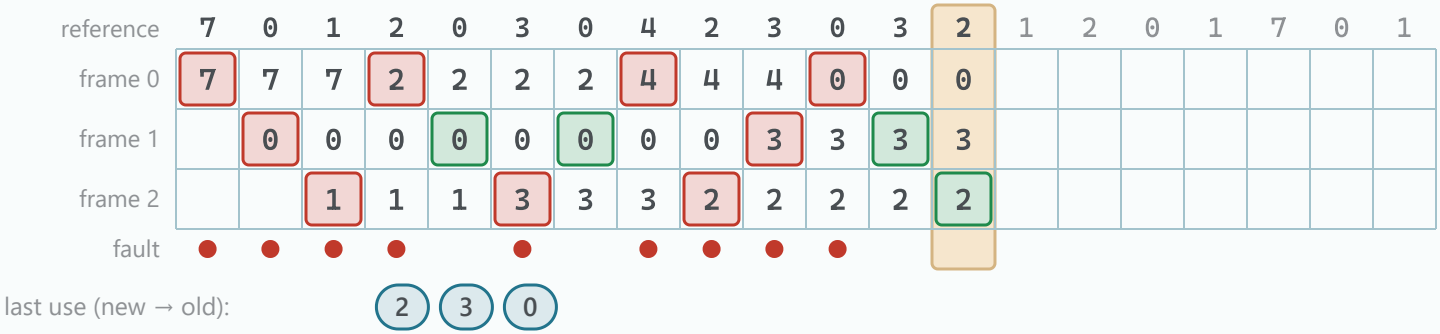
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



12. reference 3: in memory → hit; 3 becomes the most recent.

LRU: Least Recently Used

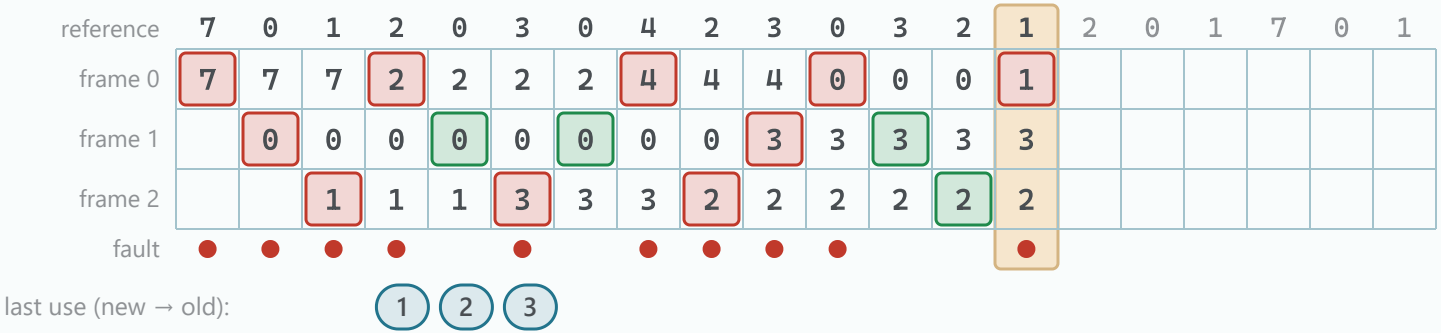
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



13. reference 2: in memory → hit; 2 becomes the most recent.

LRU: Least Recently Used

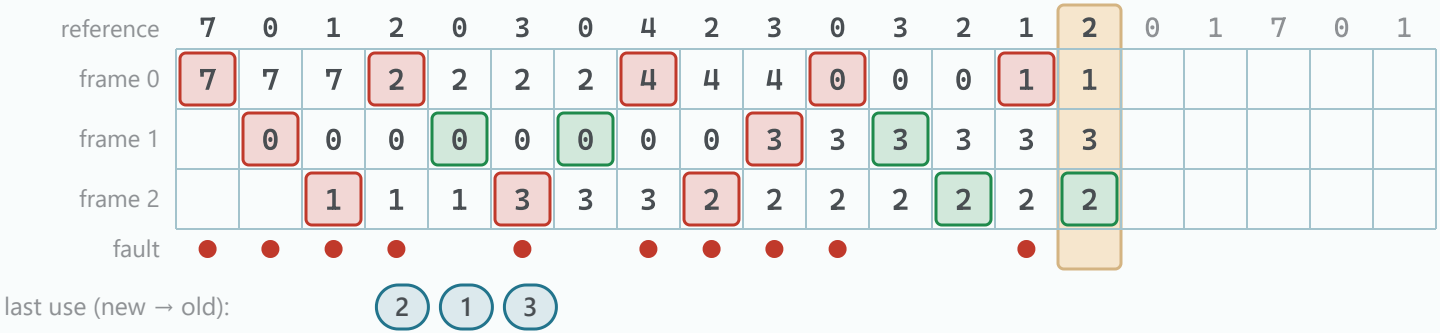
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



14. reference 1: fault; the least recently used, **0**, leaves. Faults: 10

LRU: Least Recently Used

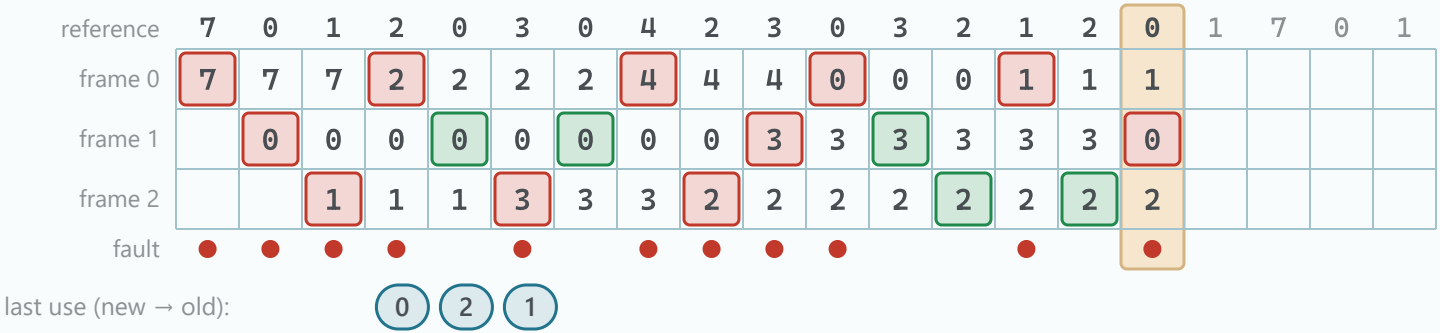
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



15. reference 2: in memory → hit; 2 becomes the most recent.

LRU: Least Recently Used

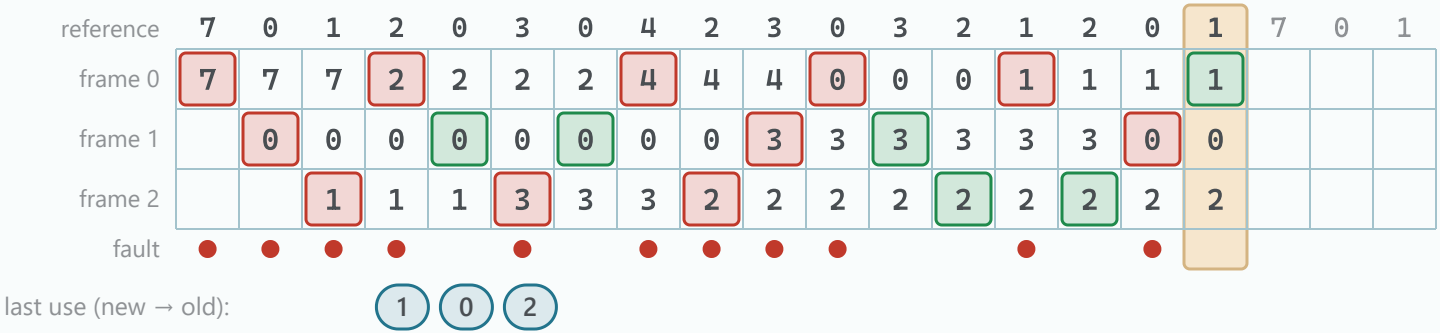
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



16. reference 0: fault; the least recently used, **3**, leaves. Faults: 11

LRU: Least Recently Used

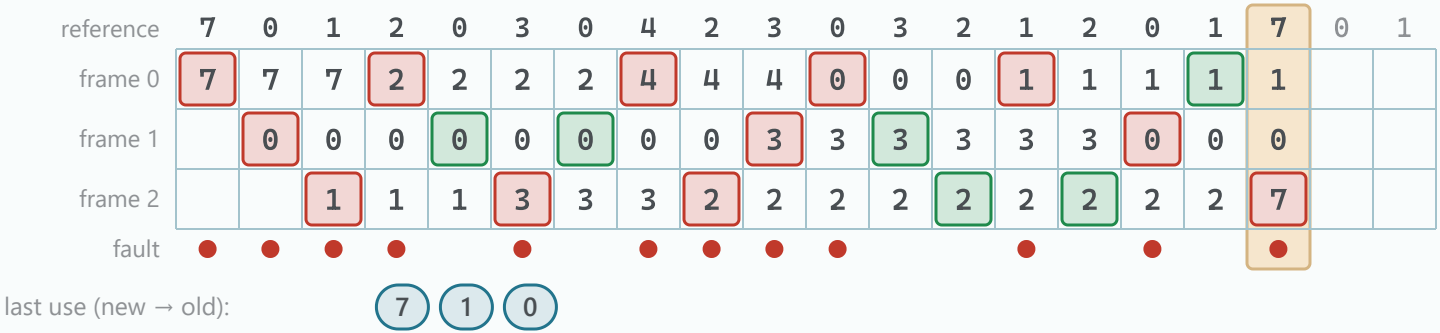
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



17. reference 1: in memory → hit; 1 becomes the most recent.

LRU: Least Recently Used

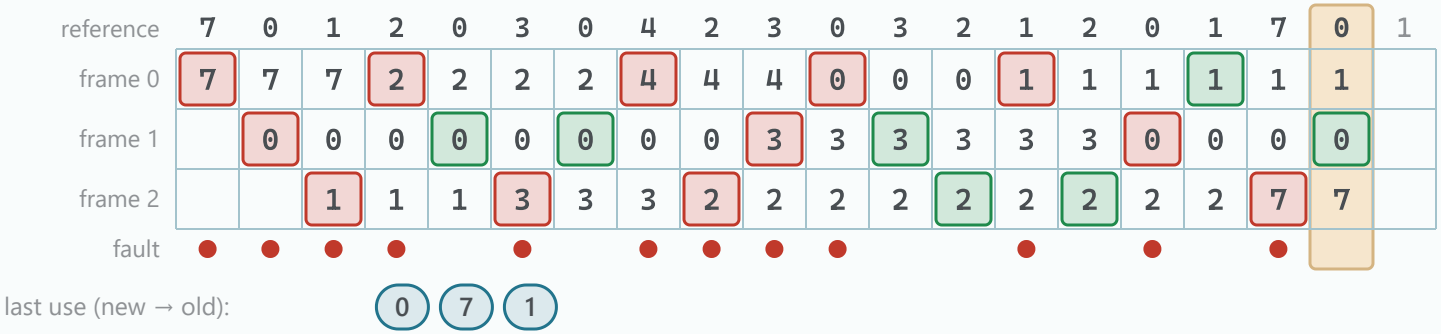
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



18. reference 7: fault; the least recently used, **2**, leaves. Faults: 12

LRU: Least Recently Used

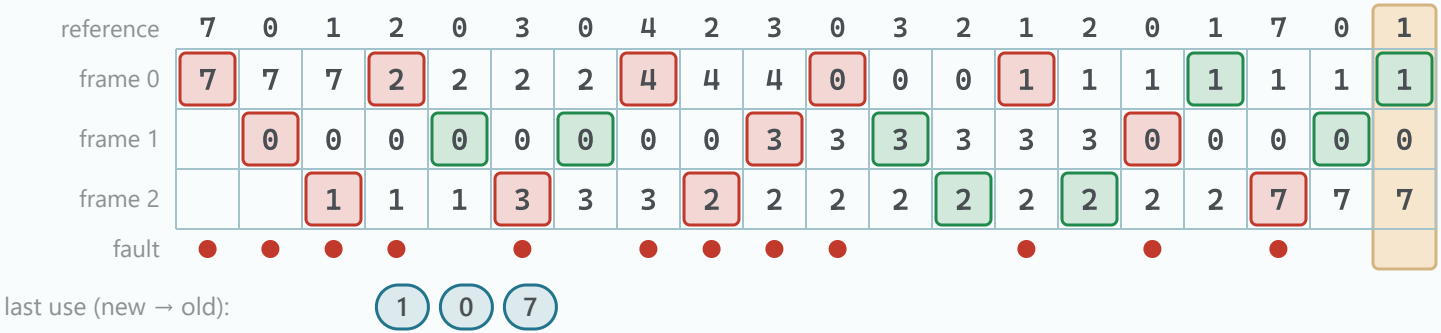
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



19. reference 0: in memory → hit; 0 becomes the most recent.

LRU: Least Recently Used

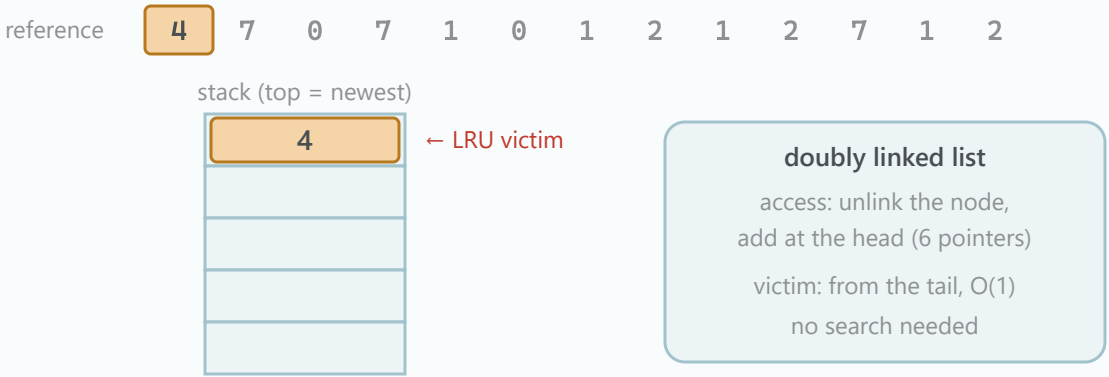
- **OPT looking backward:** a page not used in the recent past probably won't be used in the near future (locality).
- The page **least recently used** is replaced.
- Works well, but the record must be updated on **every access**: expensive without hardware support.
- The algorithm that evicts the most recently used page is MRU; LRU evicts the *least* recently used.



20. reference 1: in memory → hit; 1 becomes the most recent. **Total: 12 faults** (9 replacements): better than FIFO, worse than OPT.

Implementing LRU: Counters and a Stack

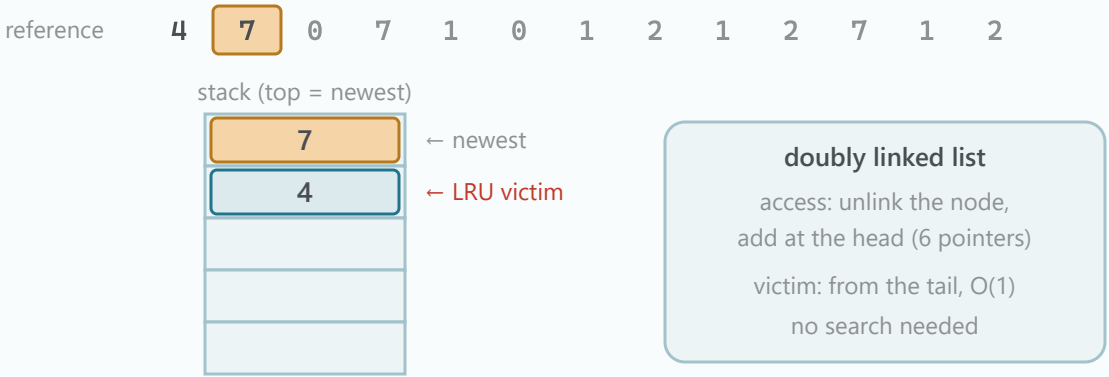
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



4 wasn't on the stack: it is **pushed** on top.

Implementing LRU: Counters and a Stack

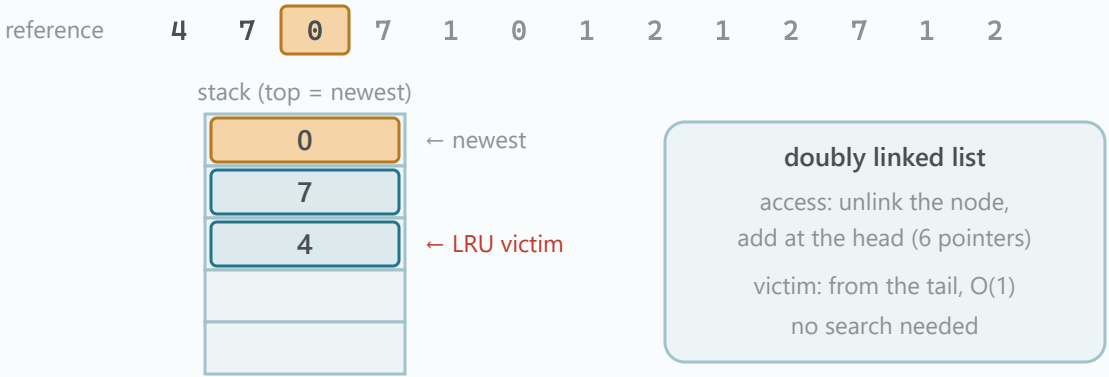
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ wasn't on the stack: it is **pushed** on top.

Implementing LRU: Counters and a Stack

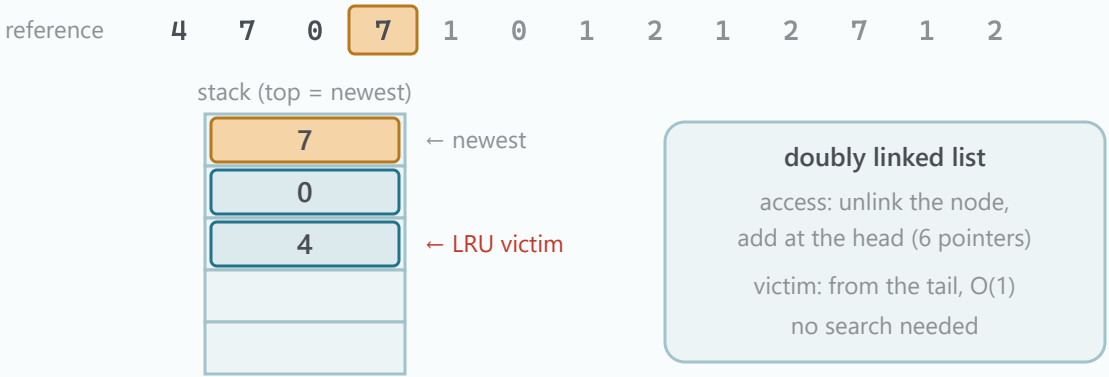
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ wasn't on the stack: it is **pushed** on top.

Implementing LRU: Counters and a Stack

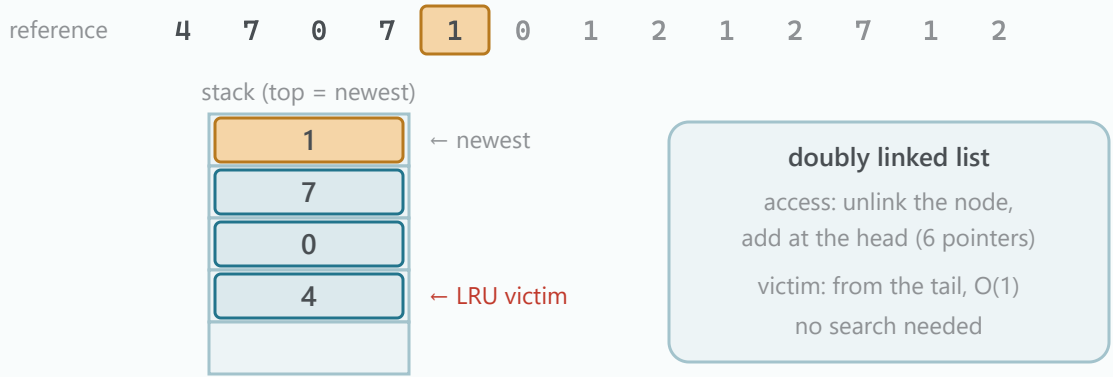
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



7 was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

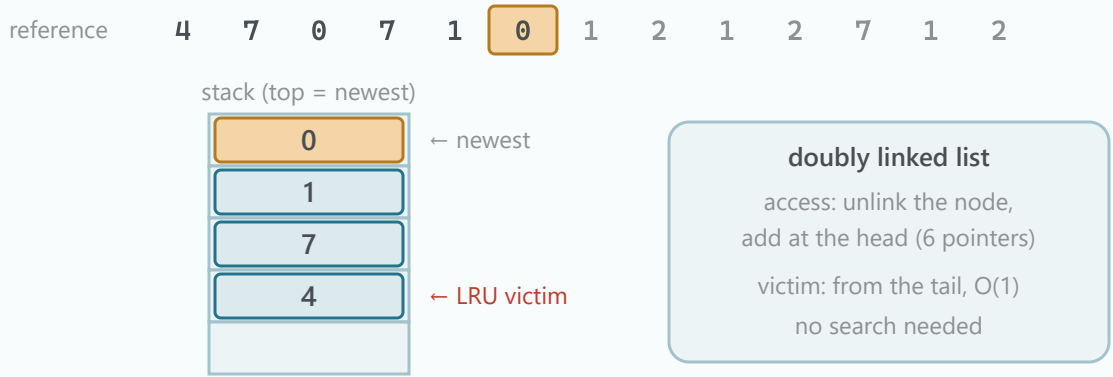
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ wasn't on the stack: it is **pushed** on top.

Implementing LRU: Counters and a Stack

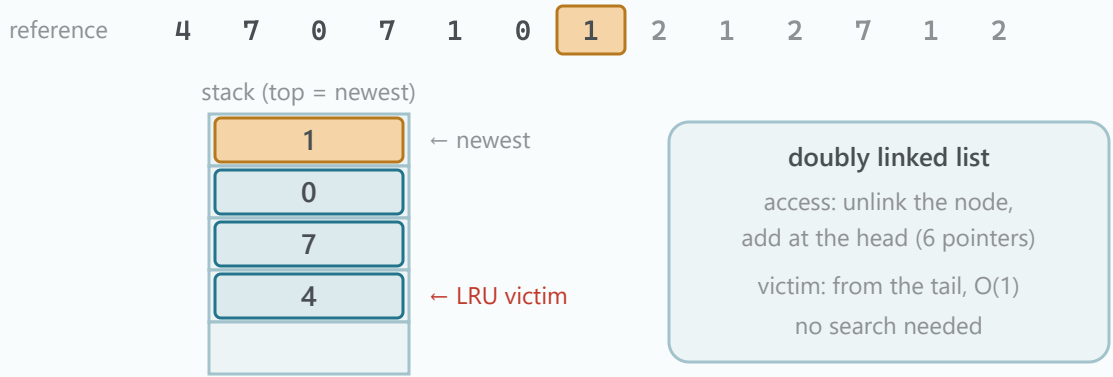
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

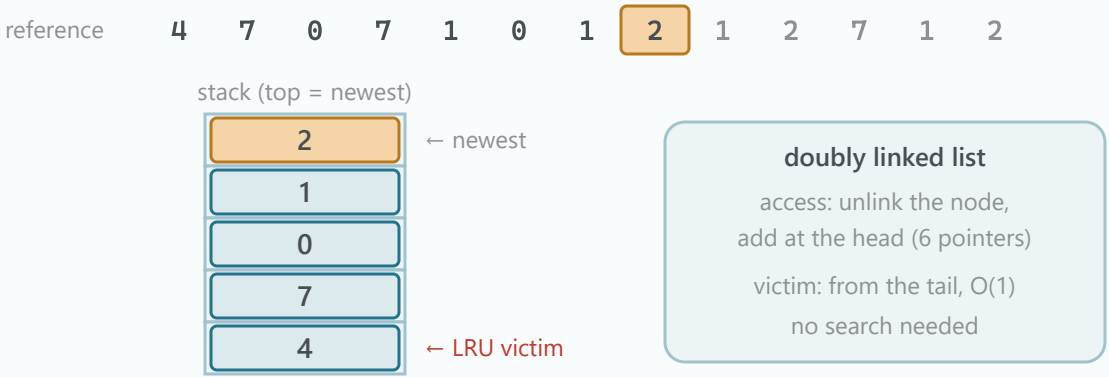
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

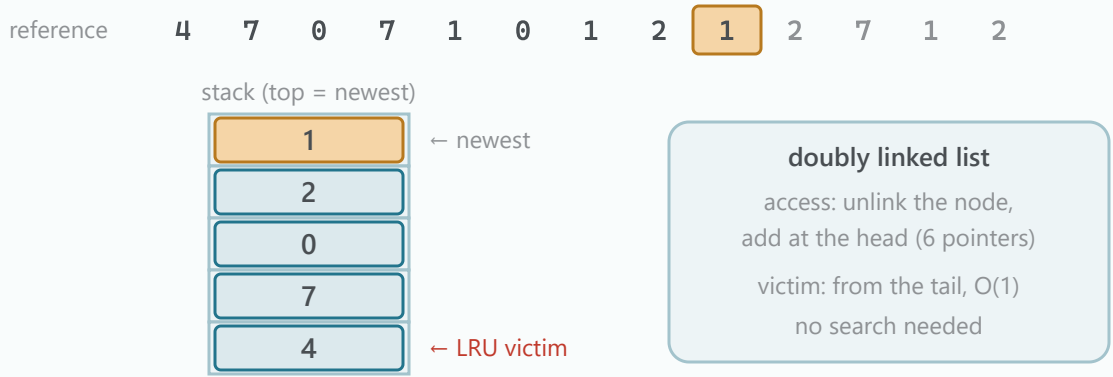
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ wasn't on the stack: it is **pushed** on top.

Implementing LRU: Counters and a Stack

- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



7 was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

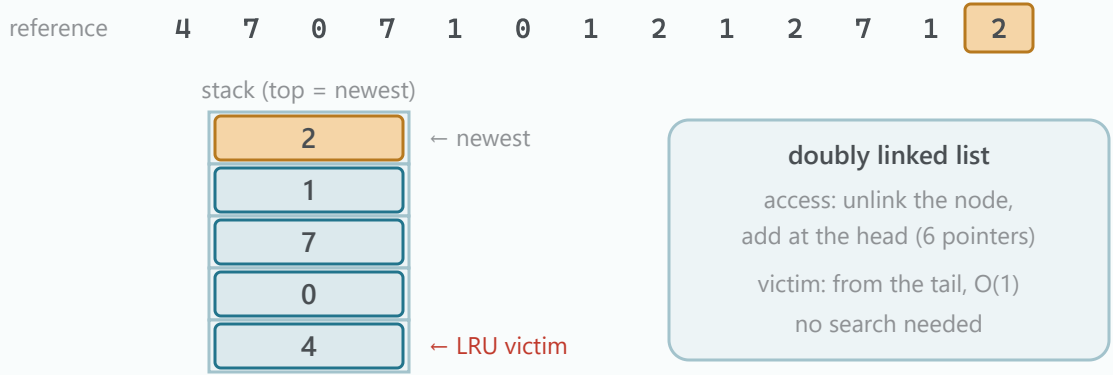
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



□ was on the stack: it is **taken out of the middle and put on top**.

Implementing LRU: Counters and a Stack

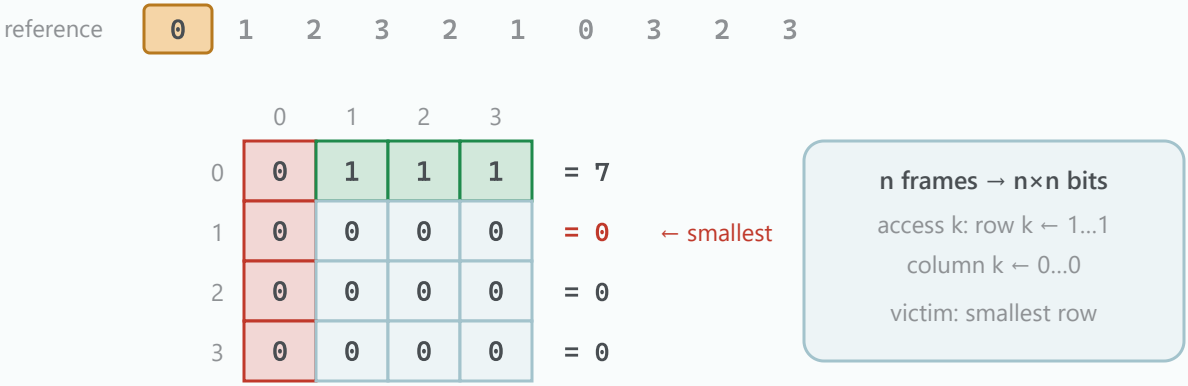
- **Counter (time stamp):** a logical clock in the CPU; on every access the clock value is written to the page's entry. Victim: the *smallest* stamp → a scan of the whole table on every fault.
- That means a table write on every memory access; hardware support is essential.
- **Stack:** page numbers kept in a stack; an accessed page moves to the *top*, and the victim is at the *bottom*.
- Updating is costly, selection is cheap. Neither exists in hardware today; in practice **approximations** are used.



2 was on the stack: it is **taken out of the middle and put on top**. **4** is at the bottom: it leaves on the next replacement.

Implementing LRU: The Matrix Method

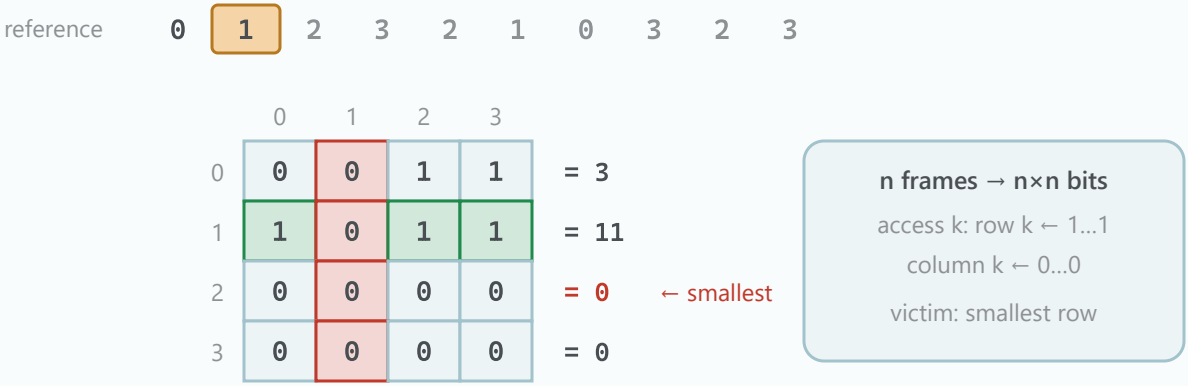
Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.



Page **0**: row 0 set to 1s, column 0 to 0s. Row values 7, 0, 0, 0 → smallest row: **1** (no victim is chosen while a frame is free).

Implementing LRU: The Matrix Method

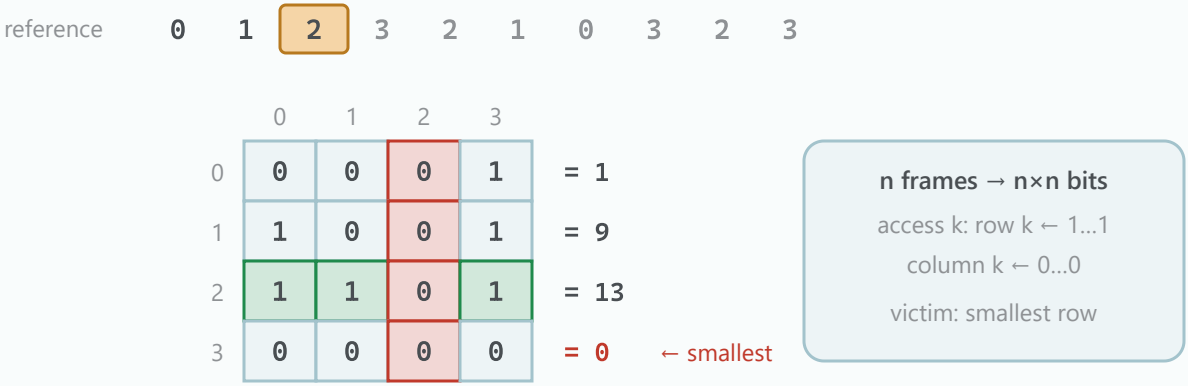
Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.



Page □: row □ set to 1s, column □ to 0s. Row values □ → smallest row: □ (no victim is chosen while a frame is free).

Implementing LRU: The Matrix Method

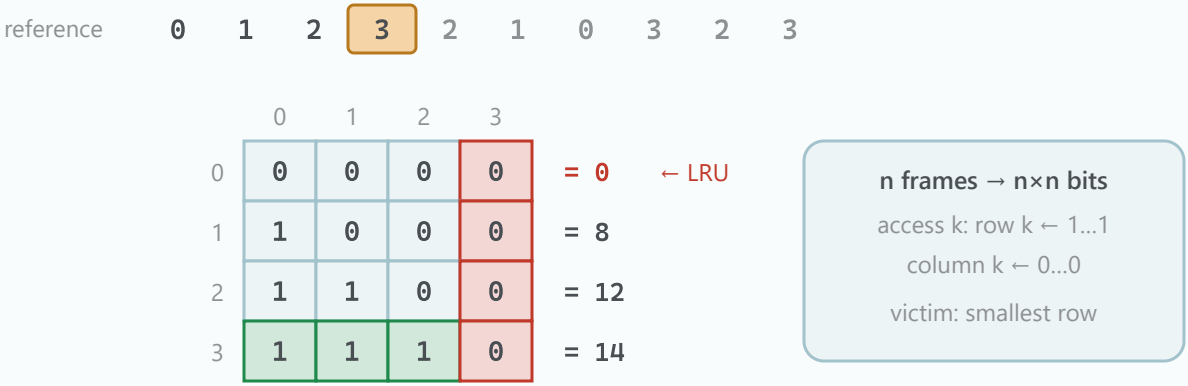
Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.



Page □: row □ set to 1s, column □ to 0s. Row values □ → smallest row: □ (no victim is chosen while a frame is free).

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.



Page **3**: row 3 set to 1s, column 3 to 0s. Row values 0, 8, 12, 14 → LRU (victim): **0**. The frames are now full.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.



Page **2**: row 2 set to 1s, column 2 to 0s. Row values 0, 8, 13, 12 → LRU (victim): **0**.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.

reference	0	1	2	3	2	**1**	0	3	2	3

	0	1	2	3		
0	0	0	0	0	= 0	← LRU
1	1	0	1	1	= 11	
2	1	0	0	1	= 9	
3	1	0	0	0	= 8	

n frames → n×n bits

access k: row k ← 1…1

column k ← 0…0

victim: smallest row

Page □: row □ set to 1s, column □ to 0s. Row values □ → LRU (victim): □.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.

reference **0** **1** **2** **3** **2** **1** **0** 3 2 3

	0	1	2	3	
0	0	1	1	1	= 7
1	0	0	1	1	= 3
2	0	0	0	1	= 1
3	0	0	0	0	= 0 ← LRU

n frames → n×n bits

access k: row k ← 1…1

column k ← 0…0

victim: smallest row

Page □: row □ set to 1s, column □ to 0s. Row values □ → LRU (victim): □.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.

reference	0	1	2	3	2	1	0	3	2	3

	0	1	2	3	
0	0	1	1	0	= 6
1	0	0	1	0	= 2
2	0	0	0	0	= 0 ← LRU
3	1	1	1	0	= 14

n frames → n×n bits

access k: row k ← 1…1

column k ← 0…0

victim: smallest row

Page □: row □ set to 1s, column □ to 0s. Row values □ → LRU (victim): □.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.

reference	0	1	2	3	2	1	0	3	2	3

	0	1	2	3	
0	0	1	0	0	= 4
1	0	0	0	0	= 0 ← LRU
2	1	1	0	1	= 13
3	1	1	0	0	= 12

n frames → n×n bits

access k: row k ← 1…1

column k ← 0…0

victim: smallest row

Page □: row □ set to 1s, column □ to 0s. Row values □ → LRU (victim): □.

Implementing LRU: The Matrix Method

Tanenbaum's hardware solution: 4 frames, access order `0 1 2 3 2 1 0 3 2 3`. The larger a row's binary value, the more recently the page was used.

reference	0	1	2	3	2	1	0	3	2	3

	0	1	2	3		
0	0	1	0	0	= 4	
1	0	0	0	0	= 0	← LRU
2	1	1	0	0	= 12	
3	1	1	1	0	= 14	

n frames → n×n bits

access k: row k ← 1…1

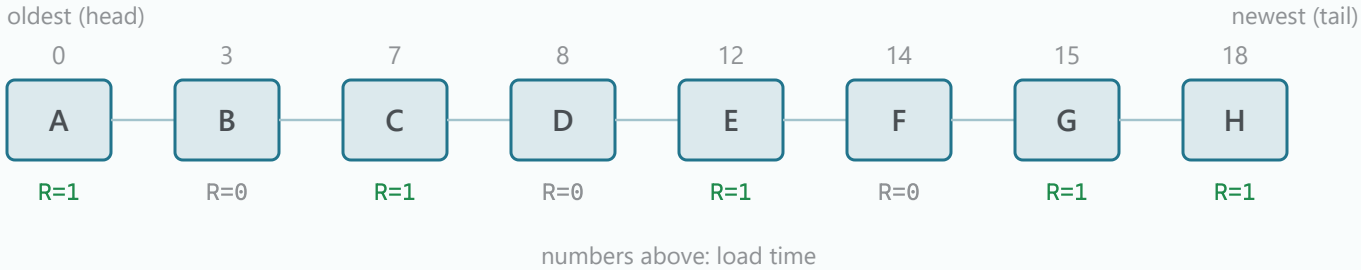
column k ← 0…0

victim: smallest row

Page □: row □ set to 1s, column □ to 0s. Row values □ → LRU (victim): □.

Second Chance

- An improved FIFO: look at the **R (referenced) bit** of the page at the head.
- R = 0 → the page is old and unused: evict it at once.
- R = 1 → clear R, move the page to the tail (a second chance), and keep searching.
- The problem: constantly **moving** pages around the list is needless work → the clock algorithm.

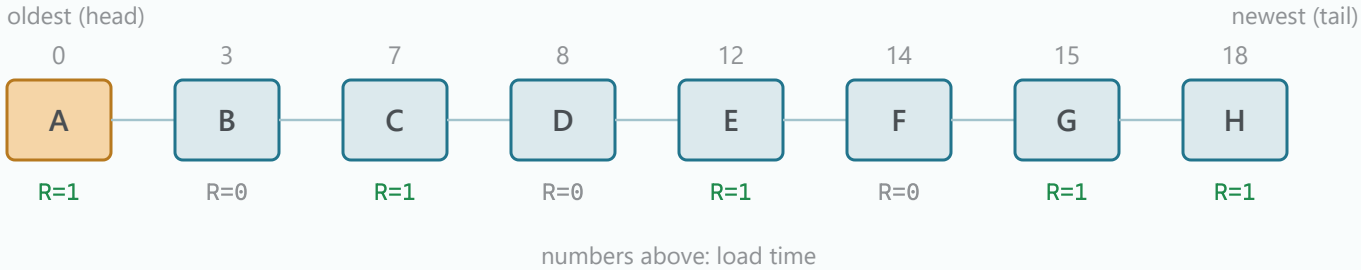


Pages are in FIFO order; at time 20 a page fault occurs and space is needed.

Tanenbaum MOS Figure 3-15 (redrawn).

Second Chance

- An improved FIFO: look at the **R (referenced) bit** of the page at the head.
- R = 0 → the page is old and unused: evict it at once.
- R = 1 → clear R, move the page to the tail (a second chance), and keep searching.
- The problem: constantly **moving** pages around the list is needless work → the clock algorithm.

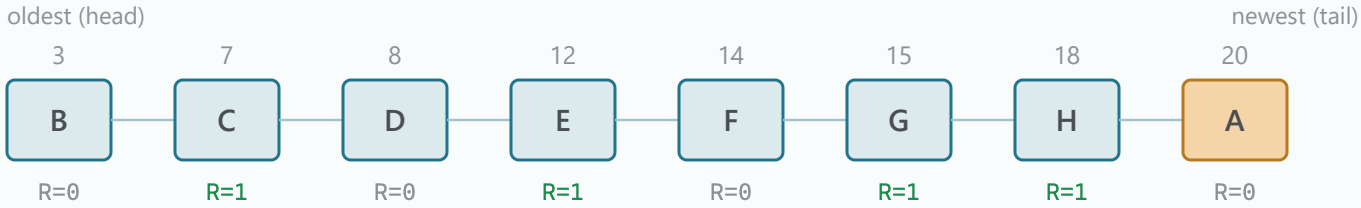


A at the head has R = 1: it was used recently. Instead of evicting it, give it a **second chance**.

Tanenbaum MOS Figure 3-15 (redrawn).

Second Chance

- An improved FIFO: look at the **R (referenced) bit** of the page at the head.
- R = 0 → the page is old and unused: evict it at once.
- R = 1 → clear R, move the page to the tail (a second chance), and keep searching.
- The problem: constantly **moving** pages around the list is needless work → the clock algorithm.

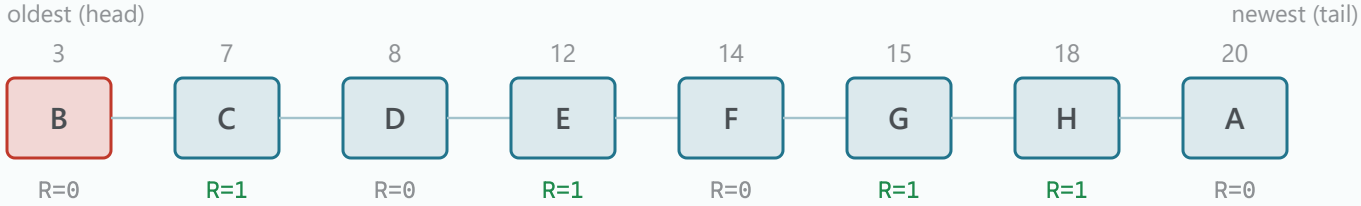


A's R is cleared and it moves to the **tail** as if newly loaded (time 20).

Tanenbaum MOS Figure 3-15 (redrawn).

Second Chance

- An improved FIFO: look at the **R (referenced) bit** of the page at the head.
- R = 0 → the page is old and unused: evict it at once.
- R = 1 → clear R, move the page to the tail (a second chance), and keep searching.
- The problem: constantly **moving** pages around the list is needless work → the clock algorithm.

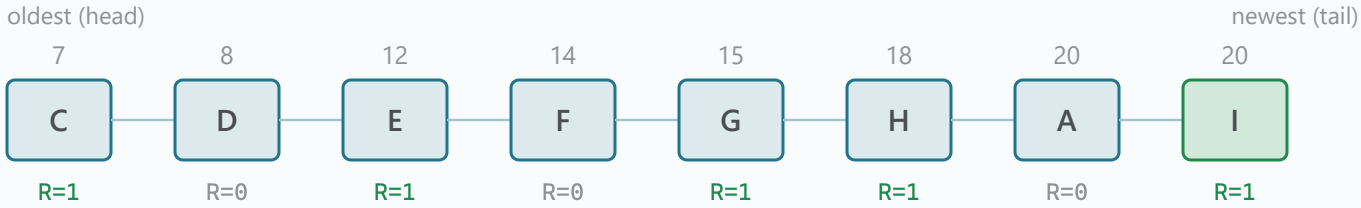


The new head, **B**: R = 0 → both old and unused: the **victim**.

Tanenbaum MOS Figure 3-15 (redrawn).

Second Chance

- An improved FIFO: look at the **R (referenced) bit** of the page at the head.
- R = 0 → the page is old and unused: evict it at once.
- R = 1 → clear R, move the page to the tail (a second chance), and keep searching.
- The problem: constantly **moving** pages around the list is needless work → the clock algorithm.



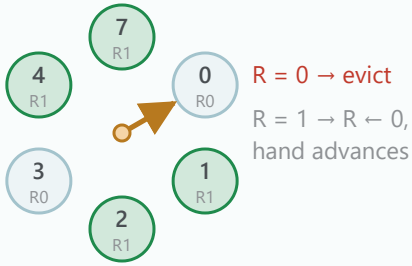
numbers above: load time

B is evicted and the new page I joins the tail. Had every R been 1, the list would go around once and behave like **pure FIFO**.

Tanenbaum MOS Figure 3-15 (redrawn).

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



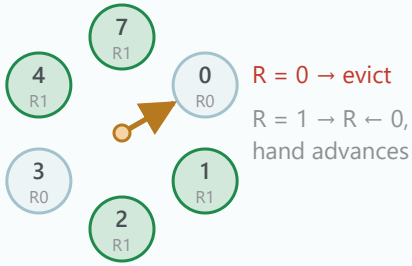
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7 R1																			
frame 1		◀																		
frame 2																				
fault	●																			

R bits: 7: 1

1. reference 7: fault; loaded into free frame 0. Faults: 1

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.

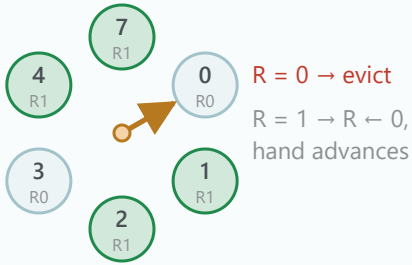


reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7 R1																		
frame 1		0 R1																		
frame 2																				
fault	●	●																		
R bits:	7: 1	0: 1																		

2. reference 0: fault; loaded into free frame 1. Faults: 2

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



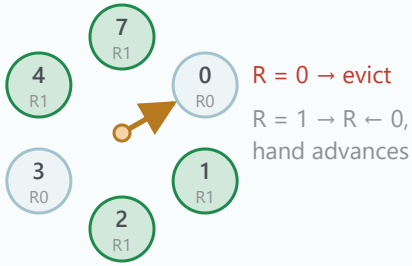
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7 R1																	
frame 1		0	0 R1																	
frame 2			1 R1																	
fault	●	●	●																	

R bits: 7: 1 | 0: 1 | 1: 1

3. reference 1: fault; loaded into free frame 2. Faults: 3

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



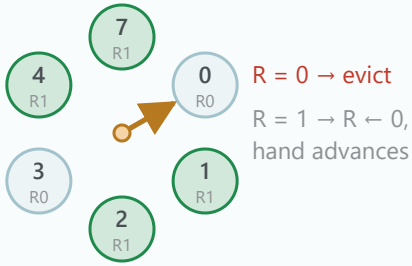
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2 R1																
frame 1		0	0	0 R0																
frame 2			1	1 R0																
fault	●	●	●	●																

R bits: 2: 1 | 0: 0 | 1: 0

4. reference 2: fault; the hand clears R for 7, 0, 1 (R=1) and moves on; **7** under the hand has R = 0 → evicted.
Faults: 4

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



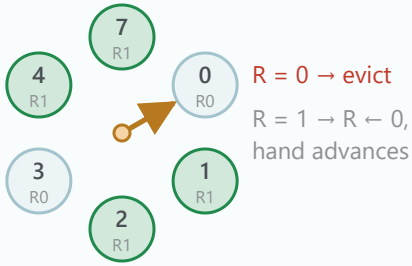
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2 R1															
frame 1		0	0	0	0 R1															
frame 2			1	1	1 R0															
fault	●	●	●	●																

R bits: 2: 1 | 0: 1 | 1: 0

5. reference 0: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



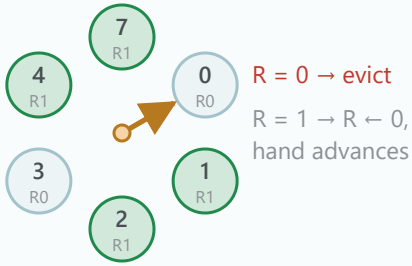
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2 R1														
frame 1		0	0	0	0	0 R0														
frame 2			1	1	1	3 R1														
fault	●	●	●	●		●														

R bits: 2: 1 | 0: 0 | 3: 1

6. reference 3: fault; the hand clears R for 0 (R=1) and moves on; **1** under the hand has R = 0 → evicted. Faults: 5

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



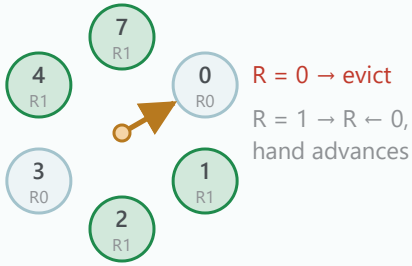
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2 R1													
frame 1		0	0	0	0	0	0 R1													
frame 2			1	1	1	3	3 R1													
fault	●	●	●	●		●														

R bits: 2: 1 | 0: 1 | 3: 1

7. reference 0: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



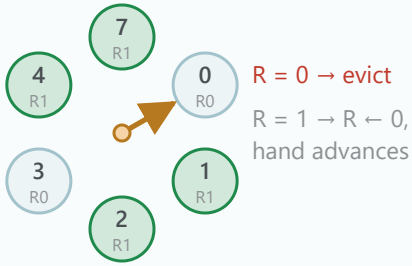
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4 R1												
frame 1		0	0	0	0	0	0	0 R0												
frame 2			1	1	1	3	3	3 R0												
fault	●	●	●	●		●		●												

R bits: 4: 1 | 0: 0 | 3: 0

8. reference 4: fault; the hand clears R for 2, 0, 3 (R=1) and moves on; **2** under the hand has R = 0 → evicted.
Faults: 6

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



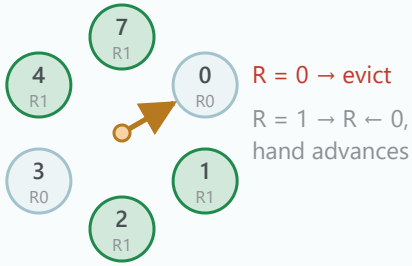
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4 R1											
frame 1		0	0	0	0	0	0	0	2 R1											
frame 2			1	1	1	3	3	3	3 R0											
fault	●	●	●	●		●		●	●											

R bits: 4: 1 | 2: 1 | 3: 0

9. reference 2: fault; **0** under the hand has R = 0 → evicted. Faults: 7

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



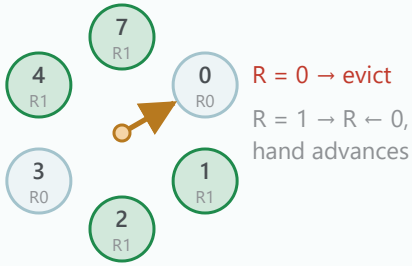
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4 R1										
frame 1		0	0	0	0	0	0	0	2	2 R1										
frame 2			1	1	1	3	3	3	3	3 R1										
fault	●	●	●	●		●		●	●											

R bits: 4: 1 | 2: 1 | 3: 1

10. reference 3: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



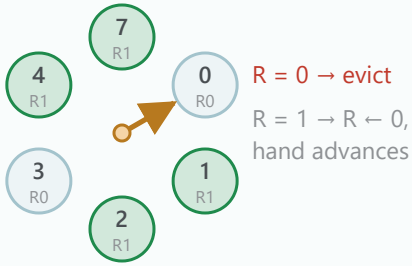
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4 R0									
frame 1		0	0	0	0	0	0	0	2	2	2 R0									
frame 2			1	1	1	3	3	3	3	3	0 R1									
fault	●	●	●	●		●		●	●		●									

R bits: 4: 0 | 2: 0 | 0: 1

11. reference 0: fault; the hand clears R for 3, 4, 2 (R=1) and moves on; **3** under the hand has R = 0 → evicted.
Faults: 8

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



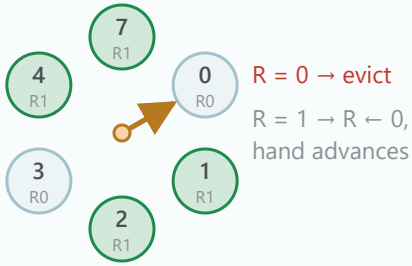
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3 R1								
frame 1		0	0	0	0	0	0	0	2	2	2	2 R0								
frame 2			1	1	1	3	3	3	3	3	0	0 R1								
fault	●	●	●	●		●		●	●		●	●								

R bits: 3: 1 | 2: 0 | 0: 1

12. reference 3: fault; **4** under the hand has R = 0 → evicted. Faults: 9

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



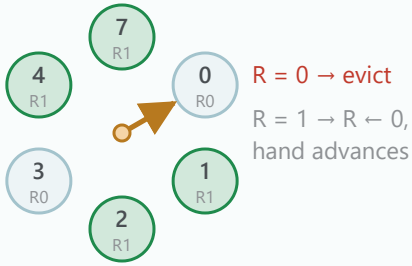
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3 R1							
frame 1		0	0	0	0	0	0	0	2	2	2	2	2 R1							
frame 2			1	1	1	3	3	3	3	3	0	0	0 R1							
fault	●	●	●	●		●		●	●		●	●								

R bits: 3: 1 | 2: 1 | 0: 1

13. reference 2: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



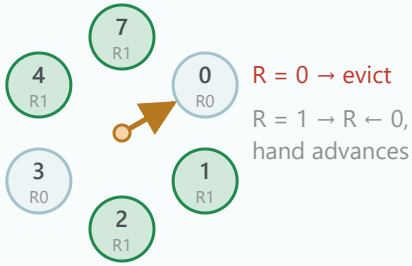
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3 R0						
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1 R1						
frame 2			1	1	1	3	3	3	3	3	0	0	0	0 R0						
fault	●	●	●	●		●		●	●		●	●		●						

R bits: 3: 0 | 1: 1 | 0: 0

14. reference 1: fault; the hand clears R for 2, 0, 3 (R=1) and moves on; **2** under the hand has R = 0 → evicted.
Faults: 10

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



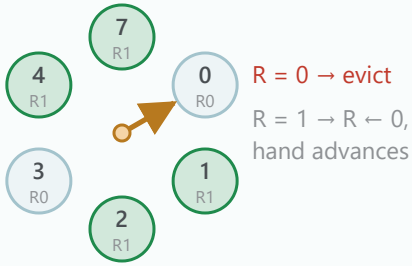
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3 R0					
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1 R1					
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2 R1					
fault	●	●	●	●		●		●	●		●	●		●	●					

R bits: 3: 0 | 1: 1 | 2: 1

15. reference 2: fault; **0** under the hand has R = 0 → evicted. Faults: 11

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



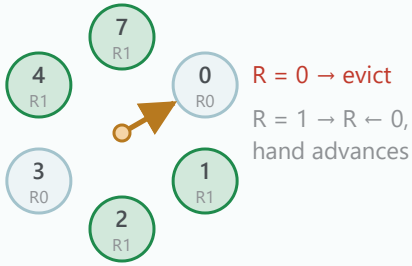
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3	0 R1				
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1	1 R1				
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2	2 R1				
fault	●	●	●	●		●		●	●		●	●		●	●	●				

R bits: 0: 1 | 1: 1 | 2: 1

16. reference 0: fault; **3** under the hand has R = 0 → evicted. Faults: 12

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



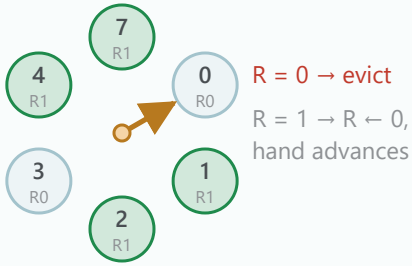
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3	0	0 R1			
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1	1	1 R1			
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2	2	2 R1			
fault	●	●	●	●		●		●	●		●	●		●	●	●				

R bits: 0: 1 | 1: 1 | 2: 1

17. reference 1: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



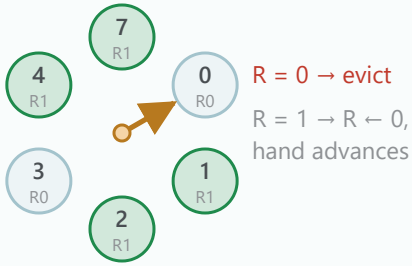
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3	0	0	0 R0		
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1	1	1	7 R1		
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2	2	2	2 R0		
fault	●	●	●	●		●		●	●		●	●		●	●	●		●		

R bits: 0: 0 | 7: 1 | 2: 0

18. reference 7: fault; the hand clears R for 1, 2, 0 (R=1) and moves on; **1** under the hand has R = 0 → evicted.
Faults: 13

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



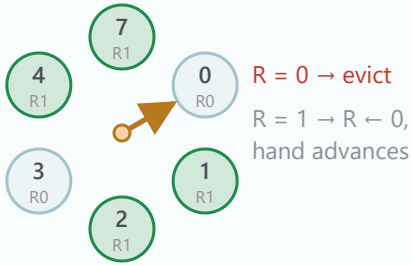
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3	0	0	0	0 R1	
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1	1	1	7	7 R1	
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2	2	2	2	2 R0	
fault	●	●	●	●		●		●	●		●	●		●	●	●		●		

R bits: 0: 1 | 7: 1 | 2: 0

19. reference 0: in memory → hit; hardware sets R ← 1, the hand doesn't move.

The Clock Algorithm

- The same decision as second chance, a different structure: frames sit in a **circular list**, a **hand** (pointer) points to the oldest page; pages don't move.
- A loaded or accessed page gets R = 1. The hand moves only on a fault.
- The hand moves around the *frame list*, not the page table.



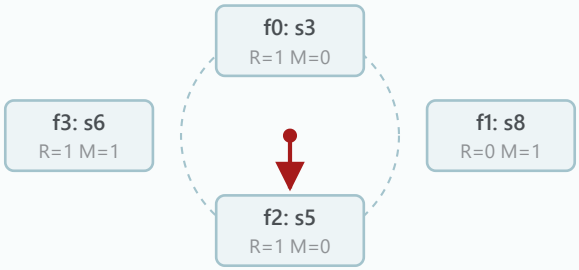
reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	4	3	3	3	3	0	0	0	0	0 R1
frame 1		0	0	0	0	0	0	0	2	2	2	2	2	1	1	1	1	7	7	7 R1
frame 2			1	1	1	3	3	3	3	3	0	0	0	0	2	2	2	2	2	1 R1
fault	●	●	●	●		●		●	●		●	●		●	●	●		●		●

R bits: 0: 1 | 7: 1 | 1: 1

20. reference 1: fault; **2** under the hand has R = 0 → evicted. Faults: 14 **Total: 14 faults**: one fewer than FIFO (15), more than LRU (12).

The Clock Algorithm: Code Trace

```
 1 int pick_victim(void) {
 2   while (frm[hand].R == 1) { /* 2nd chance */
 3     frm[hand].R = 0;
 4     hand = (hand + 1) % N;
 5   }
 6   int k = hand;             /* victim */
 7   hand = (hand + 1) % N;
 8   return k;
 9 }
10 void fault(int p) {
▶   int k = pick_victim();
▶   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



hand = 2 · written to disk: 0

Four frames are full and the hand is at frame 2. A fault occurs for page 9: `fault(9)` looks for a victim.

Step 1 / 10

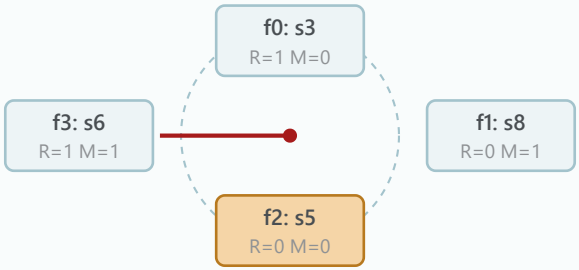
VARIABLES

`p = 9` `hand = 2` `k = –` `f2.R = 1`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
▶      frm[hand].R = 0;
▶      hand = (hand + 1) % N;
▶    }
6    int k = hand;              /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



Frame 2: s5 has R = 1, so it was used recently. R ← 0 (second chance) and the hand moves to 3.

Step 2 / 10

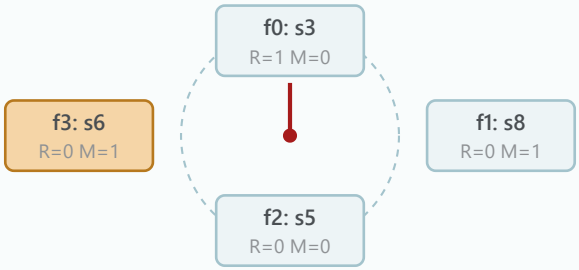
VARIABLES

`p = 9` `hand = 3` `k = –` `f3.R = 1`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
▶      frm[hand].R = 0;
▶      hand = (hand + 1) % N;
▶    }
6    int k = hand;              /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



Frame 3: s6 has R = 1, so it was used recently. R ← 0 (second chance) and the hand moves to 0.

Step 3 / 10

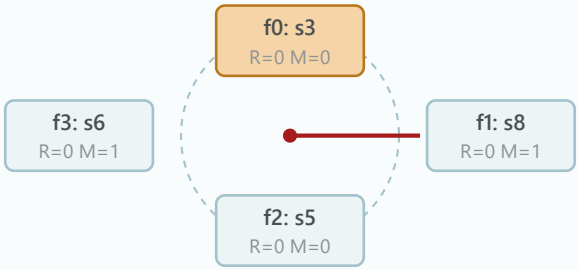
VARIABLES

`p = 9` `hand = 0` `k = –` `f0.R = 1`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
▶      frm[hand].R = 0;
▶      hand = (hand + 1) % N;
▶    }
6    int k = hand;              /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



Frame 0: s3 has R = 1, so it was used recently. R ← 0 (second chance) and the hand moves to 1.

Step 4 / 10

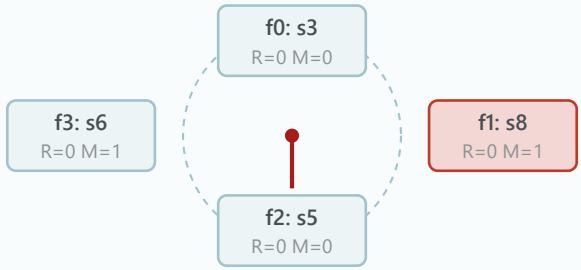
VARIABLES

`p = 9` `hand = 1` `k = –` `f1.R = 0`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
▶      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;              /* victim */
▶    hand = (hand + 1) % N;
▶    return k;
▶  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



hand = 2 · written to disk: 0

Frame 1 has R = 0: the loop ends. k = 1 is the victim; the hand moves on to the next frame (2).

Step 5 / 10

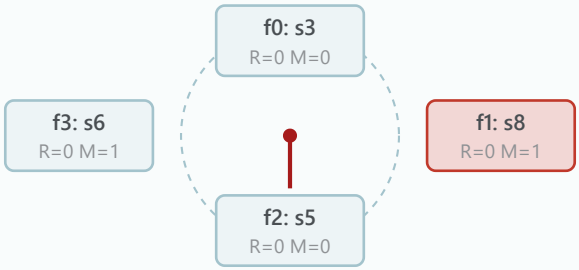
VARIABLES

`p = 9` `hand = 2` `k = 1` `f2.R = 0`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
3      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;             /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
▶    frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



M = 1: p8 written to disk first · written to disk: 1

p8's M (dirty) bit is 1: the copy in memory isn't on disk. It is **written to disk** before eviction (extra I/O).

Step 6 / 10

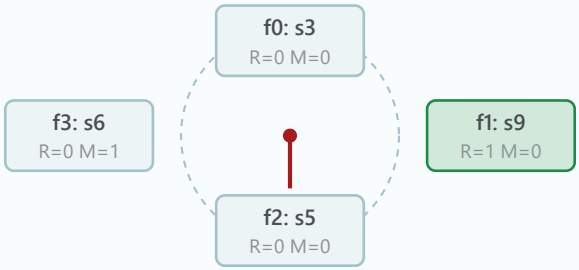
VARIABLES

`p = 9` `hand = 2` `k = 1` `f2.R = 0`

OUTPUT

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
3      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;             /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
▶    frm[k].M = 0;
▶  }
```



hand = 2 · written to disk: 1

p9 is loaded into frame 1: R = 1, M = 0. A full lap was made; three pages now have R = 0.

Step 7 / 10

VARIABLES

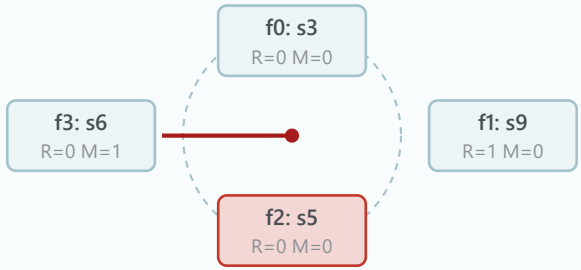
`p = 9` `hand = 2` `k = 1` `f2.R = 0`

OUTPUT

```
fault(9): victim f1 (s8, dirty)
```

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
▶      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;              /* victim */
▶    hand = (hand + 1) % N;
▶    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
▶    if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



hand = 3 · written to disk: 1

`fault(4)`: the hand is at frame 2 and R = 0 (cleared on the previous lap, unused since). The victim is found at once: k = 2.

Step 8 / 10

VARIABLES

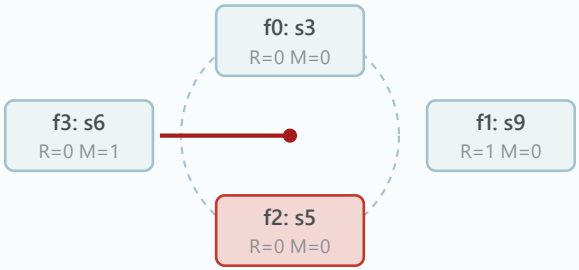
`p = 4` `hand = 3` `k = 2` `f3.R = 0`

OUTPUT

```
fault(9): victim f1 (s8, dirty)
```

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
3      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;              /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
▶    frm[k].page = p; frm[k].R = 1;
14   frm[k].M = 0;
15 }
```



hand = 3 · written to disk: 1

p5 is clean (M = 0): the copy on disk is current, no write needed. A clean victim is cheaper.

Step 9 / 10

VARIABLES

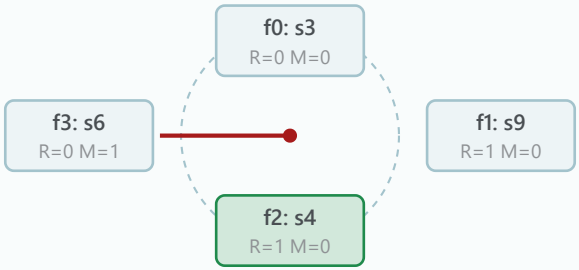
`p = 4` `hand = 3` `k = 2` `f3.R = 0`

OUTPUT

```
fault(9): victim f1 (s8, dirty)
```

The Clock Algorithm: Code Trace

```
1  int pick_victim(void) {
2    while (frm[hand].R == 1) { /* 2nd chance */
3      frm[hand].R = 0;
4      hand = (hand + 1) % N;
5    }
6    int k = hand;             /* victim */
7    hand = (hand + 1) % N;
8    return k;
9  }
10 void fault(int p) {
11   int k = pick_victim();
12   if (frm[k].M) write_back(k); /* dirty */
13   frm[k].page = p; frm[k].R = 1;
▶    frm[k].M = 0;
▶  }
```



hand = 3 · written to disk: 1

p4 has been loaded into frame 2. Two faults cleared 3 R bits and caused 1 disk write; the hand waits at frame 3.

Step 10 / 10

VARIABLES

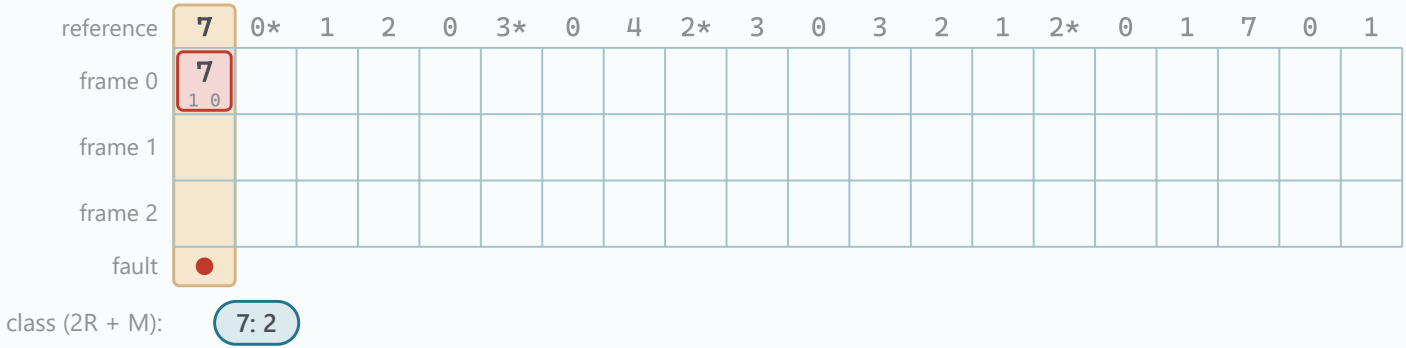
`p = 4` `hand = 3` `k = 2` `f3.R = 0`

OUTPUT

```
fault(9): victim f1 (s8, dirty)
fault(4): victim f2 (s5, clean)
```

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

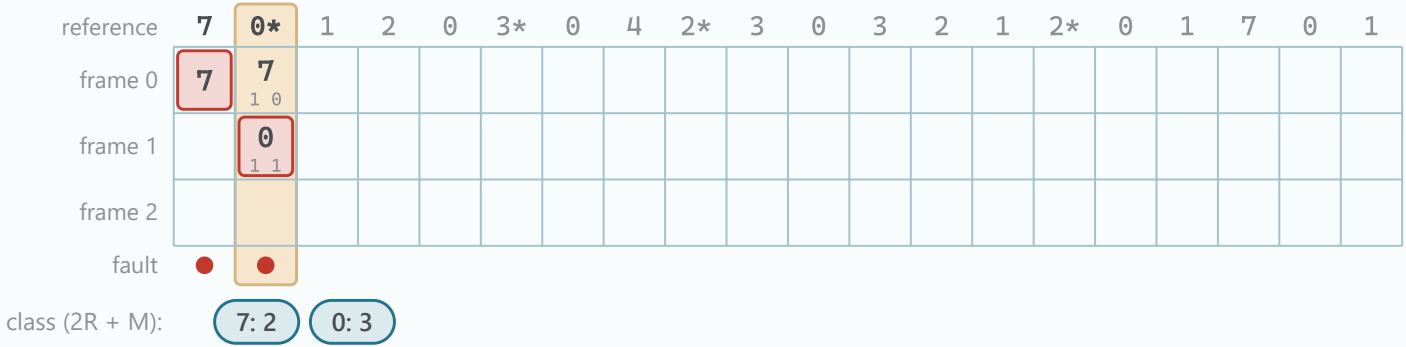


1. reference 7: fault; loaded into free frame 0. Faults: 1

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

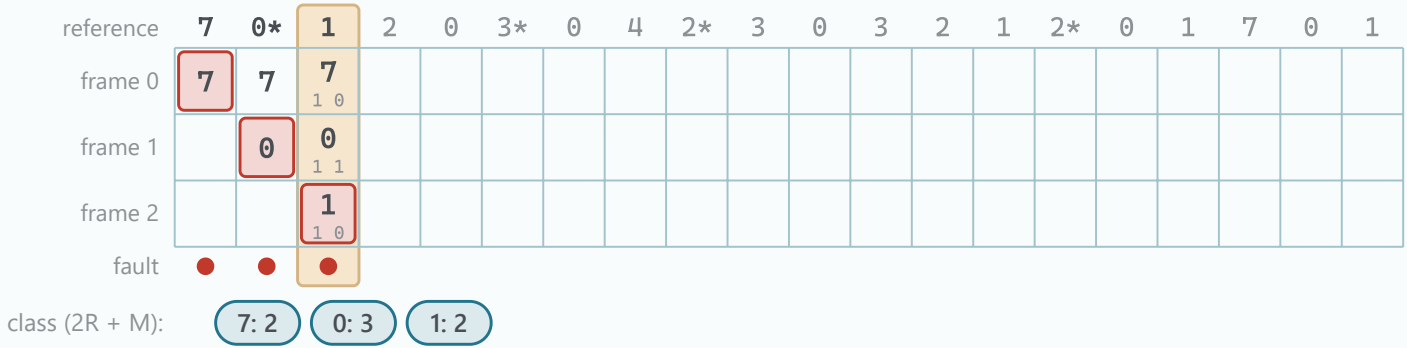


2. reference 0*: fault; loaded into free frame 1. Faults: 2

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

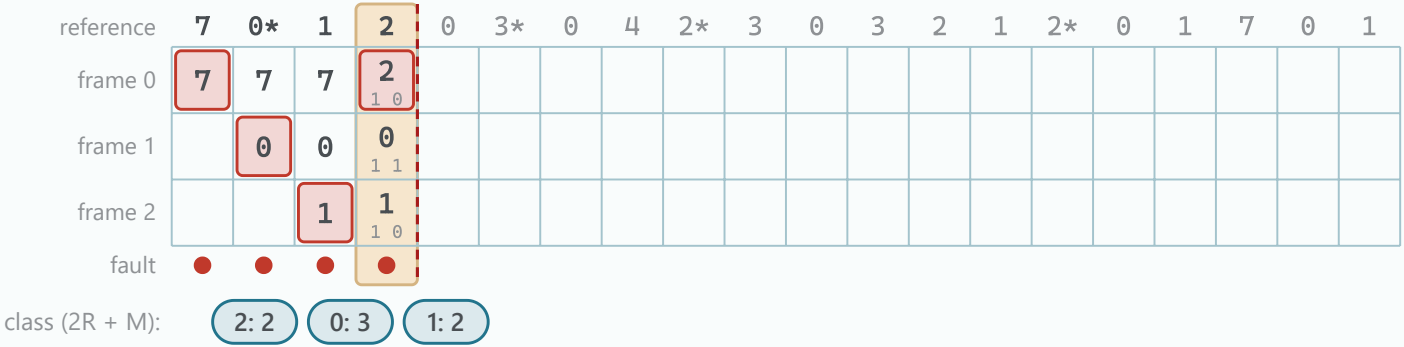


3. reference 1: fault; loaded into free frame 2. Faults: 3

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

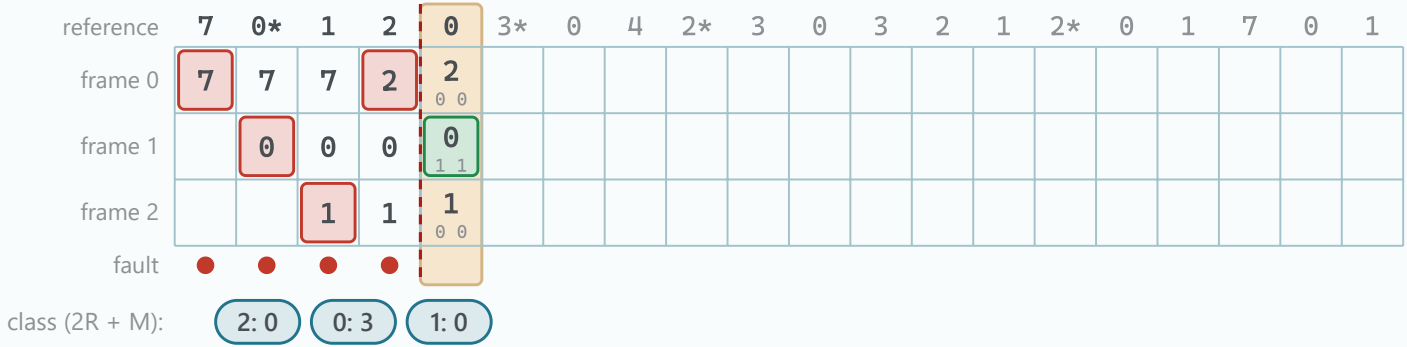


4. reference 2: fault; classes (7→2, 0→3, 1→2); **7**, in the lowest class, leaves. Faults: 4 Clock tick: all R ← 0.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

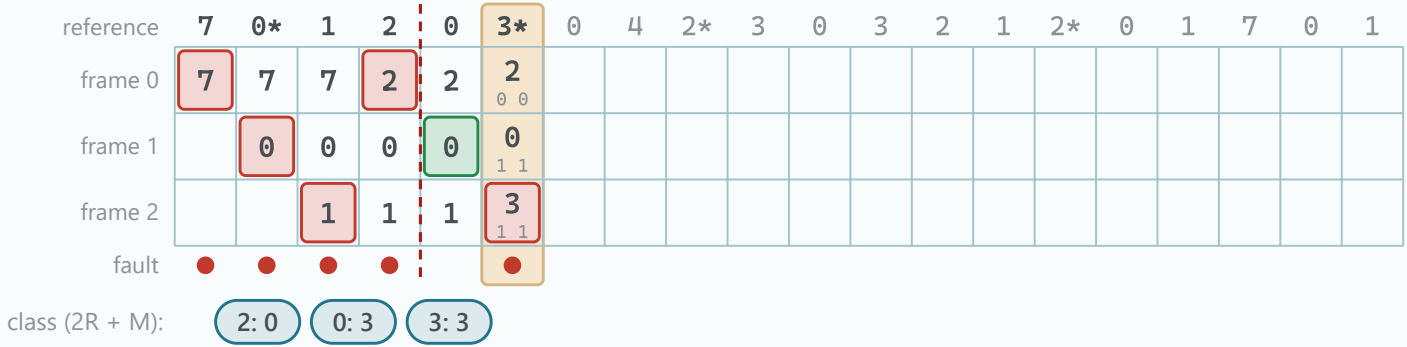


5. reference 0: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

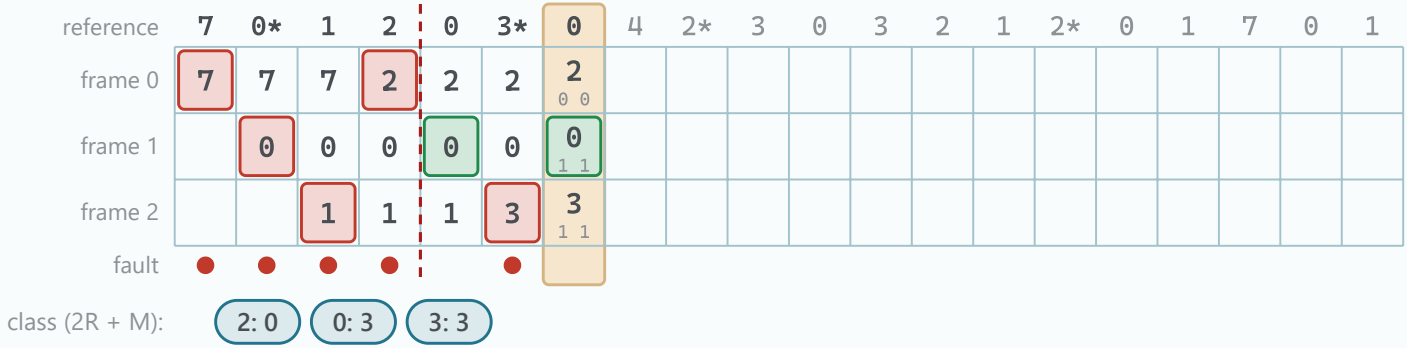


6. reference 3*: fault; classes (2→0, 0→3, 1→0); **1**, in the lowest class, leaves. Faults: 5

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

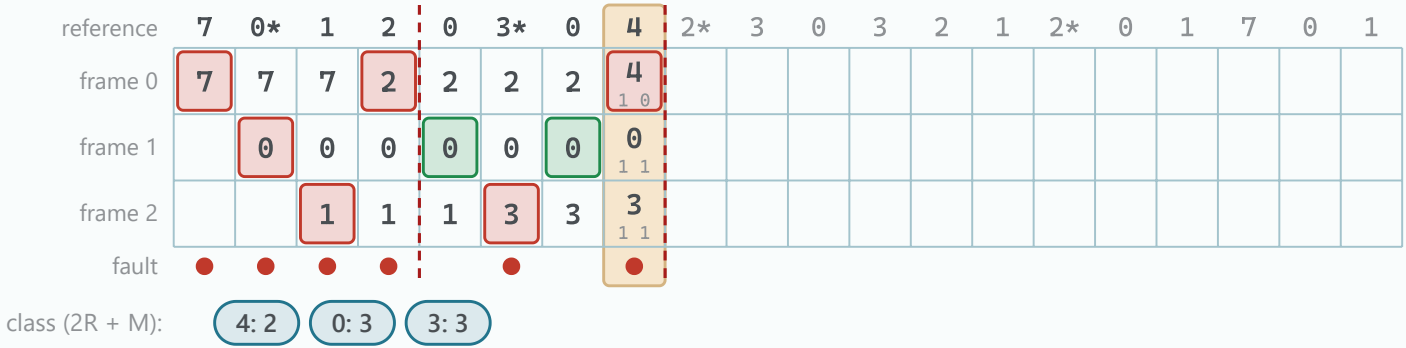


7. reference 0: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

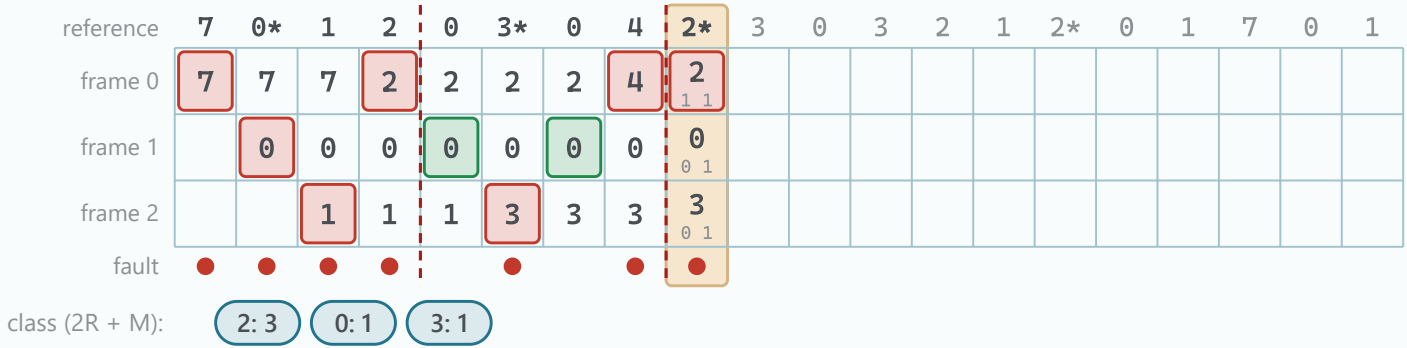


8. reference 4: fault; classes (2→0, 0→3, 3→3); **2**, in the lowest class, leaves. Faults: 6 Clock tick: all R ← 0.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

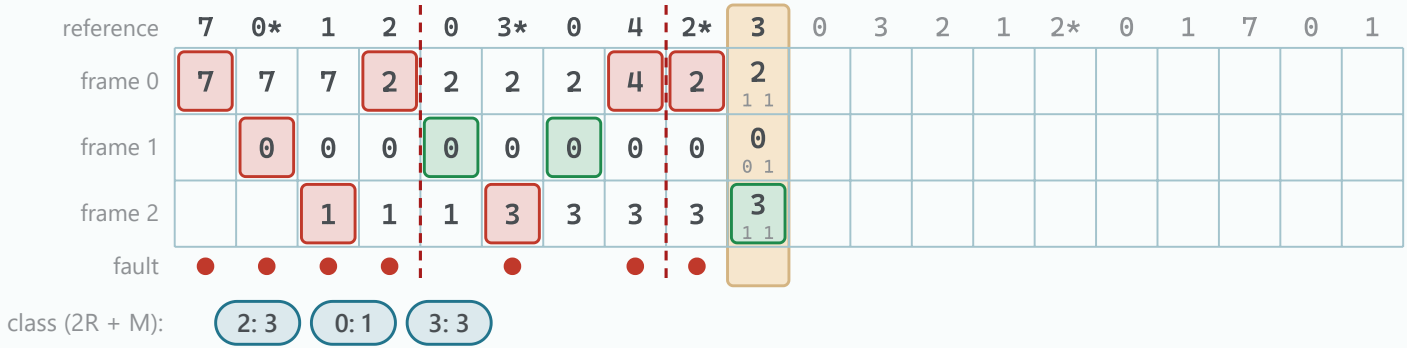


9. reference 2*: fault; classes (4→0, 0→1, 3→1); **4**, in the lowest class, leaves. Faults: 7

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

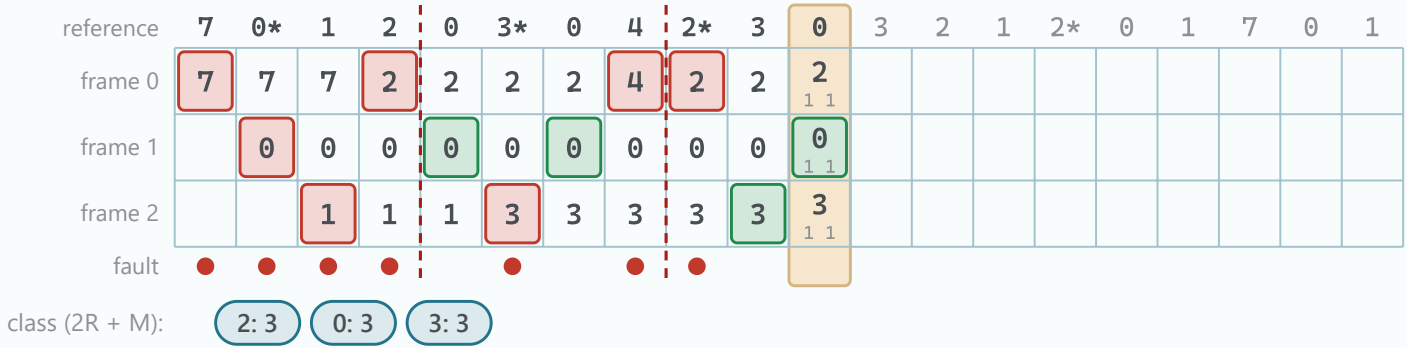


10. reference 3: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

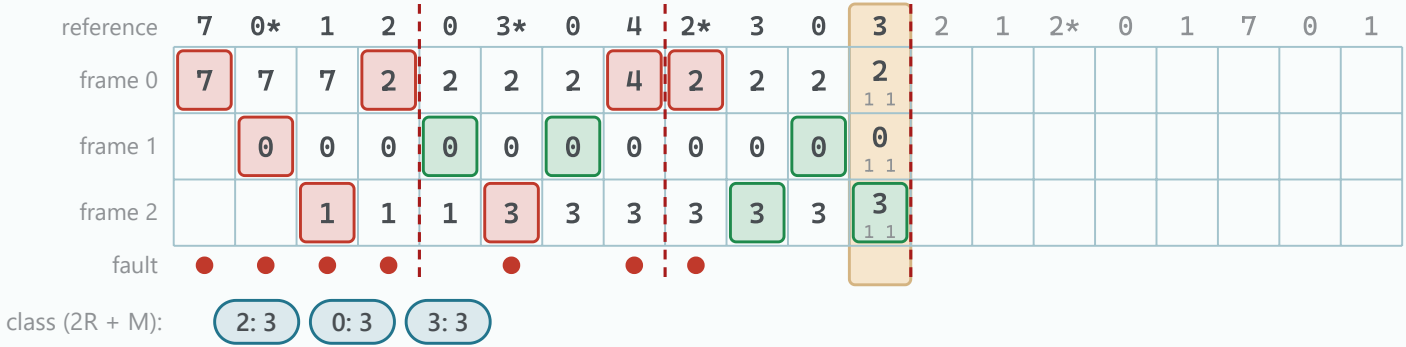


11. reference 0: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

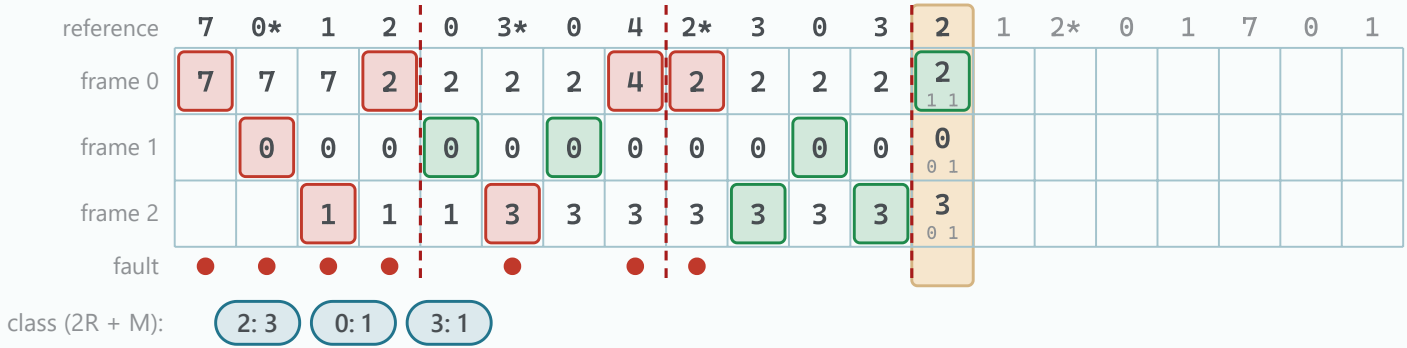


12. reference 3: hit; R ← 1. Clock tick: all R ← 0.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

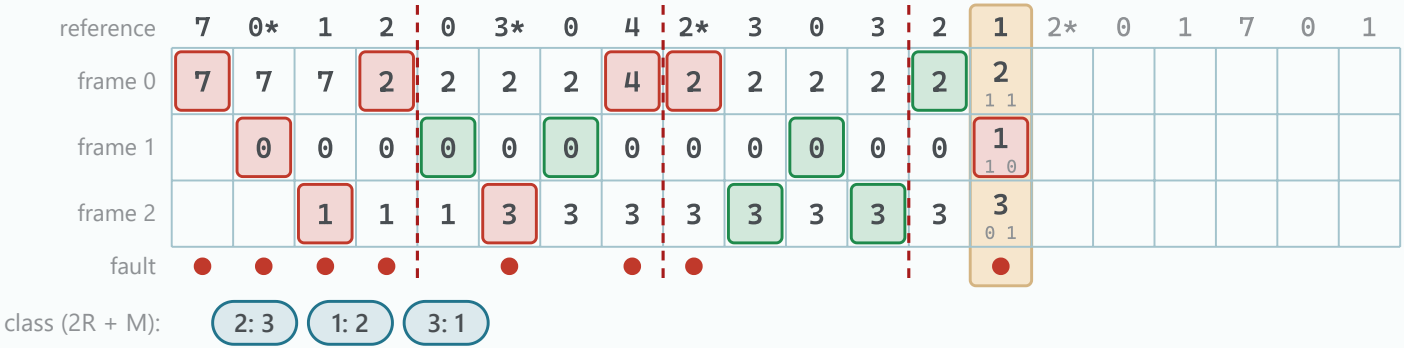


13. reference 2: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

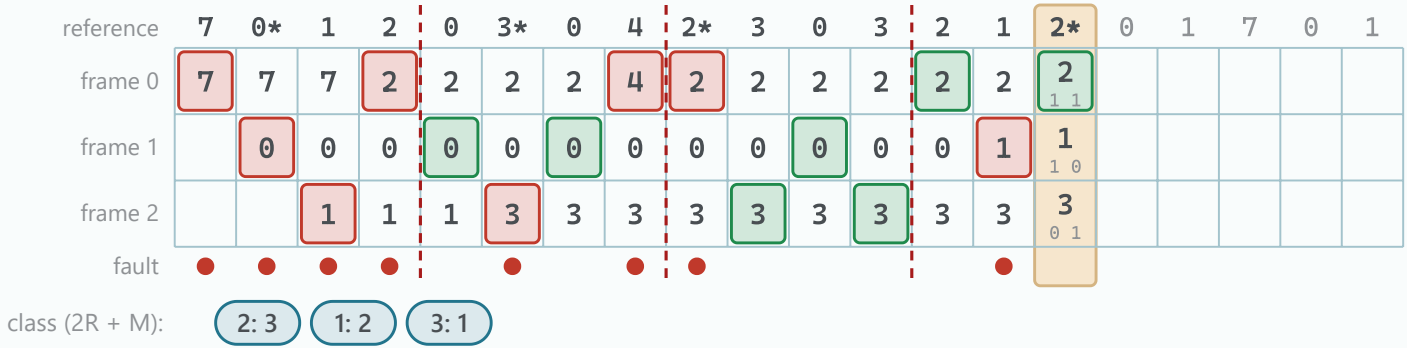


14. reference 1: fault; classes (2→3, 0→1, 3→1); **0**, in the lowest class, leaves (dirty → written to disk first). Faults: 8

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

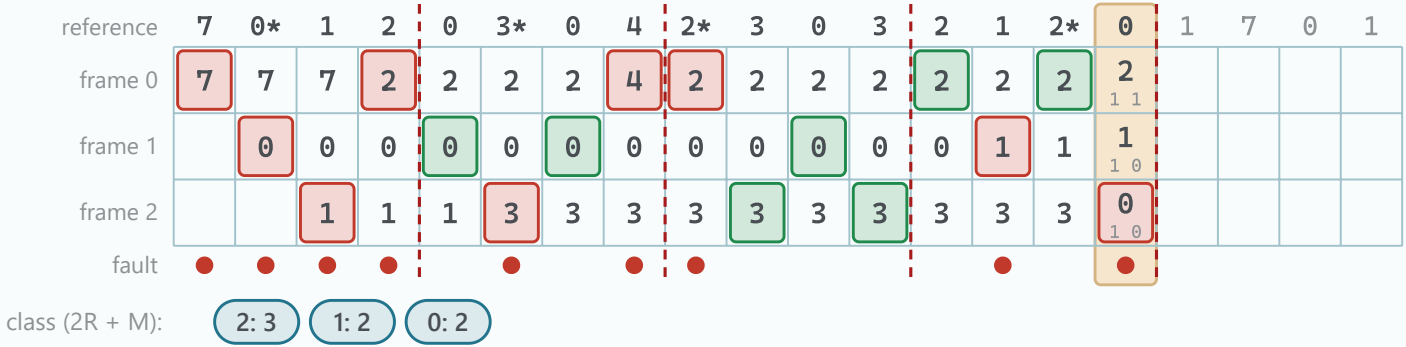


15. reference 2*: hit; R ← 1, write → M ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.



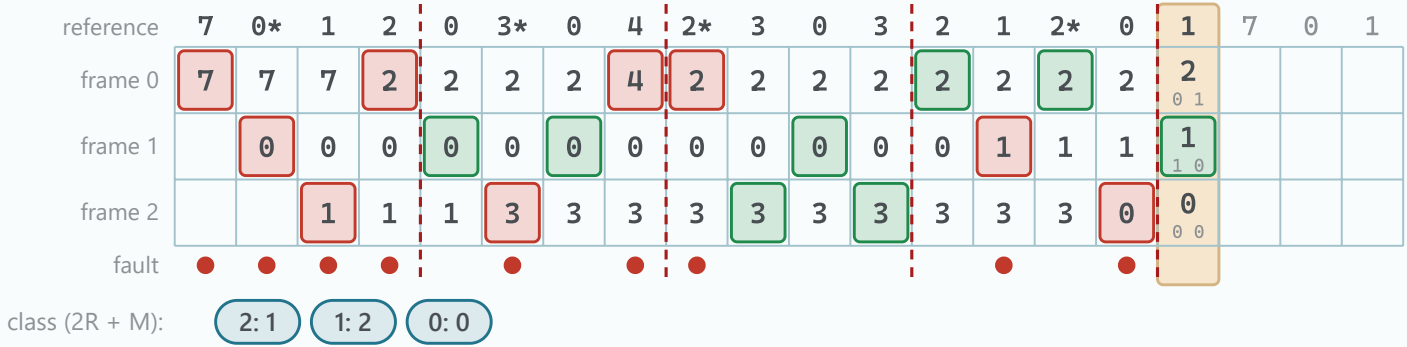
16. reference 0: fault; classes (2→3, 1→2, 3→1); **3**, in the lowest class, leaves (dirty → written to disk first). Faults: 9

Clock tick: all R ← 0.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

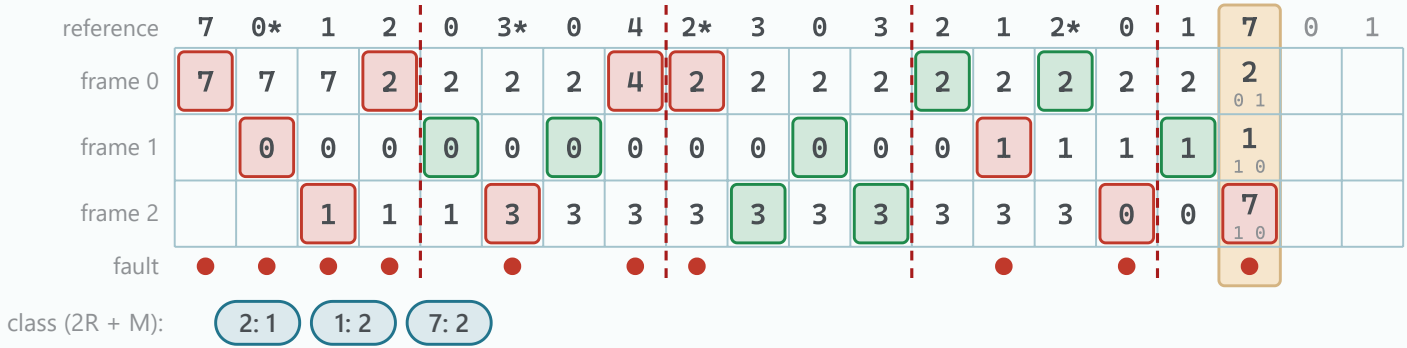


17. reference 1: hit; R ← 1.

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

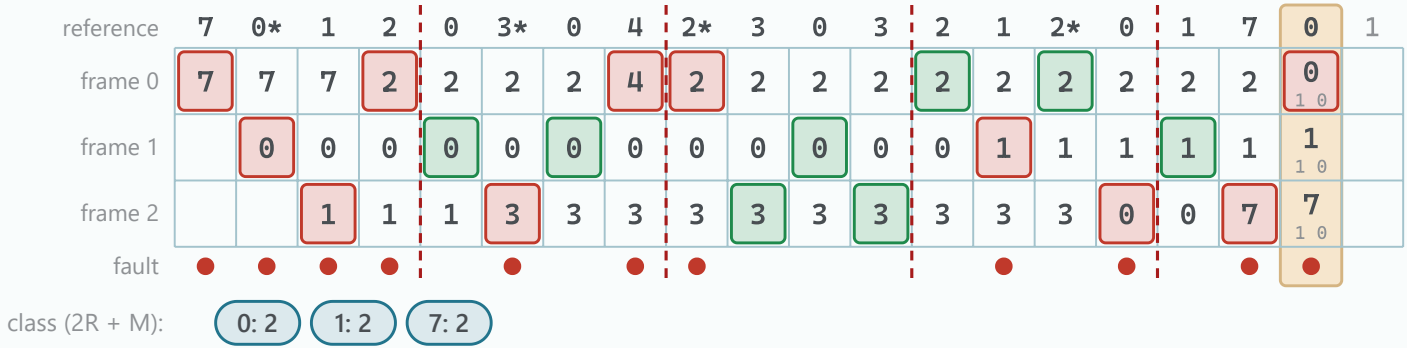


18. reference 7: fault; classes (2→1, 1→2, 0→0); **0**, in the lowest class, leaves. Faults: 10

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

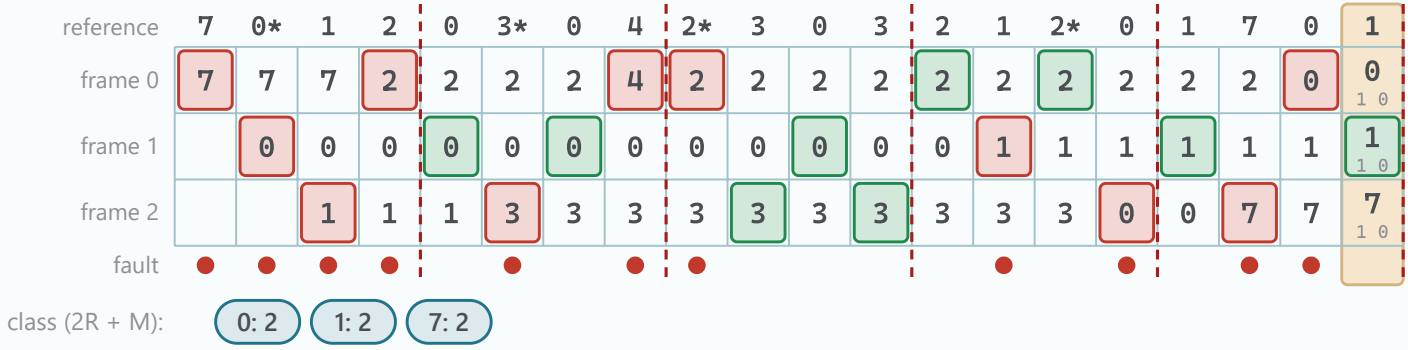


19. reference 0: fault; classes (2→1, 1→2, 7→2); **2**, in the lowest class, leaves (dirty → written to disk first). Faults: 11

* write access · small numbers in the bottom row: R M · a tick every 4 references.

NRU: Not Recently Used

- Each page has an **R** (referenced) and an **M** (modified) bit; the clock interrupt clears the R bits periodically.
- Classes: 0 = (0,0), 1 = (0,1), 2 = (1,0), 3 = (1,1). A page from the **lowest** nonempty class is evicted (ties here go to the oldest loaded).
- Class 1 is possible: a modified page's R bit was cleared by a tick.
- **LRU vs. NRU:** LRU tracks the exact *order* of use; NRU only asks whether the page was used in the last clock interval and whether it is dirty: crude but cheap.

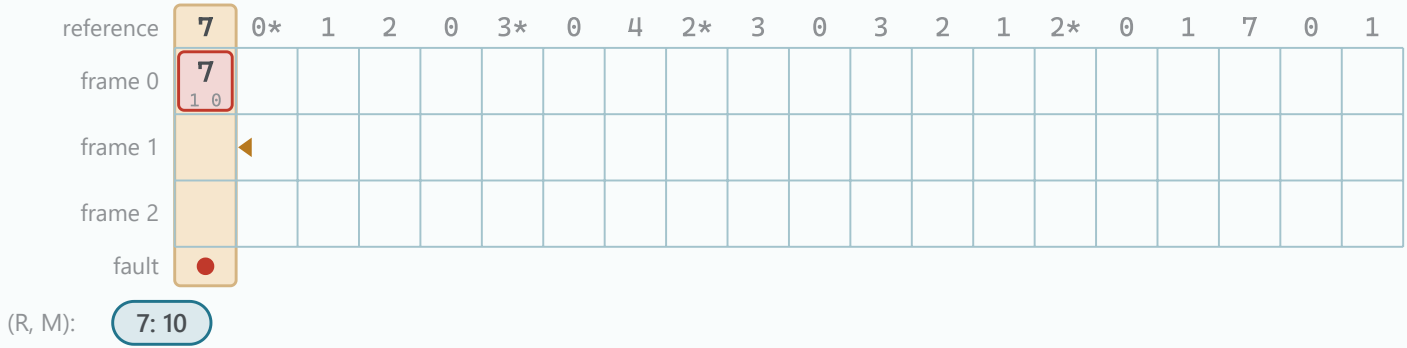


20. reference 1: hit; R ← 1. Clock tick: all R ← 0. **Total: 11 faults, 3 dirty-page writes.**

* write access · small numbers in the bottom row: R M · a tick every 4 references.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

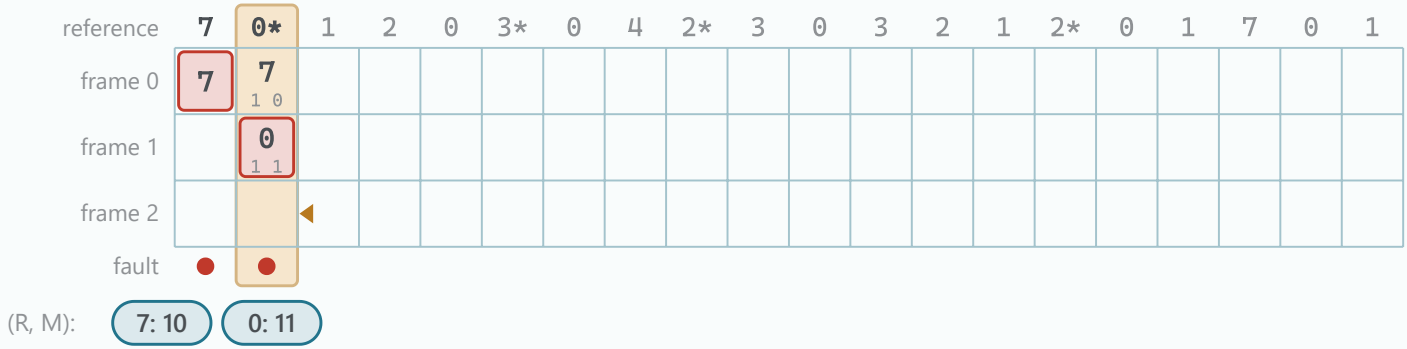


1. reference 7: fault; loaded into free frame 0. Faults: 1

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

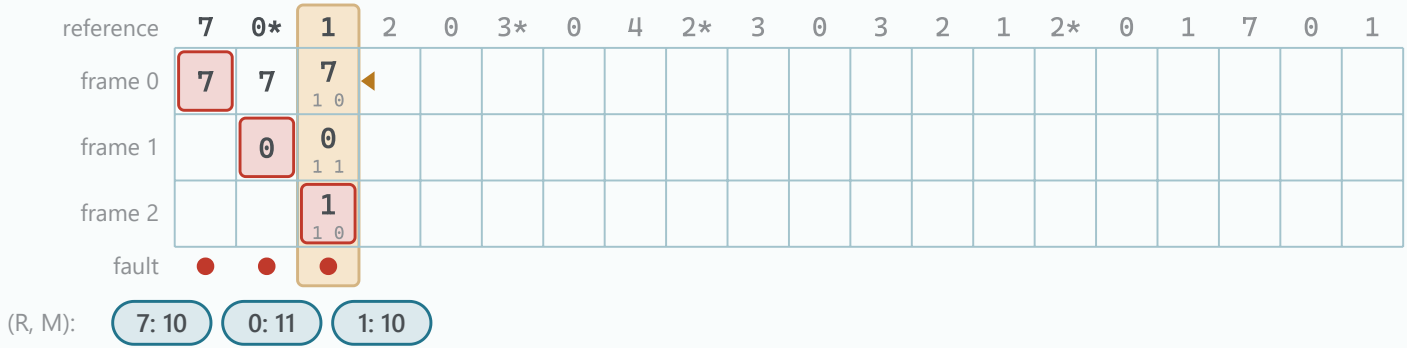


2. reference 0*: fault; loaded into free frame 1. Faults: 2

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

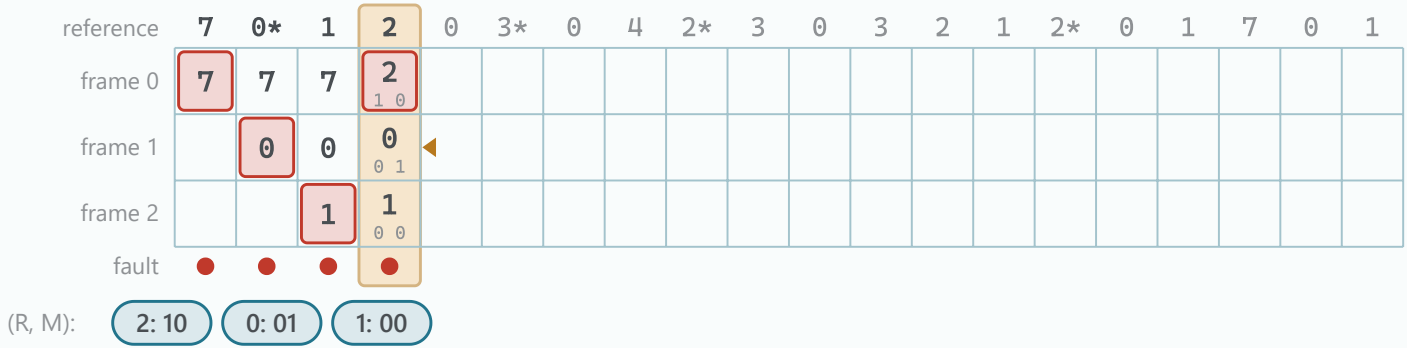


3. reference 1: fault; loaded into free frame 2. Faults: 3

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

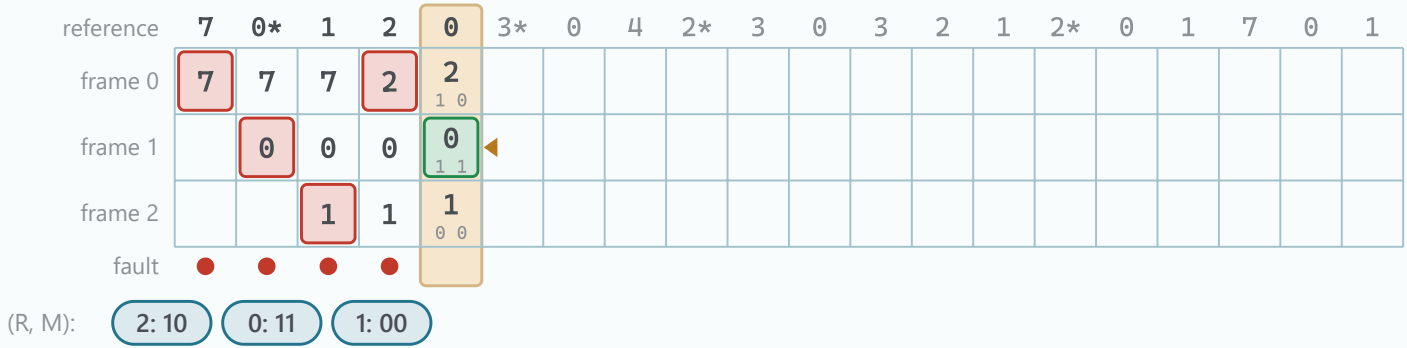


4. reference 2: fault; a 2-pass scan picks **7** from the lowest class. Faults: 4

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

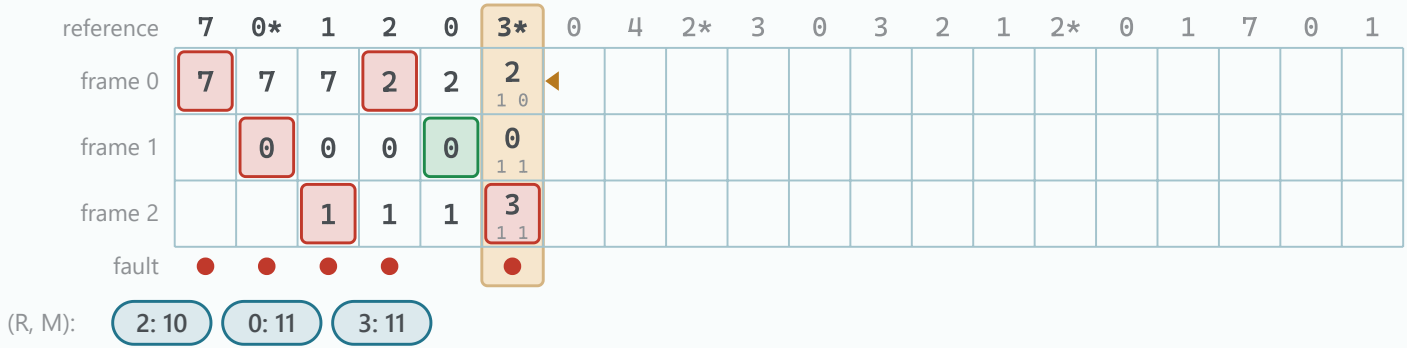


5. reference 0: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

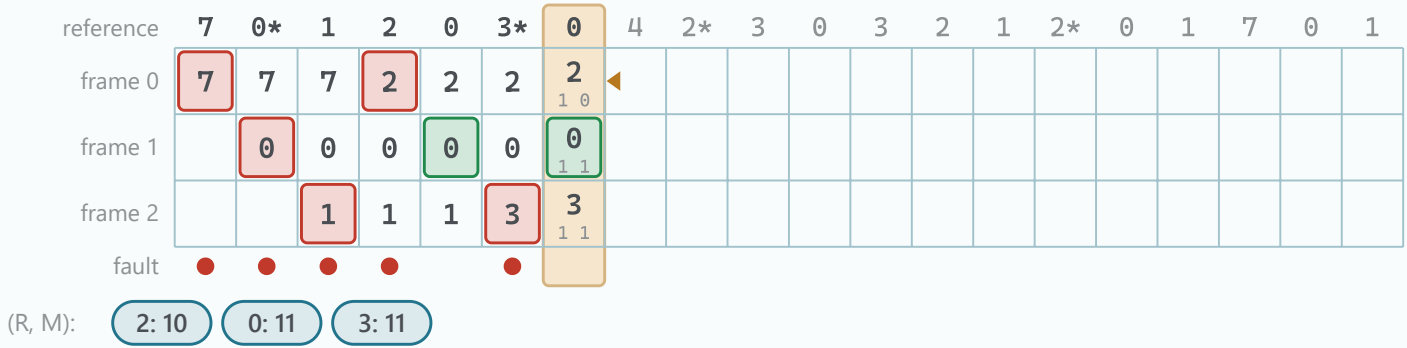


6. reference 3*: fault; a 1-pass scan picks **1** from the lowest class. Faults: 5

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

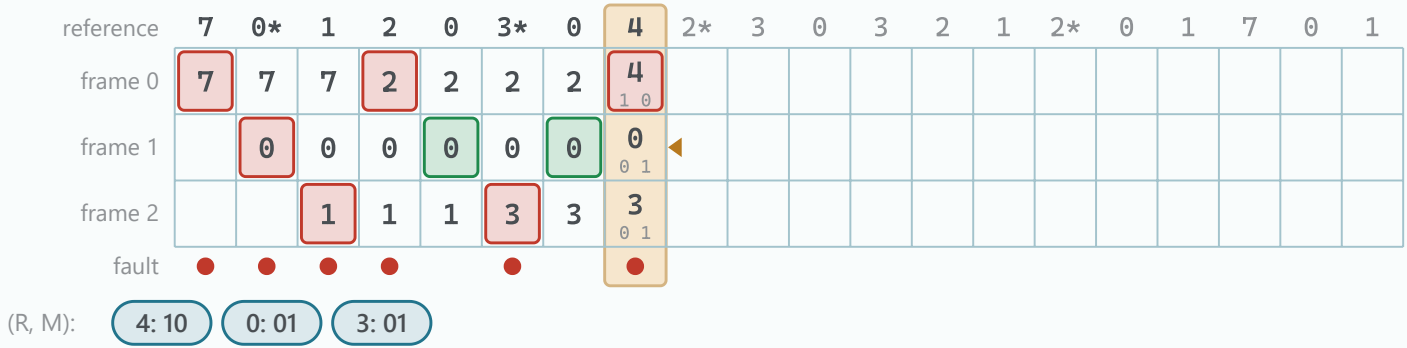


7. reference 0: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

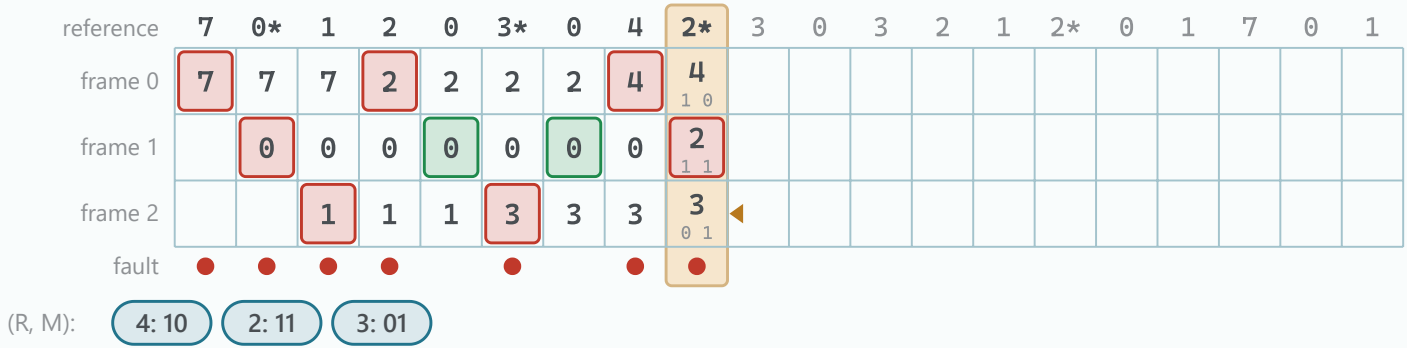


8. reference 4: fault; a 2-pass scan picks **2** from the lowest class. Faults: 6

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

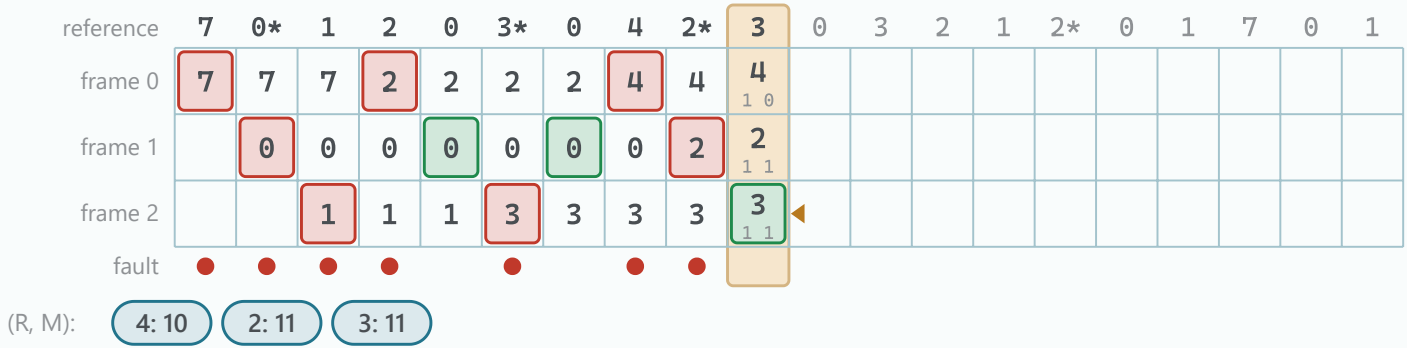


9. reference 2*: fault; a 1-pass scan picks **0** from the lowest class; it is dirty, so it is written to disk first. Faults: 7

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

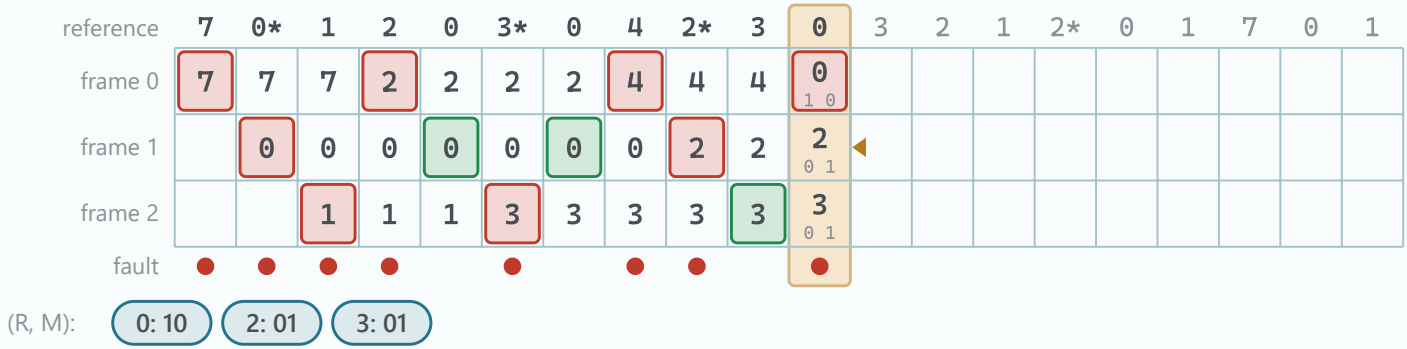


10. reference 3: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

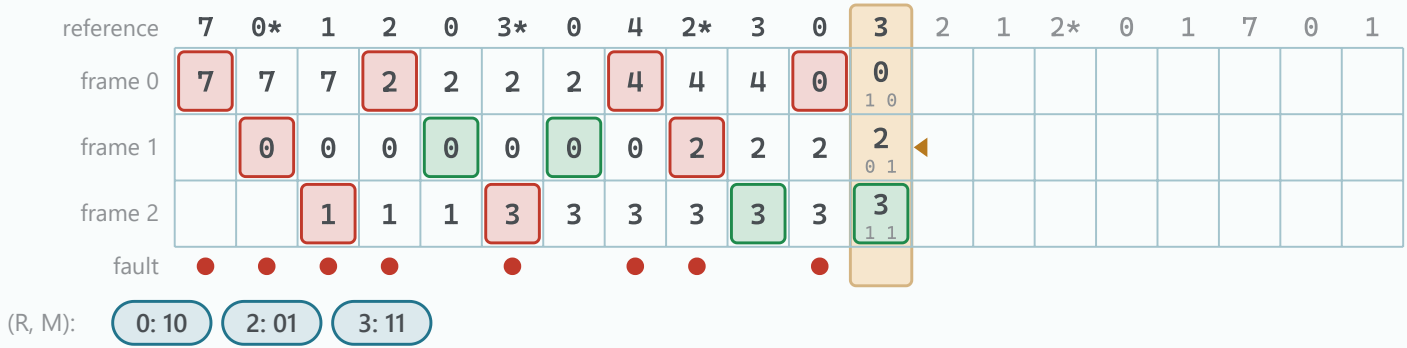


11. reference 0: fault; a 2-pass scan picks **4** from the lowest class. Faults: 8

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

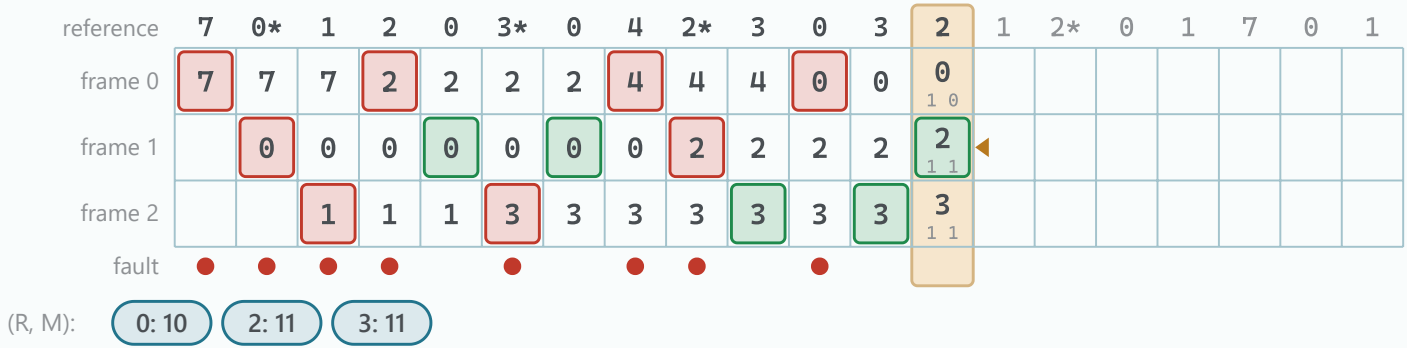


12. reference 3: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

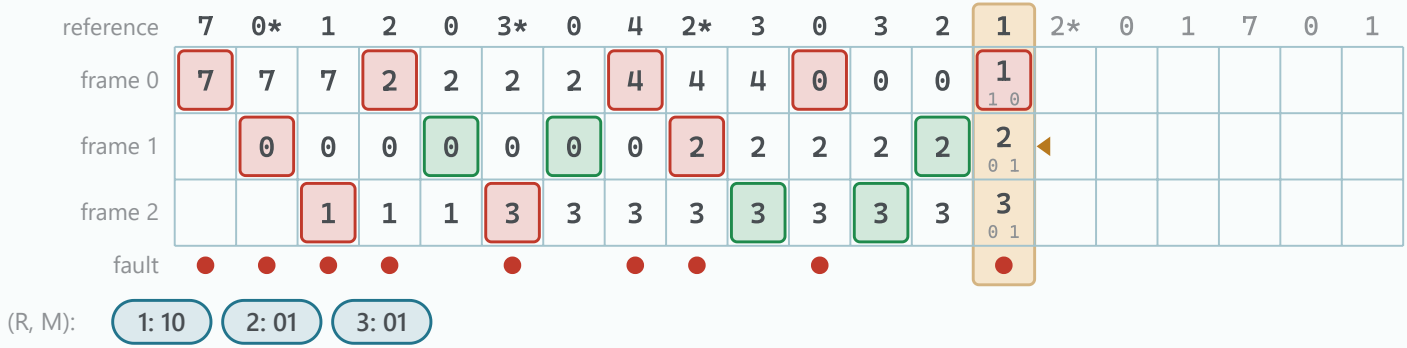


13. reference 2: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

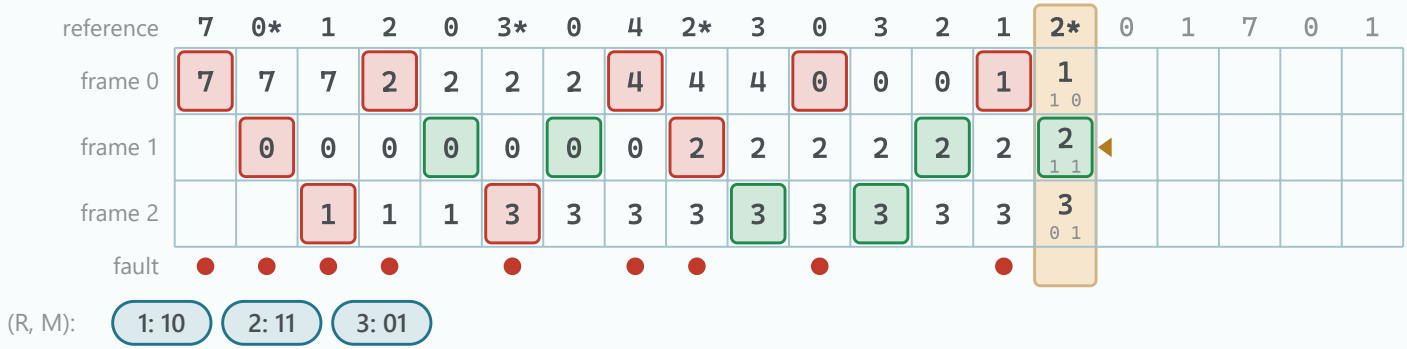


14. reference 1: fault; a 2-pass scan picks **0** from the lowest class. Faults: 9

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

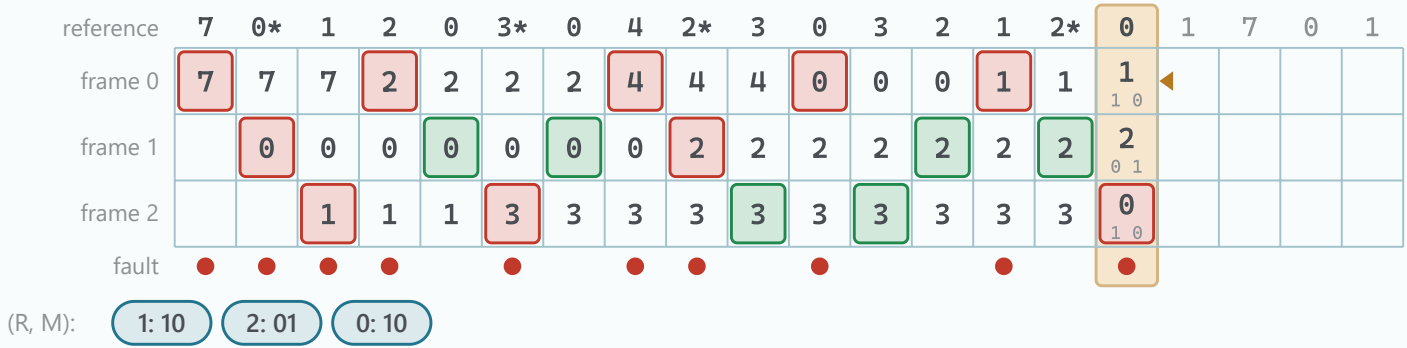


15. reference 2*: hit; R ← 1, write → M ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

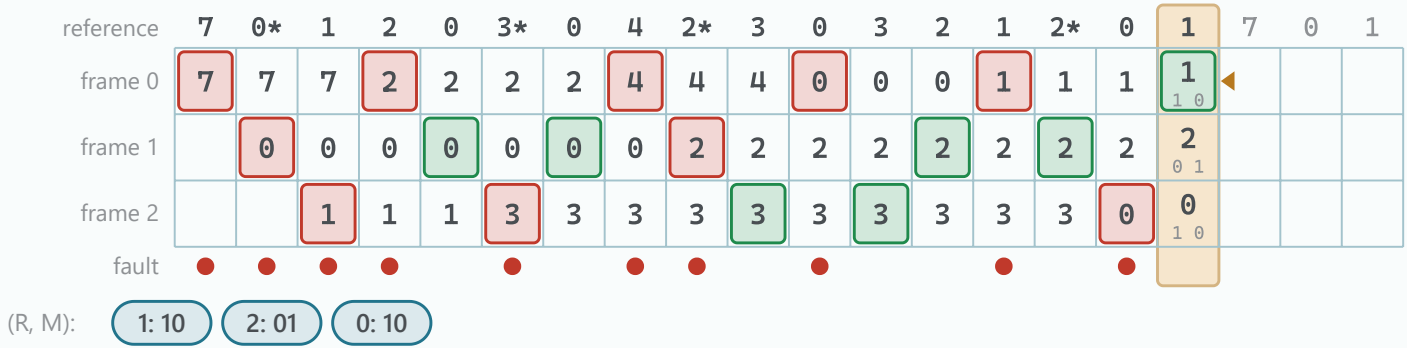


16. reference 0: fault; a 1-pass scan picks **3** from the lowest class; it is dirty, so it is written to disk first. Faults: 10

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

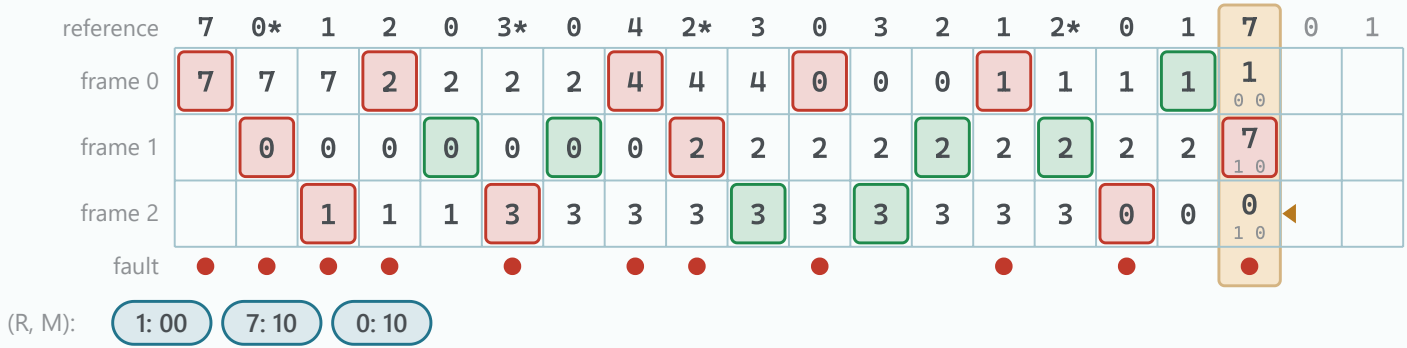


17. reference 1: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

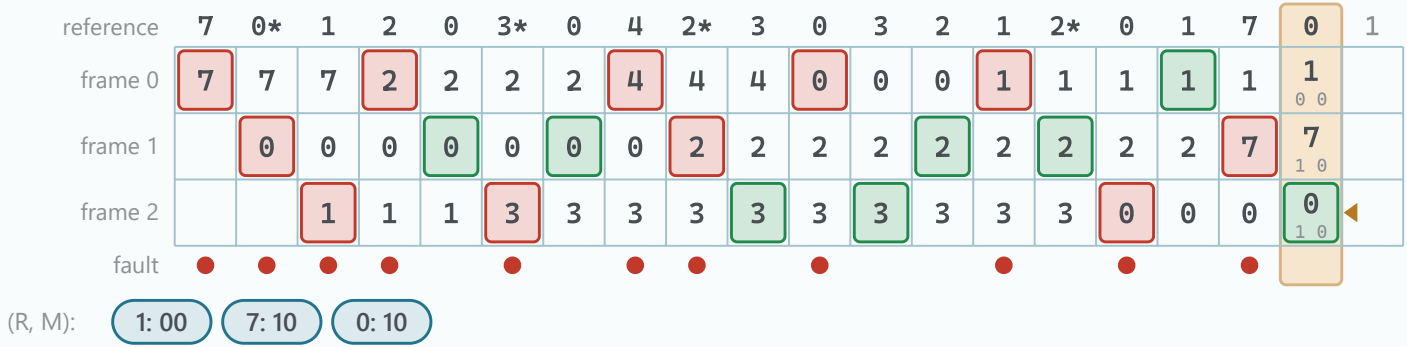


18. reference 7: fault; a 1-pass scan picks **2** from the lowest class; it is dirty, so it is written to disk first. Faults: 11

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

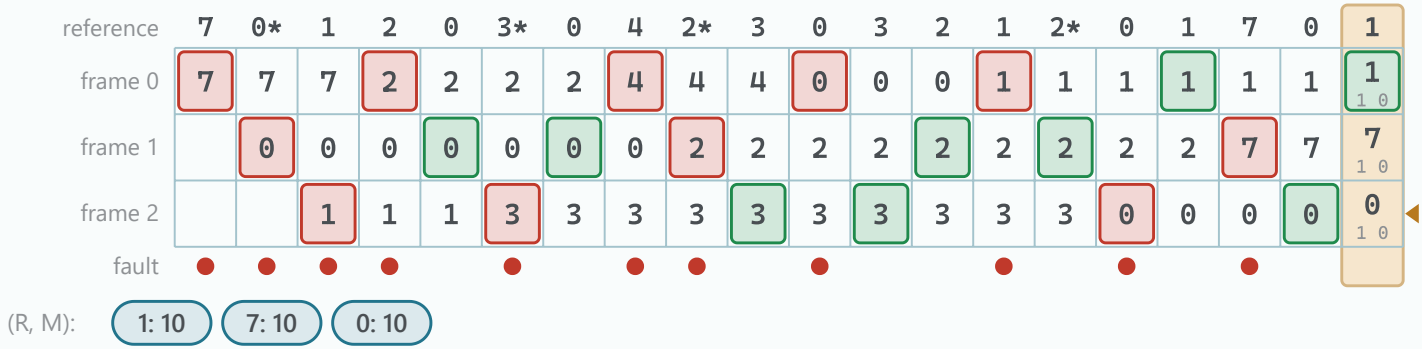


19. reference 0: hit; R ← 1.

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

Enhanced Clock: (R, M) Classes

- The clock looks at **M** as well as R; evicting a clean page needs no I/O.
- **Pass 1:** starting at the hand, look for (0,0) without touching any bits.
- **Pass 2:** look for (0,1); the hand clears the R bit of every page it passes. If none is found, go back to pass 1 (now all R = 0).
- Old macOS, and Silberschatz's "enhanced second-chance" algorithm.

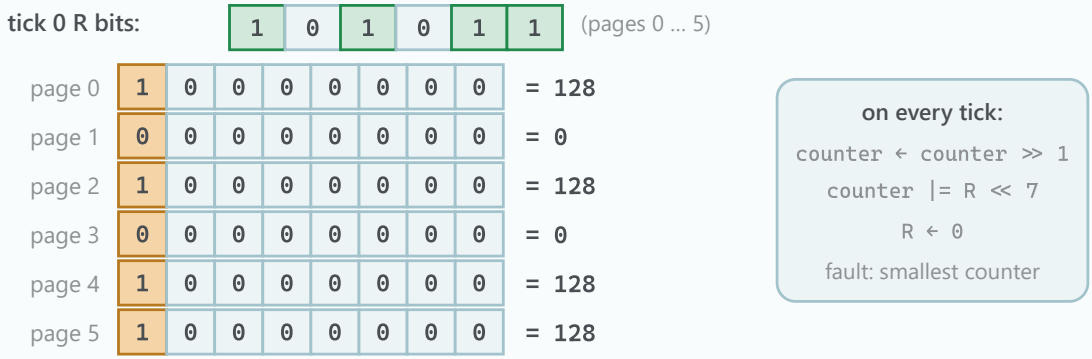


20. reference 1: hit; R ← 1. **Total: 11 faults, 3 dirty writes.**

* write access · small numbers: R M. The same string as NRU; here R is cleared only as the hand passes, with no clock tick.

NFU and Aging

- **NFU** (not frequently used): a software counter per page; on every tick, counter += R. The page with the smallest counter is evicted.
- NFU **never forgets**: a page heavily used long ago stays forever. Access times aren't recorded, only the counter.
- **Aging:** first shift right, then add R at the *leftmost* bit → the recent past weighs more.
- A good approximation of LRU; the differences: order within a tick is unknown, and 8 bits remember only 8 ticks.

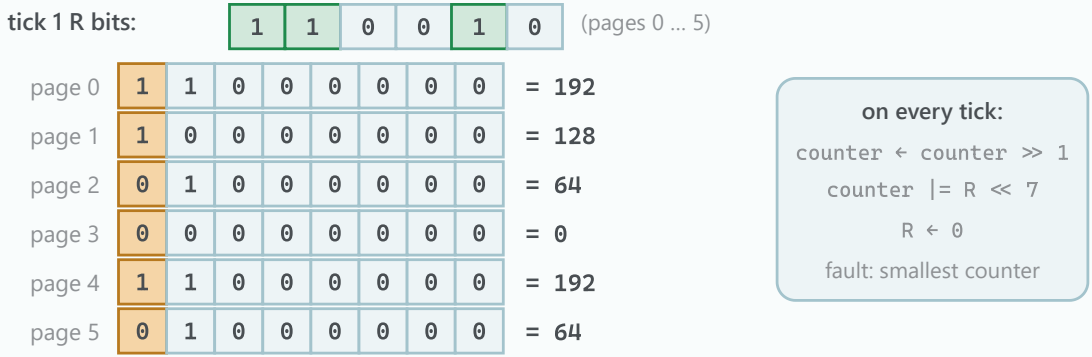


Tick 0: R bits 1 0 1 0 1 1 added on the left.

Tanenbaum MOS Figure 3-17; counter values recomputed from the R bits.

NFU and Aging

- **NFU** (not frequently used): a software counter per page; on every tick, counter += R. The page with the smallest counter is evicted.
- NFU **never forgets**: a page heavily used long ago stays forever. Access times aren't recorded, only the counter.
- **Aging:** first shift right, then add R at the *leftmost* bit → the recent past weighs more.
- A good approximation of LRU; the differences: order within a tick is unknown, and 8 bits remember only 8 ticks.

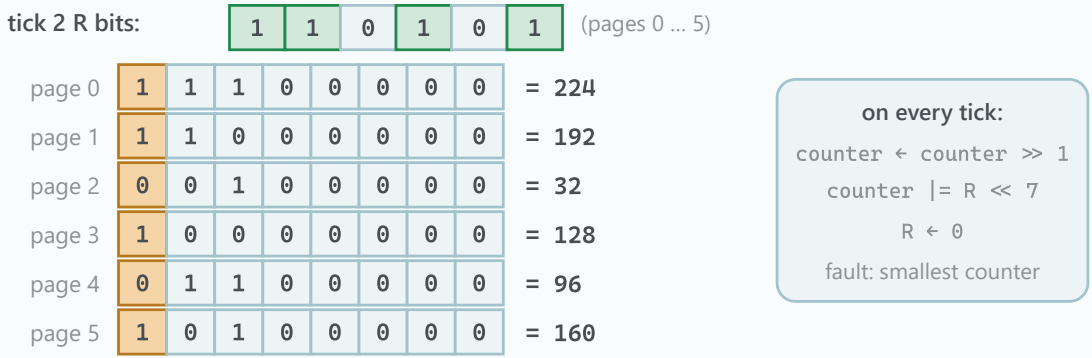


Tick 1: R bits 1 1 0 0 1 0 added on the left.

Tanenbaum MOS Figure 3-17; counter values recomputed from the R bits.

NFU and Aging

- **NFU** (not frequently used): a software counter per page; on every tick, counter += R. The page with the smallest counter is evicted.
- NFU **never forgets**: a page heavily used long ago stays forever. Access times aren't recorded, only the counter.
- **Aging:** first shift right, then add R at the *leftmost* bit → the recent past weighs more.
- A good approximation of LRU; the differences: order within a tick is unknown, and 8 bits remember only 8 ticks.

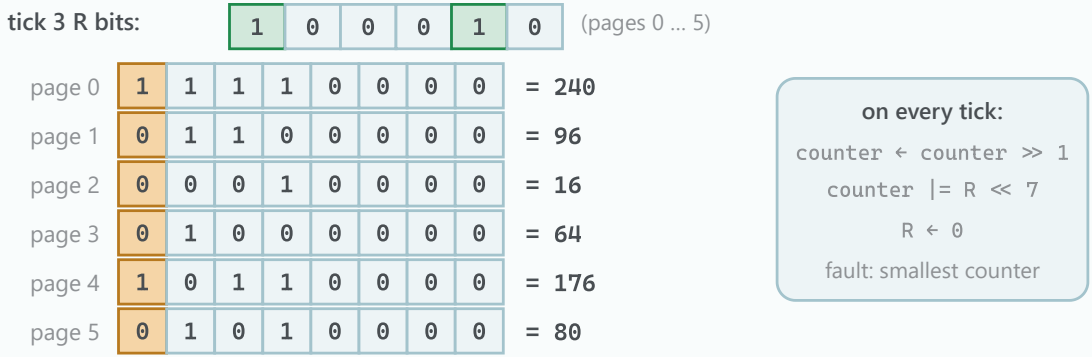


Tick 2: R bits 1 1 0 1 0 1 added on the left.

Tanenbaum MOS Figure 3-17; counter values recomputed from the R bits.

NFU and Aging

- **NFU** (not frequently used): a software counter per page; on every tick, counter += R. The page with the smallest counter is evicted.
- NFU **never forgets**: a page heavily used long ago stays forever. Access times aren't recorded, only the counter.
- **Aging:** first shift right, then add R at the *leftmost* bit → the recent past weighs more.
- A good approximation of LRU; the differences: order within a tick is unknown, and 8 bits remember only 8 ticks.

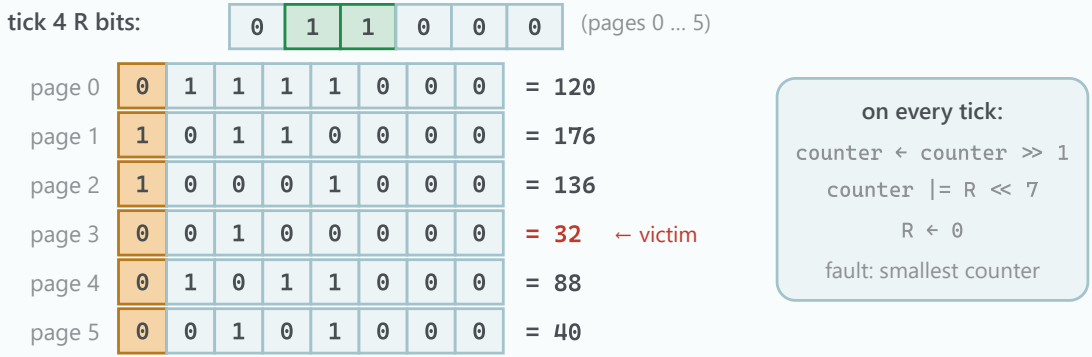


Tick 3: R bits 1 0 0 0 1 0 added on the left.

Tanenbaum MOS Figure 3-17; counter values recomputed from the R bits.

NFU and Aging

- **NFU** (not frequently used): a software counter per page; on every tick, counter += R. The page with the smallest counter is evicted.
- NFU **never forgets**: a page heavily used long ago stays forever. Access times aren't recorded, only the counter.
- **Aging:** first shift right, then add R at the *leftmost* bit → the recent past weighs more.
- A good approximation of LRU; the differences: order within a tick is unknown, and 8 bits remember only 8 ticks.

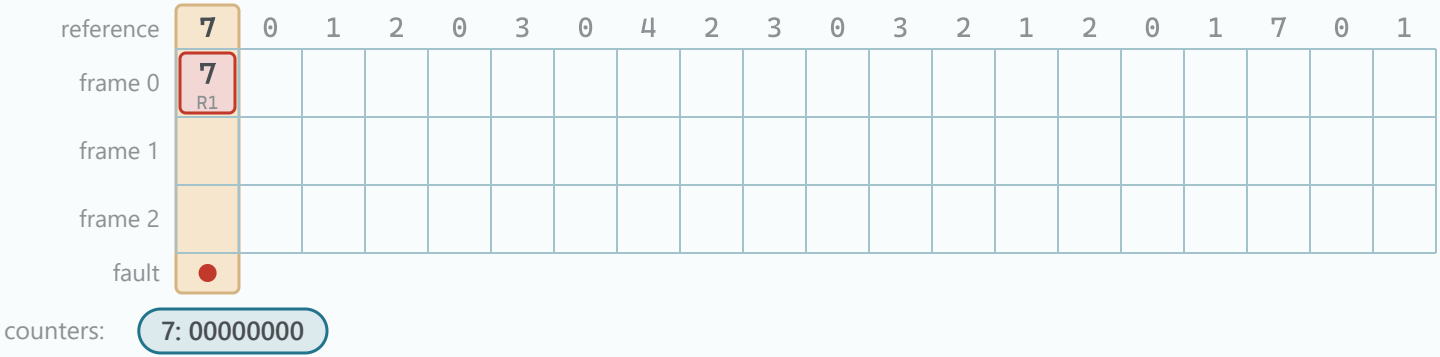


Tick 4: R bits 0 1 1 0 0 0 added on the left. A fault now would evict **page 3**, whose counter is smallest.

Tanenbaum MOS Figure 3-17; counter values recomputed from the R bits.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

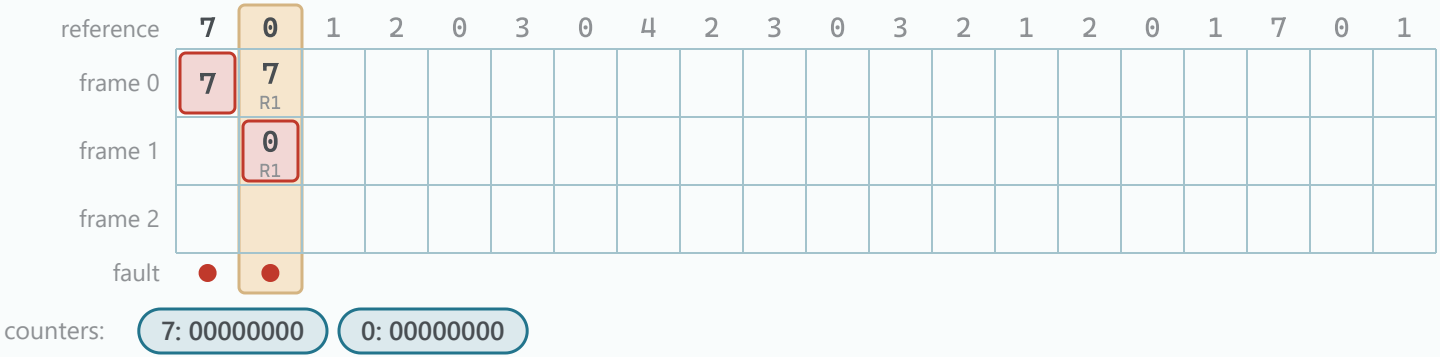


1. reference 7: fault; loaded into free frame 0. Faults: 1

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

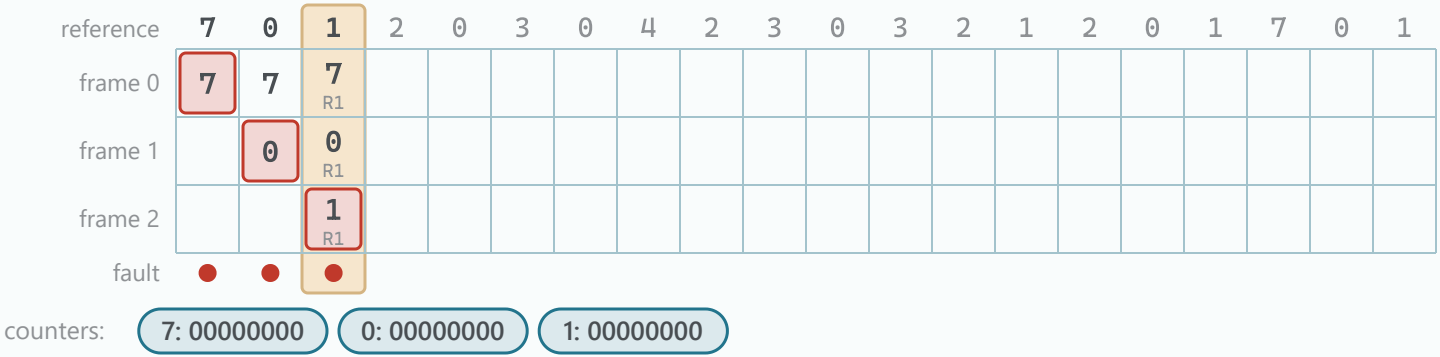


2. reference 0: fault; loaded into free frame 1. Faults: 2

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

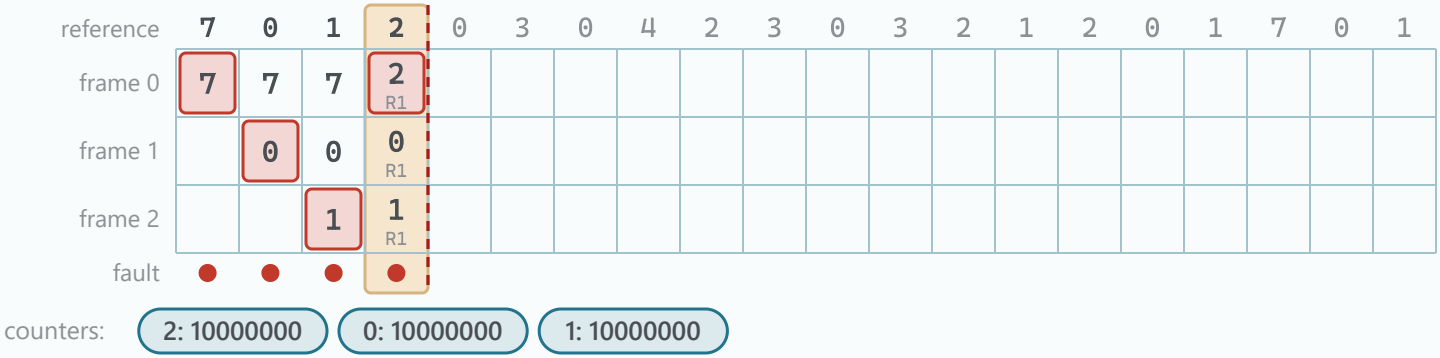


3. reference 1: fault; loaded into free frame 2. Faults: 3

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

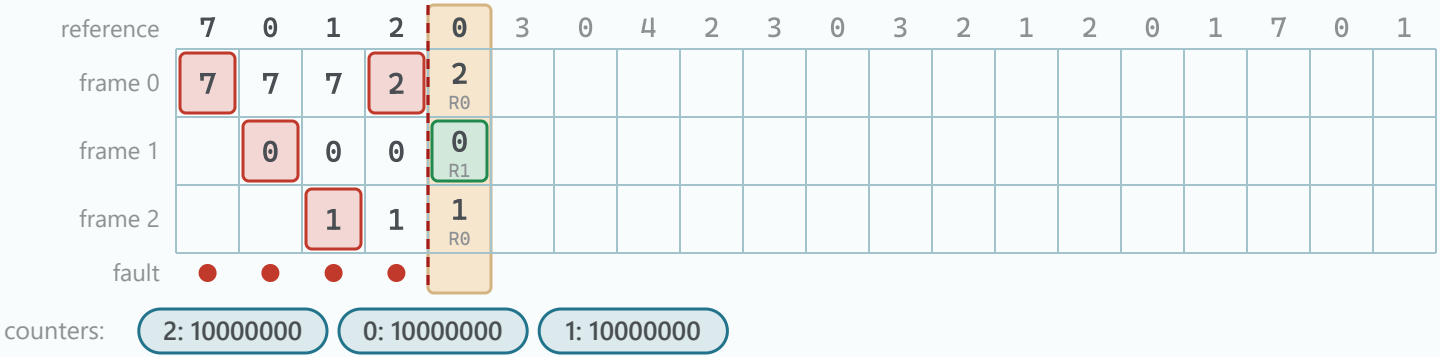


4. reference 2: fault; R and counter together (R as the top bit): 7: 100000000, 0: 100000000, 1: 100000000 → the smallest, **7**, leaves. Faults: 4 Tick: counters shifted, R bits added on the left.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

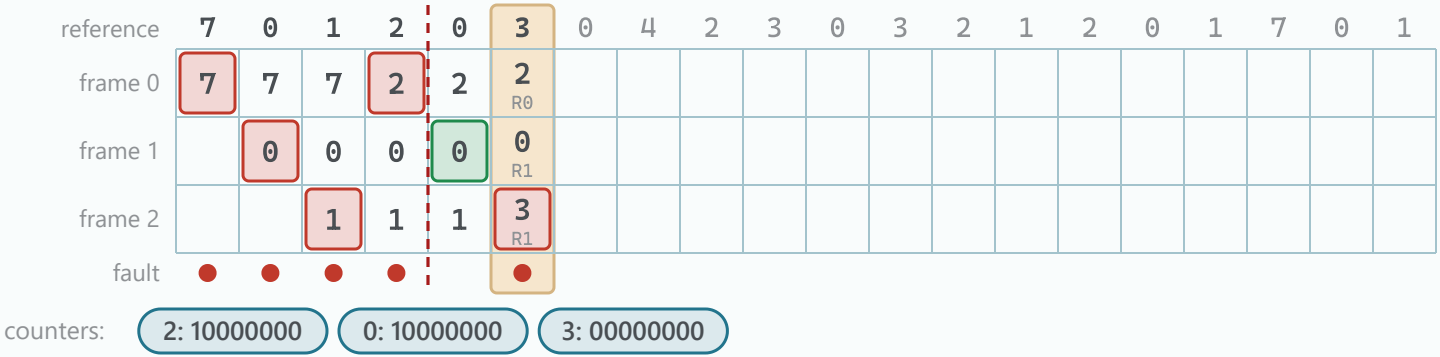


5. reference 0: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

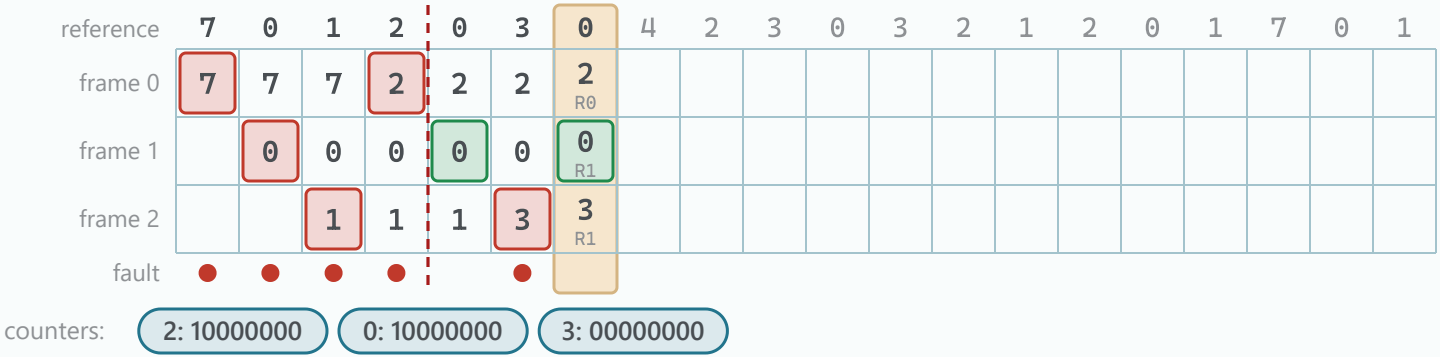


6. reference 3: fault; R and counter together (R as the top bit): 2: 010000000, 0: 110000000, 1: 010000000 → the smallest, **1**, leaves. Faults: 5

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

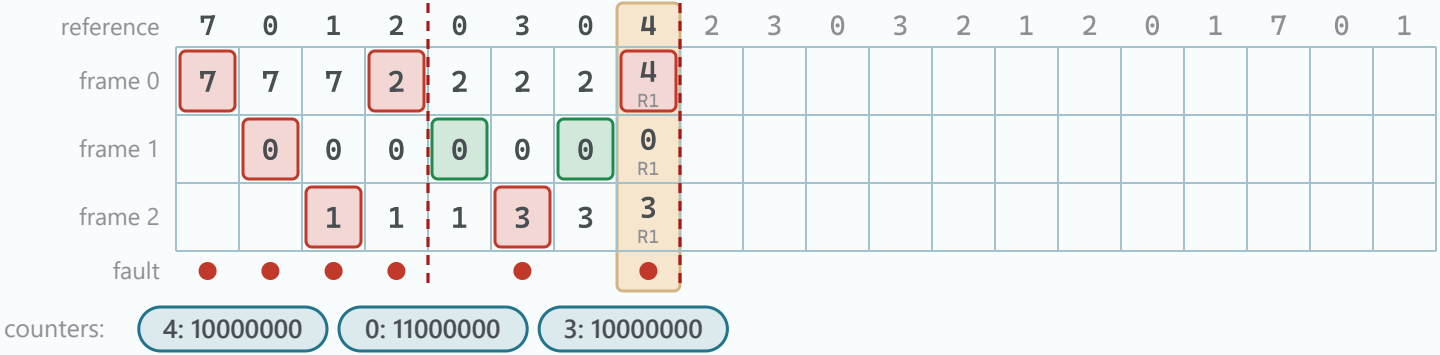


7. reference 0: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

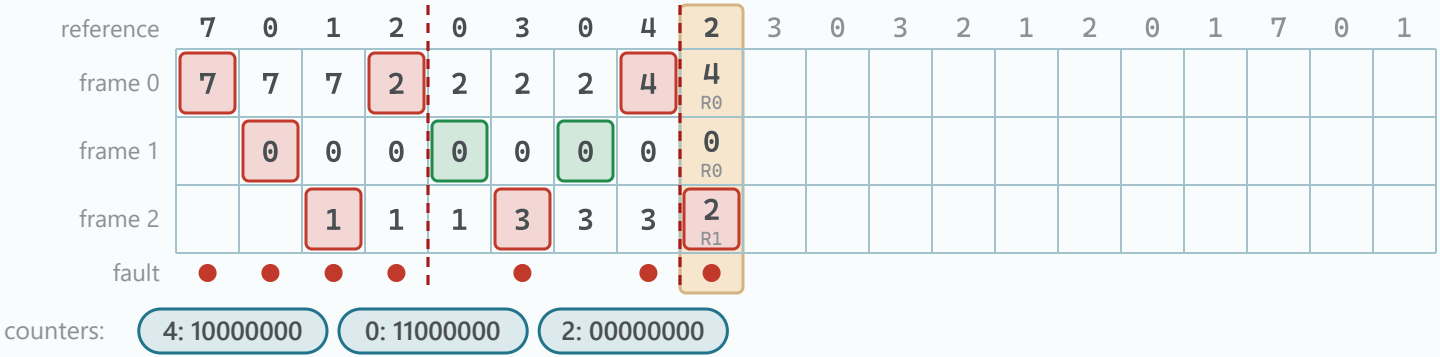


8. reference 4: fault; R and counter together (R as the top bit): 2: 010000000, 0: 110000000, 3: 100000000 → the smallest, **2**, leaves. Faults: 6 Tick: counters shifted, R bits added on the left.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

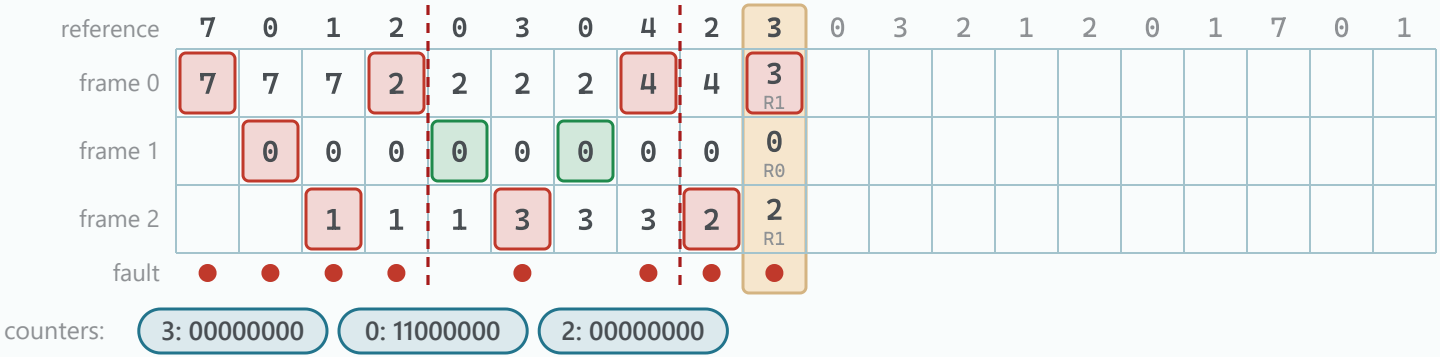


9. reference 2: fault; R and counter together (R as the top bit): 4: 010000000, 0: 011000000, 3: 010000000 → the smallest, **3**, leaves. Faults: 7

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

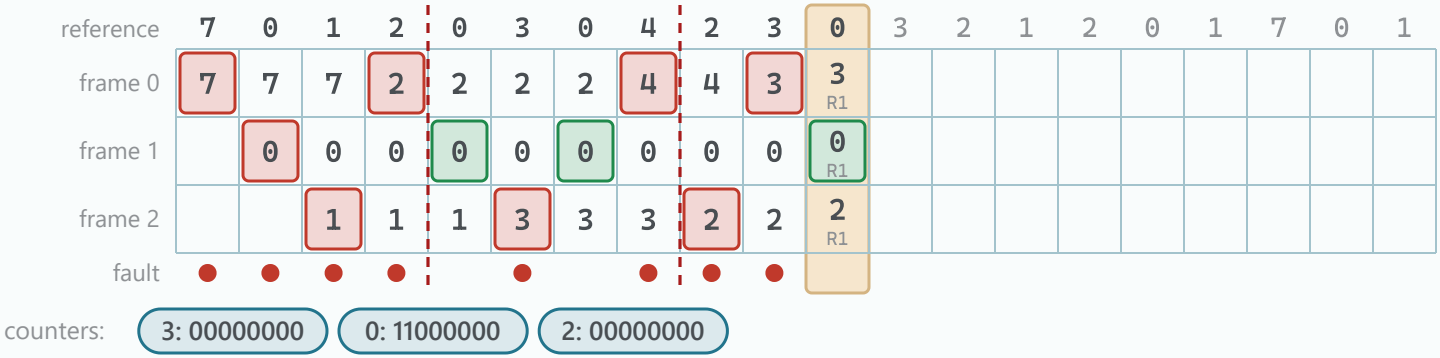


10. reference 3: fault; R and counter together (R as the top bit): 4: 010000000, 0: 011000000, 2: 100000000 → the smallest, **4**, leaves. Faults: 8

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

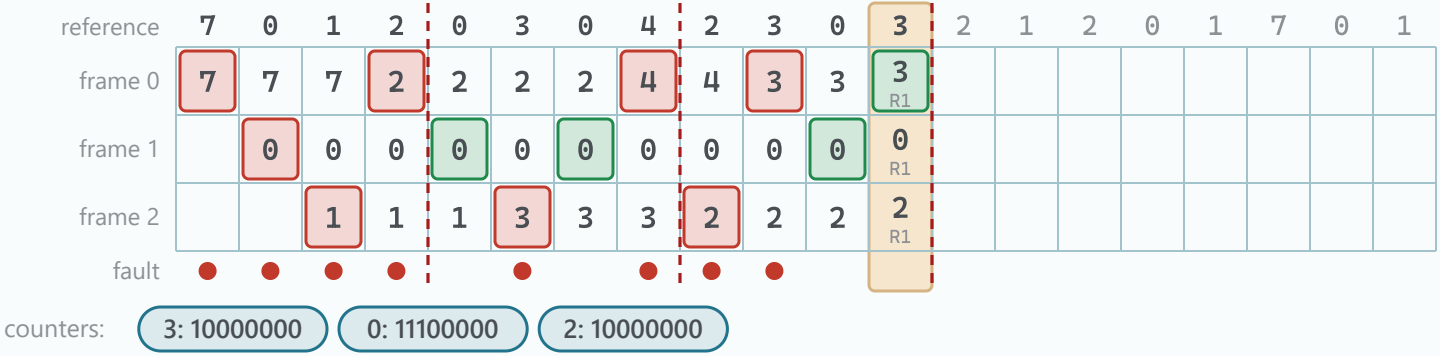


11. reference 0: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

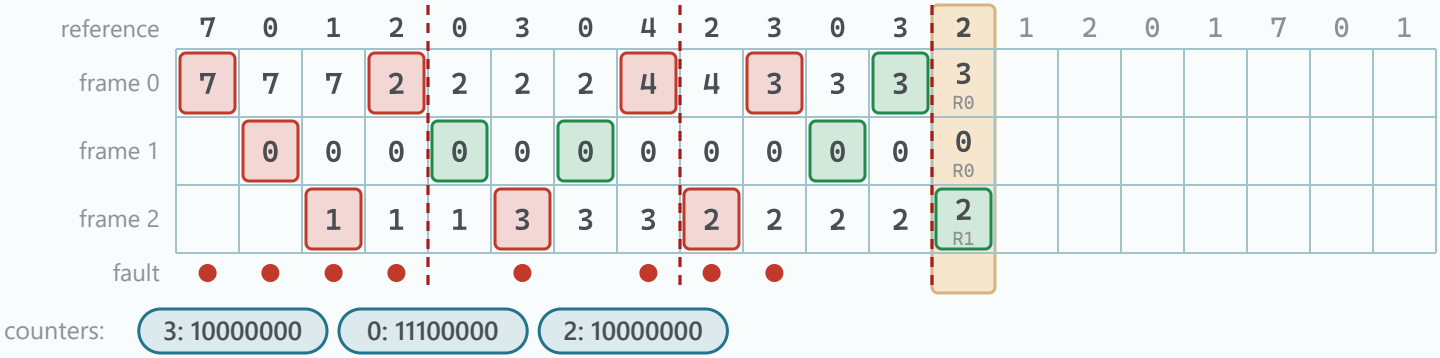


12. reference 3: hit; R ← 1. Tick: counters shifted, R bits added on the left.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

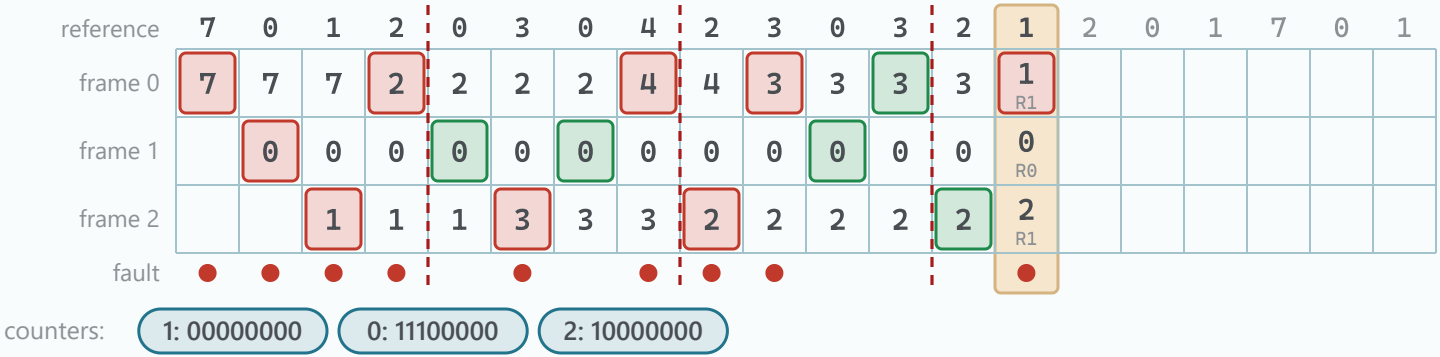


13. reference 2: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

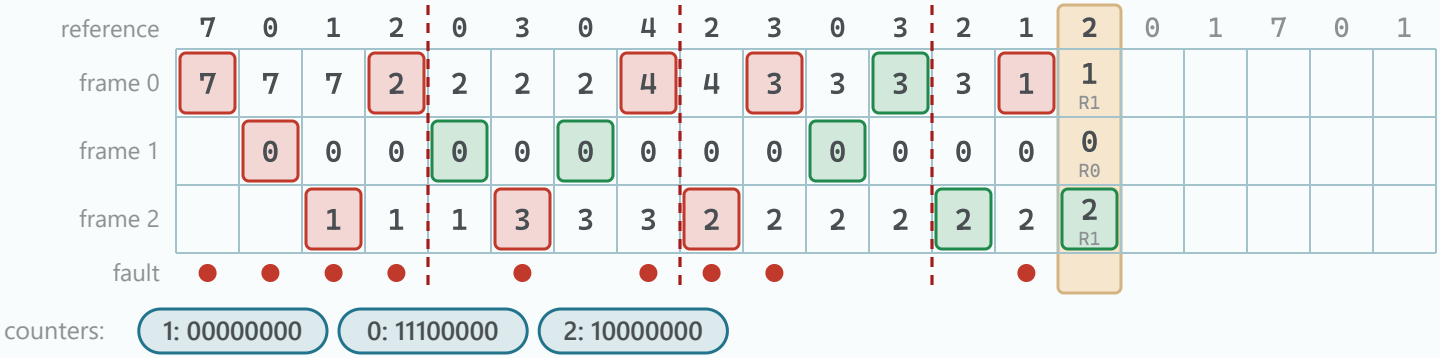


14. reference 1: fault; R and counter together (R as the top bit): 3: 010000000, 0: 011100000, 2: 110000000 → the smallest, **3**, leaves. Faults: 9

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

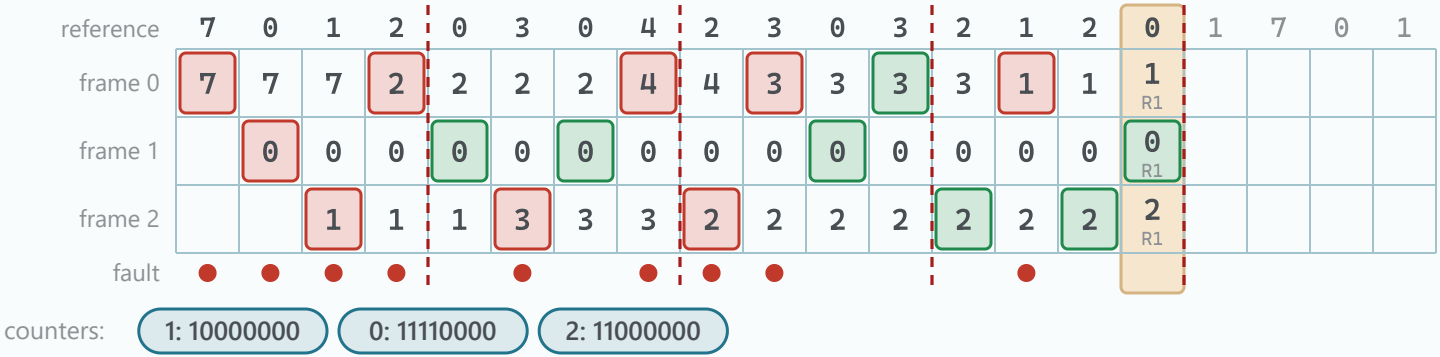


15. reference 2: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

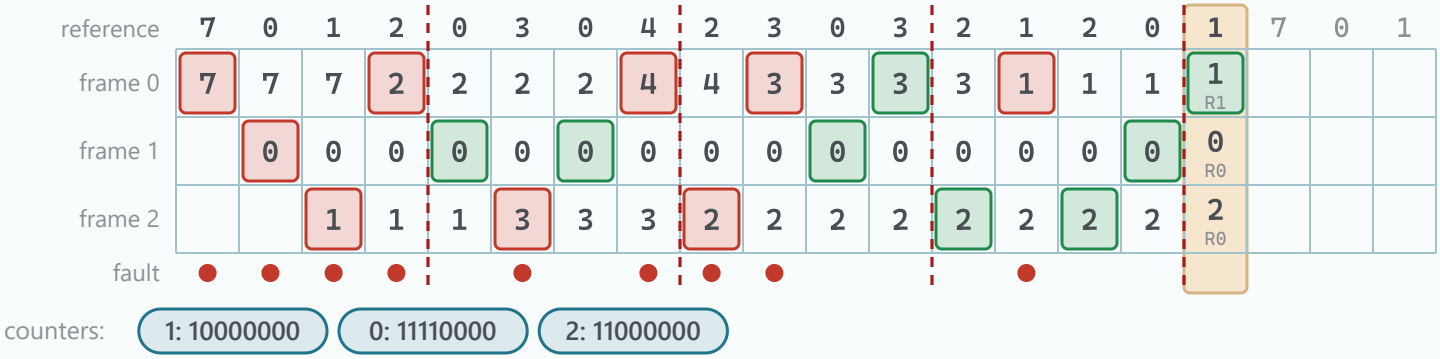


16. reference 0: hit; R ← 1. Tick: counters shifted, R bits added on the left.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

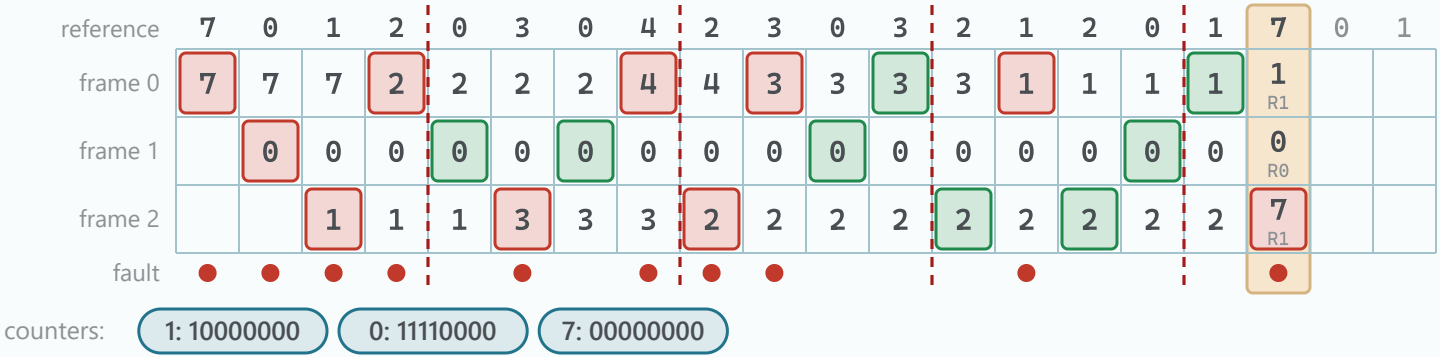


17. reference 1: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

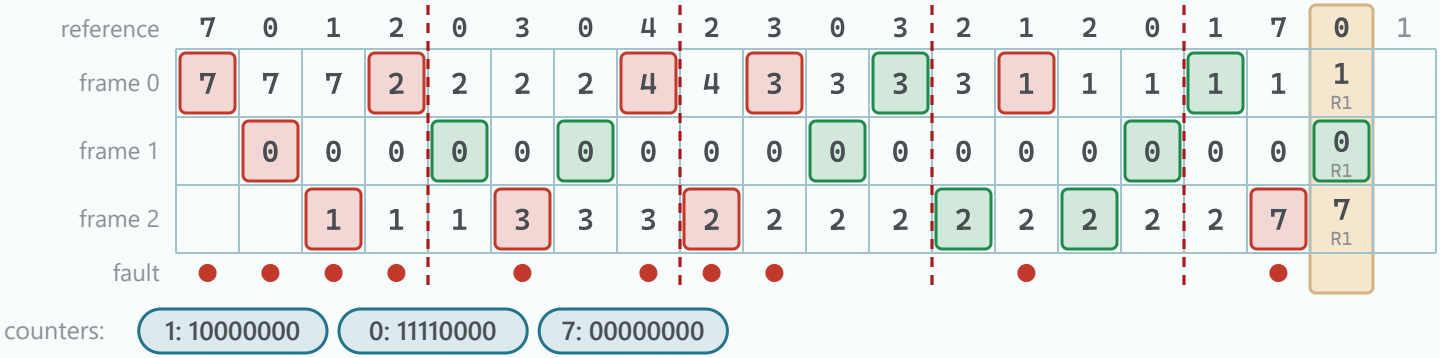


18. reference 7: fault; R and counter together (R as the top bit): 1: 110000000, 0: 011110000, 2: 011000000 → the smallest, **2**, leaves. Faults: 10

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

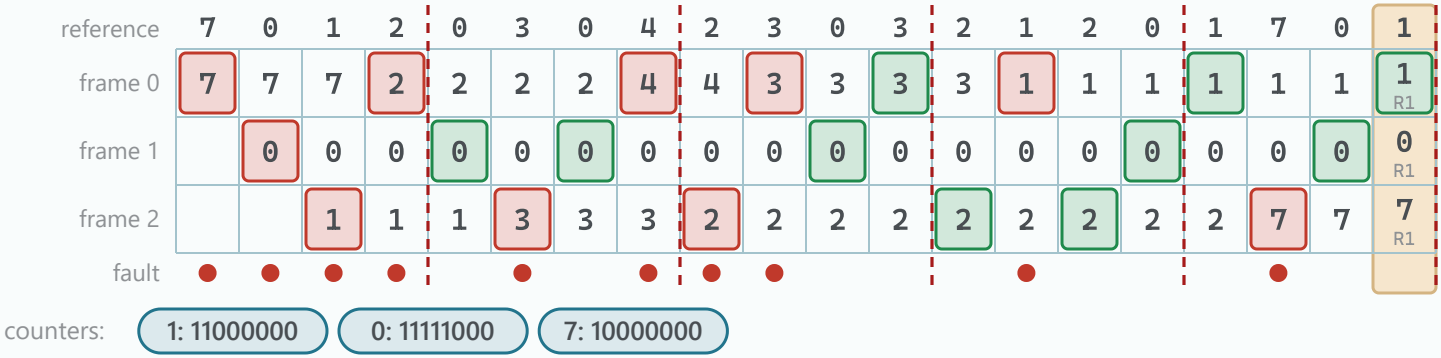


19. reference 0: hit; R ← 1.

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

Aging on the Common String

The same string, 3 frames, a clock tick every 4 references. A newly loaded page has counter 0 and R = 1; in comparisons **this interval's R bit counts as the most significant bit** (otherwise a new page would be evicted at once).

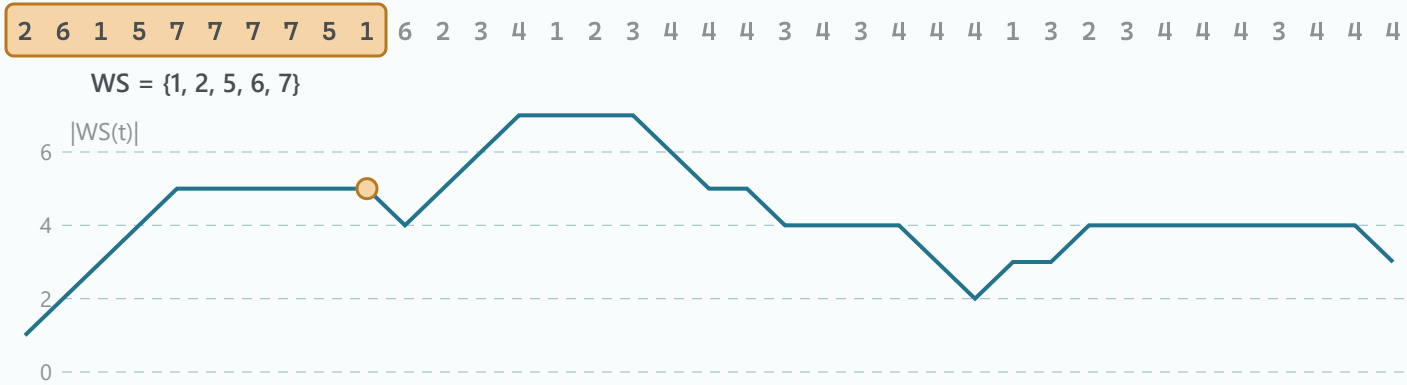


20. reference 1: hit; R ← 1. Tick: counters shifted, R bits added on the left. **Total: 10 faults.**

Dashed line: a clock tick. A single short string can't rank the algorithms; the tick rate and tie rule change the outcome.

The Working-Set Model

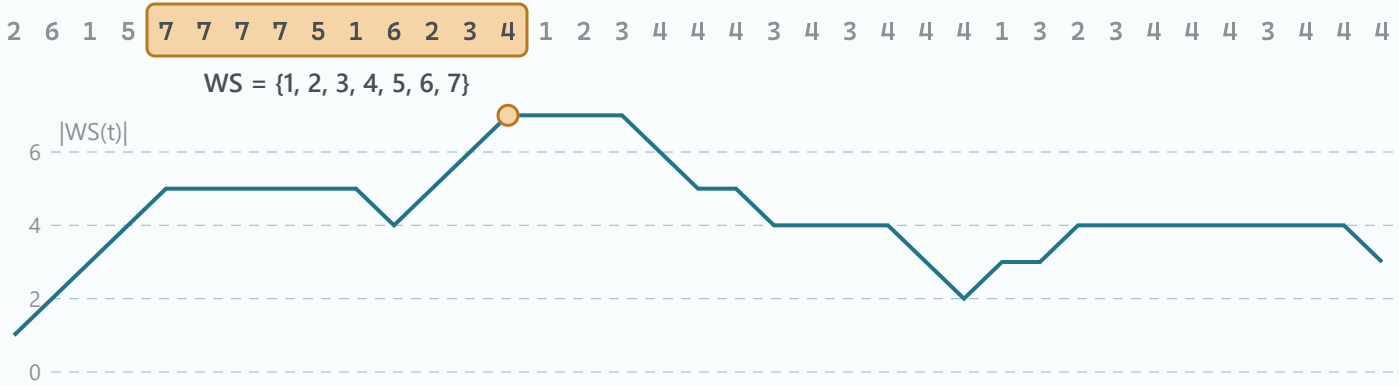
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 10: window = references 1–10; working set {1, 2, 5, 6, 7}, size **5**. (t_1 in Silberschatz)

The Working-Set Model

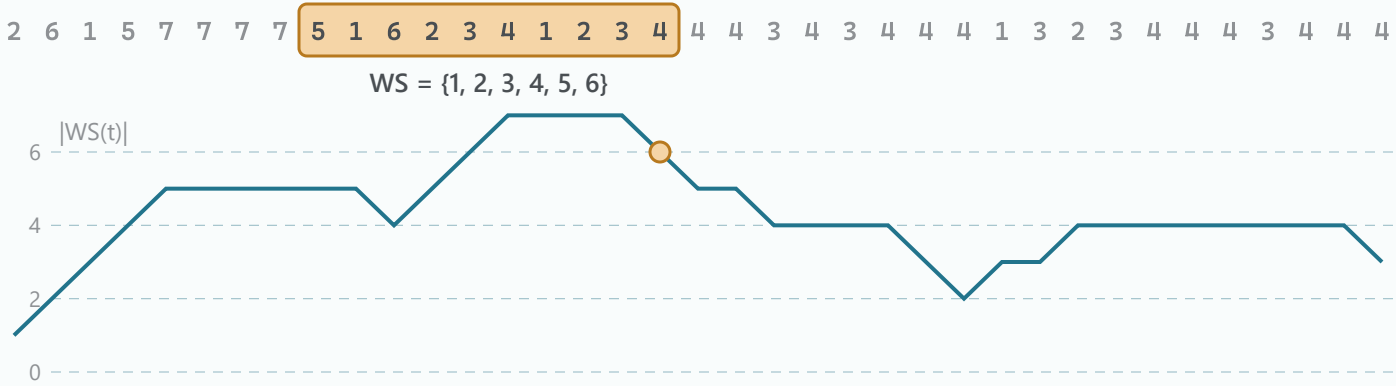
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 14: window = references 5–14; working set {1, 2, 3, 4, 5, 6, 7}, size **7**.

The Working-Set Model

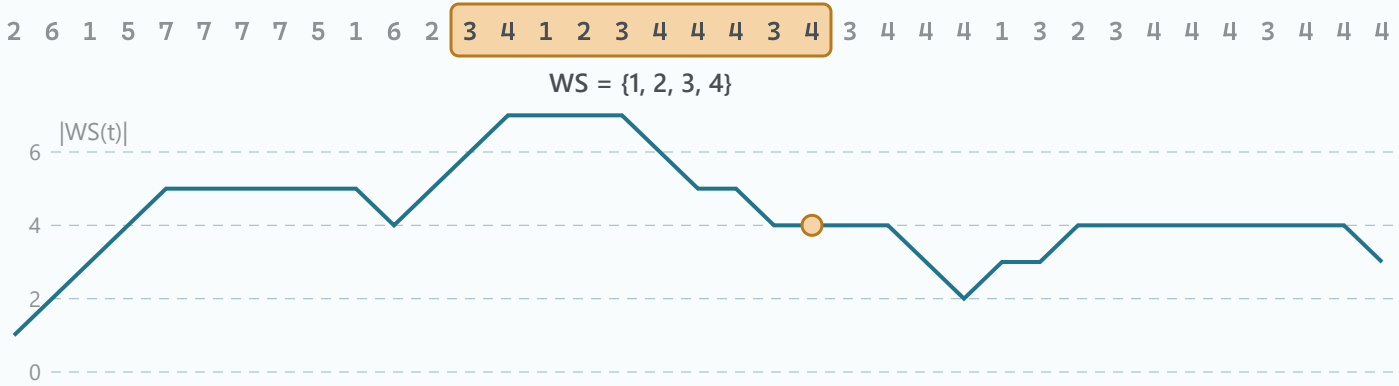
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 18: window = references 9–18; working set {1, 2, 3, 4, 5, 6}, size **6**.

The Working-Set Model

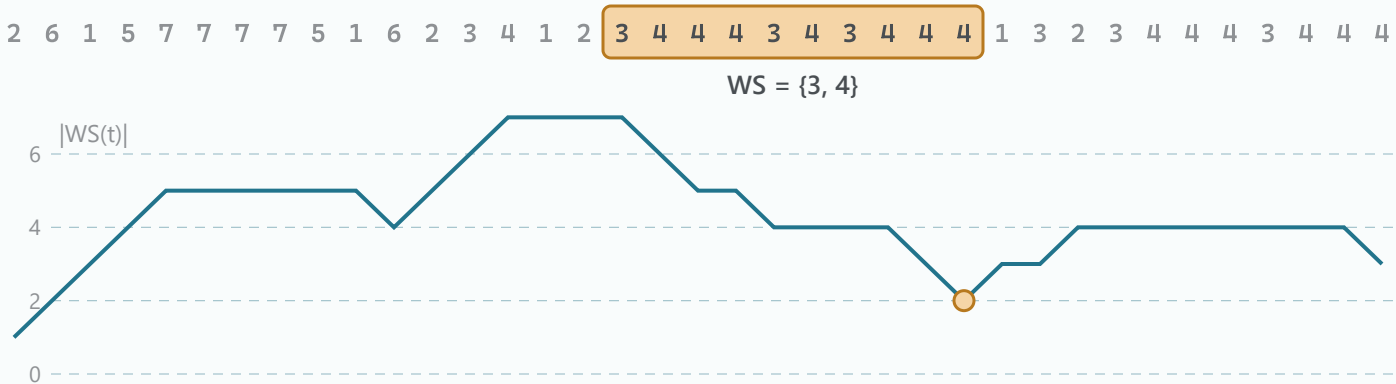
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 22: window = references 13–22; working set {1, 2, 3, 4}, size **4**.

The Working-Set Model

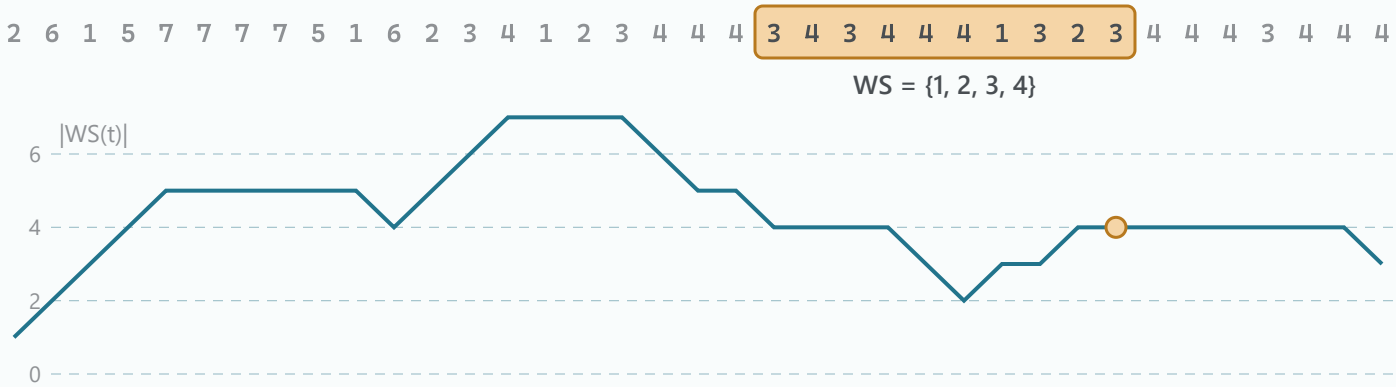
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 26: window = references 17–26; working set {3, 4}, size **2**. (t_2): the process has moved to a new **locality**.

The Working-Set Model

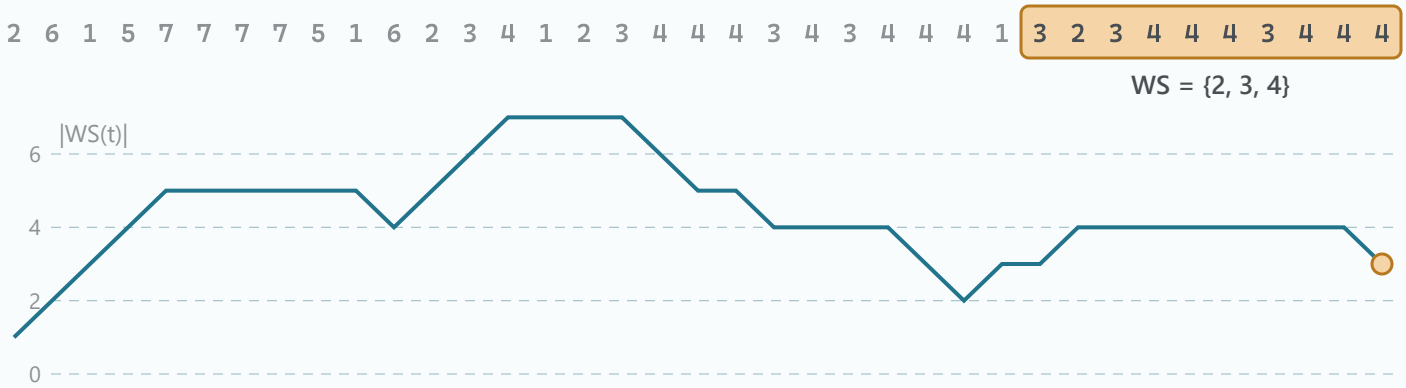
- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 30: window = references 21–30; working set {1, 2, 3, 4}, size **4**.

The Working-Set Model

- **Locality:** a process uses a small group of pages for a while, then moves to another group.
- The **working set** WS(t, Δ): the set of pages used in the most recent Δ *memory references* (Denning, 1968).
- If Δ is too small it misses the locality; if too large it overlaps several localities; Δ = ∞ is the whole program.
- If the working set is in memory, faults are few; if not, **thrashing**.



t = 37: window = references 28–37; working set {2, 3, 4}, size **3**.

The Working-Set Algorithm

- The exact definition (last Δ references) needs a register shifted on every access: far too expensive. The approximation: the pages used in the **last τ of virtual time**.

time of last use	R
2084	1
2003	1
1980	1
1213	0
2014	1
2020	1
2032	1
1620	0

2204 current virtual time

scan all pages:

R = 1 → time ← now,
R ← 0

R = 0, age > τ → evict

R = 0, age ≤ τ → remember the oldest

if none: the oldest (preferably clean) page

Each entry has its **time of last use** and an R bit. Time is **virtual** time: the process's own CPU time (measured in ticks). Let τ = 500.

Tanenbaum MOS Figure 3-20 (redrawn). The drawback: scanning the *whole* table on every fault.

The Working-Set Algorithm

- The exact definition (last Δ references) needs a register shifted on every access: far too expensive. The approximation: the pages used in the **last τ of virtual time**.

time of last use	R
2204	0
2204	0
2204	0
1213	0
2204	0
2204	0
2204	0
1620	0

2204 current virtual time

scan all pages:

R = 1 → time ← now,
R ← 0

R = 0, age > τ → evict

R = 0, age ≤ τ → remember the oldest

if none: the oldest (preferably clean) page

On a page fault the table is scanned: pages with R = 1 were used in this tick → their time is set to **2204** and R cleared.

Tanenbaum MOS Figure 3-20 (redrawn). The drawback: scanning the *whole* table on every fault.

The Working-Set Algorithm

- The exact definition (last Δ references) needs a register shifted on every access: far too expensive. The approximation: the pages used in the **last τ of virtual time**.

time of last use	R	
2204	0	
2204	0	
2204	0	
1213	0	age 991 > τ
2204	0	
2204	0	
2204	0	
1620	0	age 584 > τ

2204 current virtual time

scan all pages:

R = 1 → time ← now,
R ← 0

R = 0, age > τ → evict

R = 0, age ≤ τ → remember the oldest

if none: the oldest (preferably clean) page

For pages with R = 0 the age is computed: 2204 − 1213 = 991 and 2204 − 1620 = 584; both exceed τ → they are **not** in the working set.

Tanenbaum MOS Figure 3-20 (redrawn). The drawback: scanning the *whole* table on every fault.

The Working-Set Algorithm

- The exact definition (last Δ references) needs a register shifted on every access: far too expensive. The approximation: the pages used in the **last τ of virtual time**.

time of last use	R	
2204	0	
2204	0	
2204	0	
1213	0	age 991 > τ → evict
2204	0	
2204	0	
2204	0	
1620	0	age 584 > τ

2204 current virtual time

scan all pages:

R = 1 → time ← now,
R ← 0

R = 0, age > τ → evict

R = 0, age ≤ τ → remember the oldest

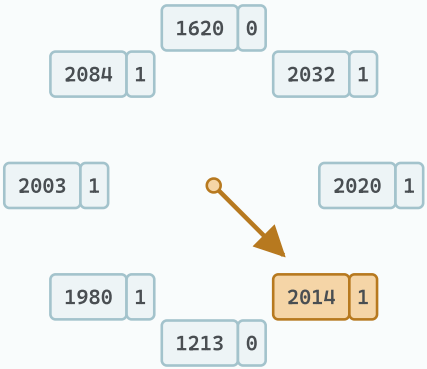
if none: the oldest (preferably clean) page

The first suitable page found (1213) is evicted; the scan continues, updating the other entries. If all are in the set, the oldest R = 0 page is chosen, or failing that a random (preferably clean) page.

Tanenbaum MOS Figure 3-20 (redrawn). The drawback: scanning the *whole* table on every fault.

WSClock

- The working-set idea + the clock's **circular list** (Carr and Hennessy, 1981): no need to scan the whole table.
- An old dirty page isn't evicted at once: its **write is scheduled** and the hand looks for a clean page.
- If the hand goes all the way around: if at least one write was scheduled, it keeps going until it finds the first page cleaned.
- If no writes were scheduled, all pages are in the working set: any clean page; failing that, the page under the hand is written and evicted.



when the hand reaches a page:

R = 1 → R ← 0, time ← now, advance

R = 0, age > τ, clean → evict

R = 0, age > τ, dirty → schedule the write, advance

R = 0, age ≤ τ → advance

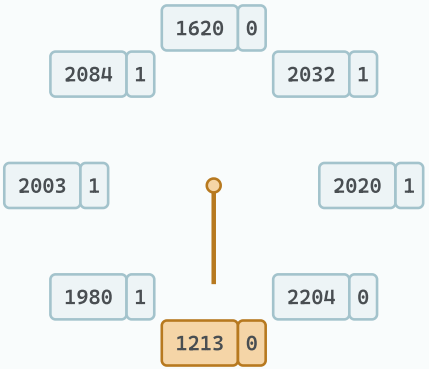
τ = 500, now = 2204

Page fault: the hand is at the page with time 2014; R = 1 → used in this tick, not a good victim.

Tanenbaum MOS Figure 3-21 (redrawn). A simple, efficient design, widely used in practice.

WSClock

- The working-set idea + the clock's **circular list** (Carr and Hennessy, 1981): no need to scan the whole table.
- An old dirty page isn't evicted at once: its **write is scheduled** and the hand looks for a clean page.
- If the hand goes all the way around: if at least one write was scheduled, it keeps going until it finds the first page cleaned.
- If no writes were scheduled, all pages are in the working set: any clean page; failing that, the page under the hand is written and evicted.



when the hand reaches a page:

R = 1 → R ← 0, time ← now, advance

R = 0, age > τ, clean → evict

R = 0, age > τ, dirty → schedule the write, advance

R = 0, age ≤ τ → advance

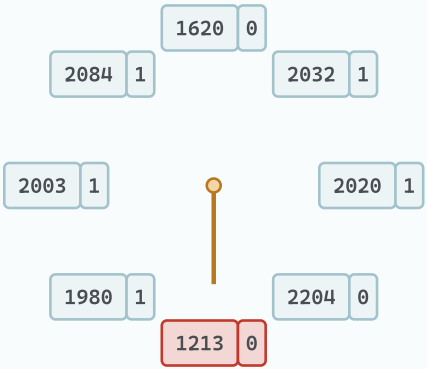
τ = 500, now = 2204

R ← 0, its time becomes 2204, and the hand advances. Next page: time 1213, R = 0.

Tanenbaum MOS Figure 3-21 (redrawn). A simple, efficient design, widely used in practice.

WSClock

- The working-set idea + the clock's **circular list** (Carr and Hennessy, 1981): no need to scan the whole table.
- An old dirty page isn't evicted at once: its **write is scheduled** and the hand looks for a clean page.
- If the hand goes all the way around: if at least one write was scheduled, it keeps going until it finds the first page cleaned.
- If no writes were scheduled, all pages are in the working set: any clean page; failing that, the page under the hand is written and evicted.



when the hand reaches a page:

R = 1 → R ← 0, time ← now, advance

R = 0, age > τ, clean → evict

R = 0, age > τ, dirty → schedule the write, advance

R = 0, age ≤ τ → advance

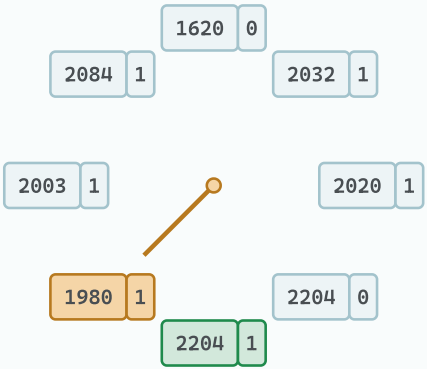
τ = 500, now = 2204

Age = 2204 − 1213 = 991 > τ: not in the working set. If clean, it's the **victim**; if dirty, its write would be scheduled and the hand would move on.

Tanenbaum MOS Figure 3-21 (redrawn). A simple, efficient design, widely used in practice.

WSClock

- The working-set idea + the clock's **circular list** (Carr and Hennessy, 1981): no need to scan the whole table.
- An old dirty page isn't evicted at once: its **write is scheduled** and the hand looks for a clean page.
- If the hand goes all the way around: if at least one write was scheduled, it keeps going until it finds the first page cleaned.
- If no writes were scheduled, all pages are in the working set: any clean page; failing that, the page under the hand is written and evicted.



when the hand reaches a page:

R = 1 → R ← 0, time ← now, advance

R = 0, age > τ, clean → evict

R = 0, age > τ, dirty → schedule the write, advance

R = 0, age ≤ τ → advance

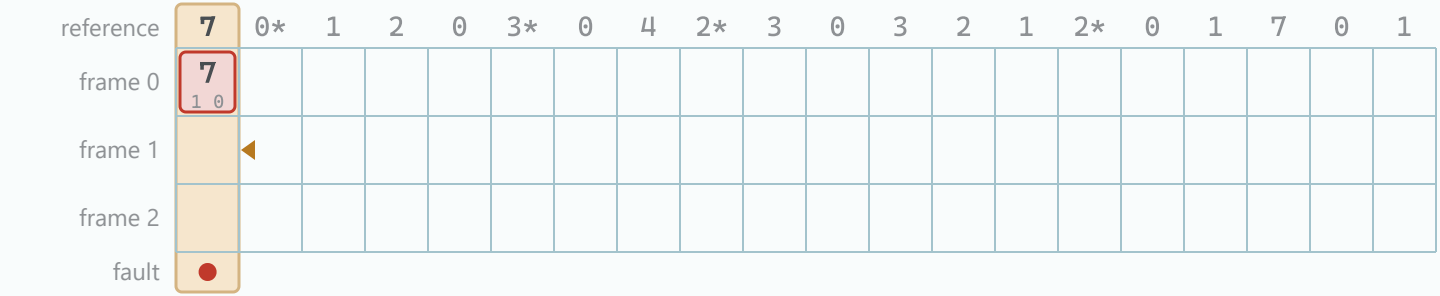
τ = 500, now = 2204

The new page is loaded into this frame: time 2204, R = 1. The hand moves to the next frame.

Tanenbaum MOS Figure 3-21 (redrawn). A simple, efficient design, widely used in practice.

WSClock on the Common String

Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.

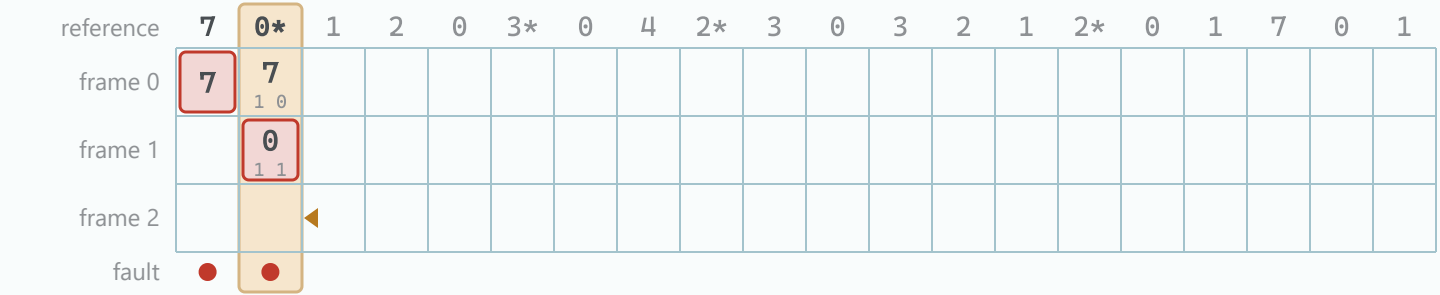


last use (virtual time t = 1): 7: 1

1. reference 7: fault; loaded into free frame 0. Faults: 1

WSClock on the Common String

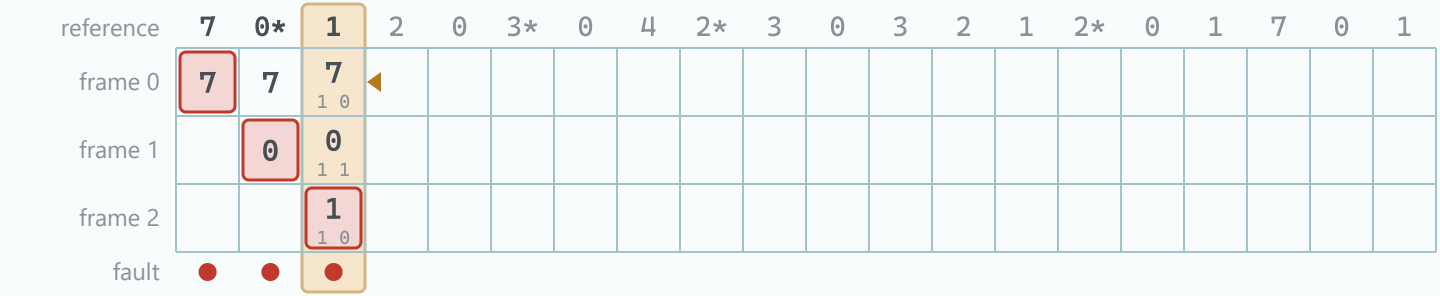
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



2. reference 0*: fault; loaded into free frame 1. Faults: 2

WSClock on the Common String

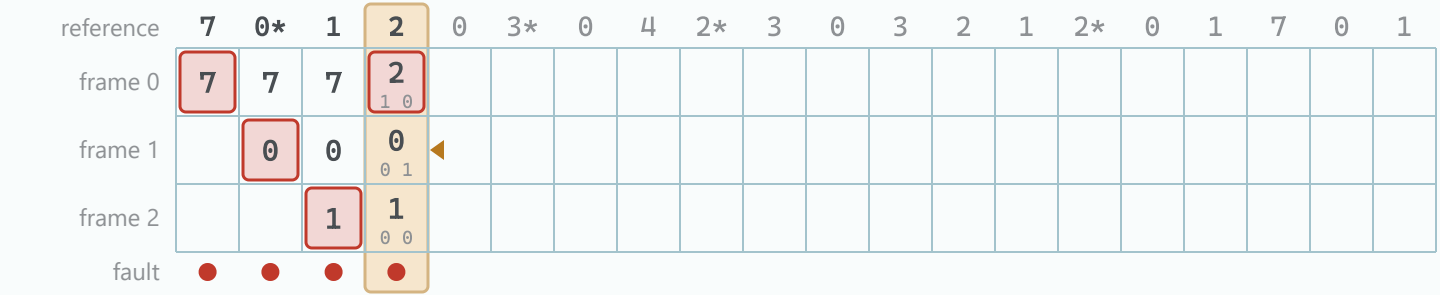
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



3. reference 1: fault; loaded into free frame 2. Faults: 3

WSClock on the Common String

Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.

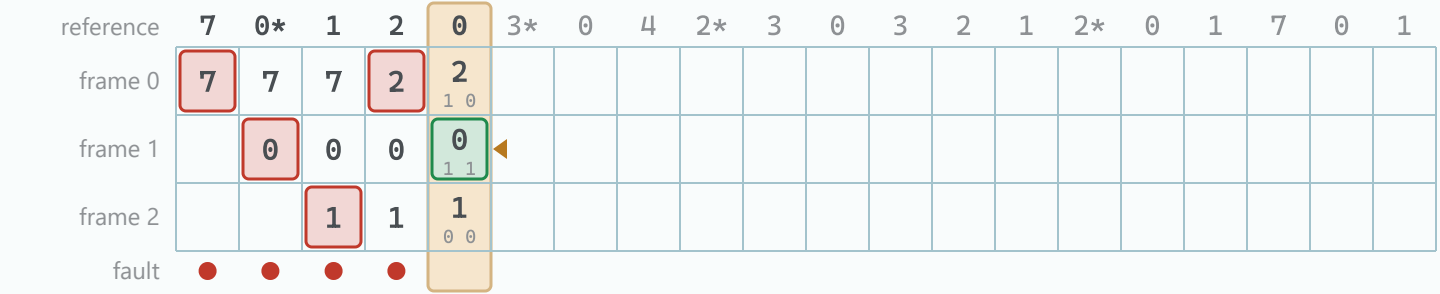


last use (virtual time t = 4): 2: 4 | 0: 4 | 1: 4

4. reference 2: fault; hand scan: 7: R=1 → R=0, time=4; 0: R=1 → R=0, time=4; 1: R=1 → R=0, time=4; pass over, no writes: first clean page 7 → victim. Faults: 4

WSClock on the Common String

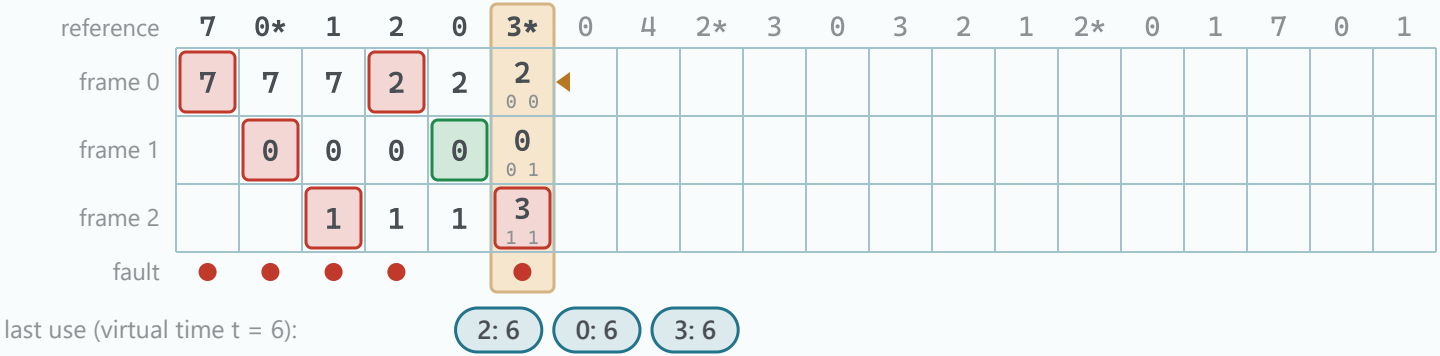
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



5. reference 0: hit; R ← 1.

WSClock on the Common String

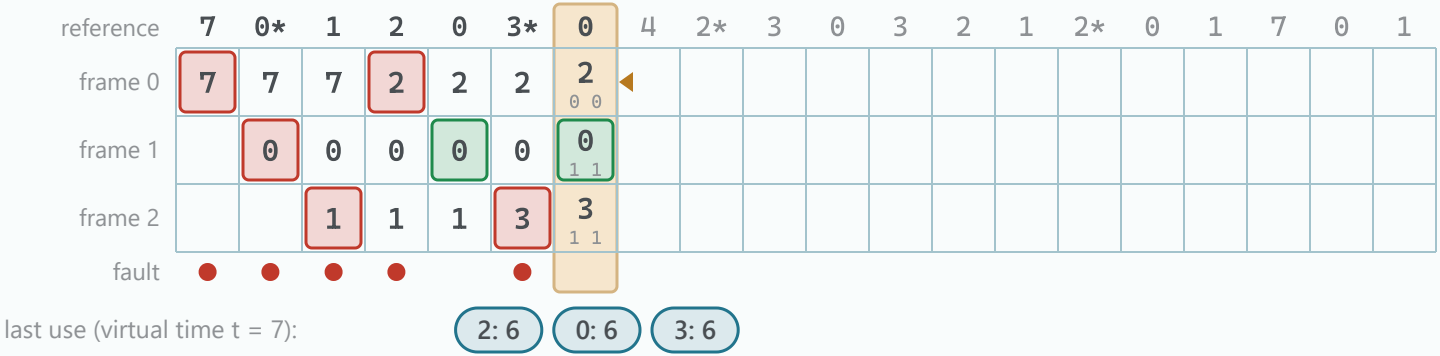
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



6. reference 3*: fault; hand scan: 0: R=1 → R=0, time=6; 1: R=0, age 2 ≤ τ → in working set; 2: R=1 → R=0, time=6; pass over, no writes: first clean page 1 → victim. Faults: 5

WSClock on the Common String

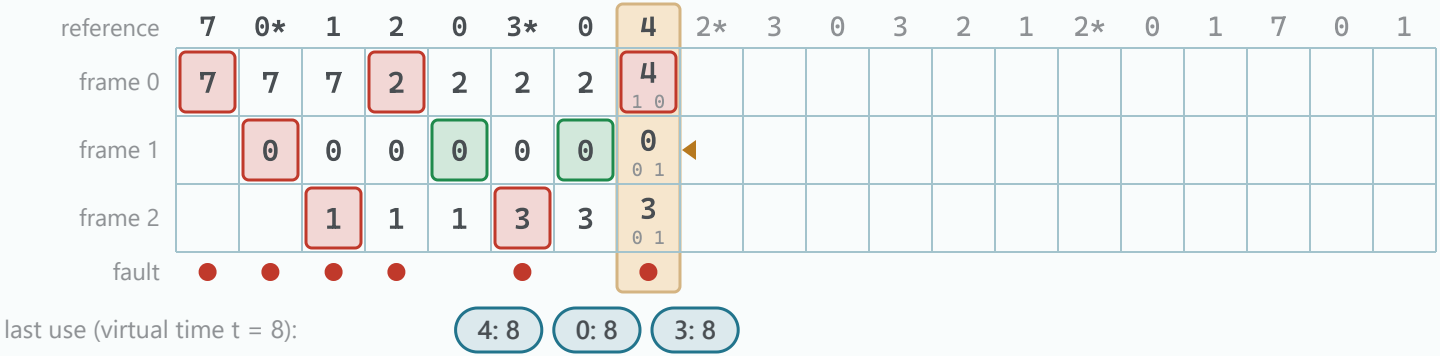
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



7. reference 0: hit; R ← 1.

WSClock on the Common String

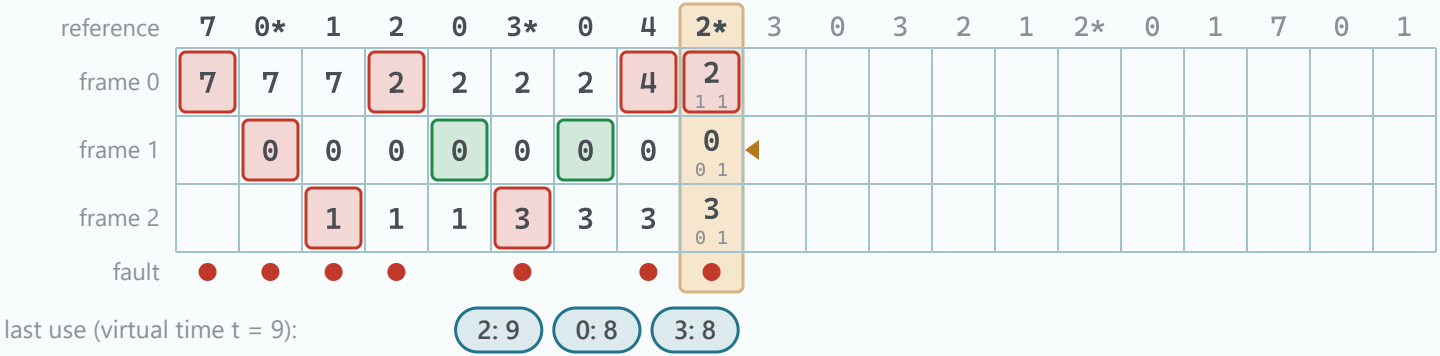
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



8. reference 4: fault; hand scan: 2: R=0, age 2 ≤ τ → in working set; 0: R=1 → R=0, time=8; 3: R=1 → R=0, time=8; pass over, no writes: first clean page 2 → victim. Faults: 6

WSClock on the Common String

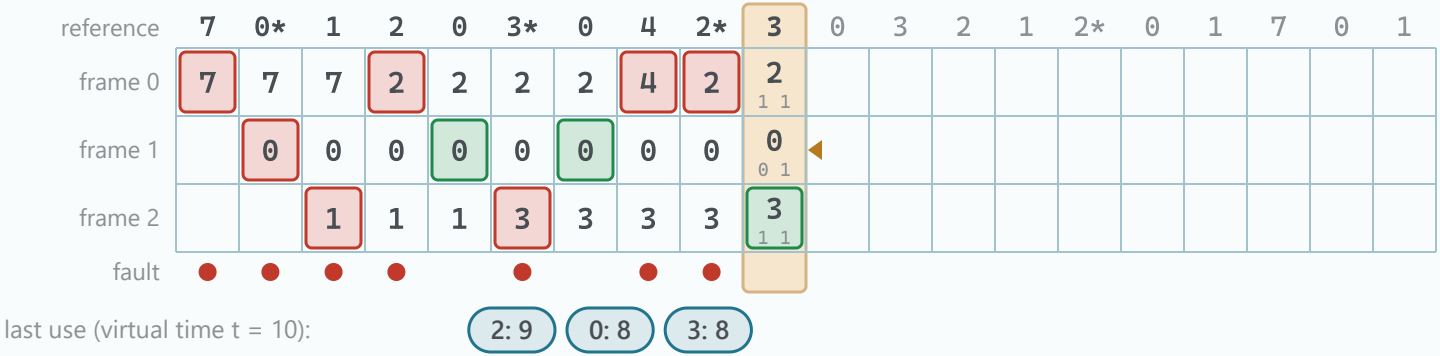
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



9. reference 2*: fault; hand scan: 0: R=0, age 1 ≤ τ → in working set; 3: R=0, age 1 ≤ τ → in working set; 4: R=1 → R=0, time=9; pass over, no writes: first clean page 4 → victim. Faults: 7

WSClock on the Common String

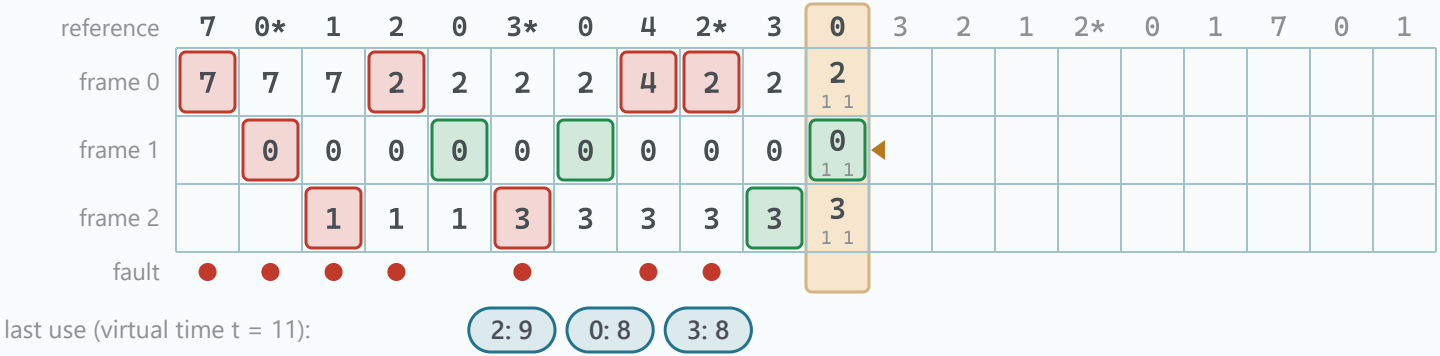
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



10. reference 3: hit; R ← 1.

WSClock on the Common String

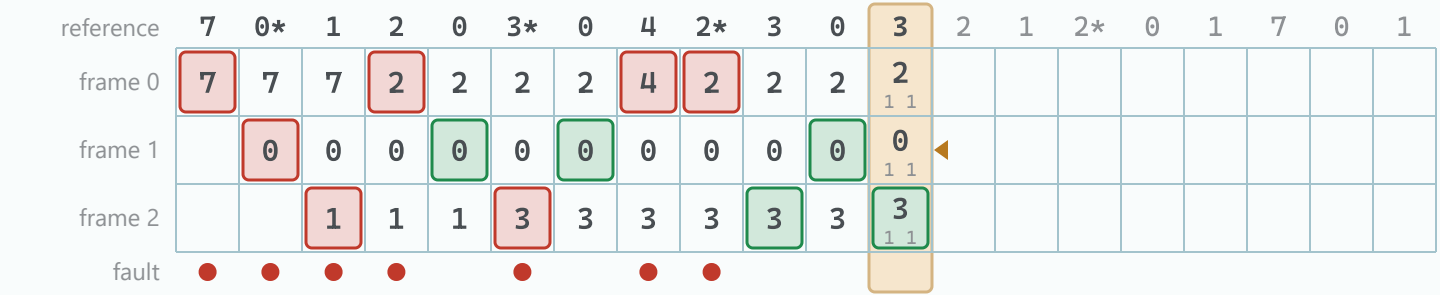
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



11. reference 0: hit; R ← 1.

WSClock on the Common String

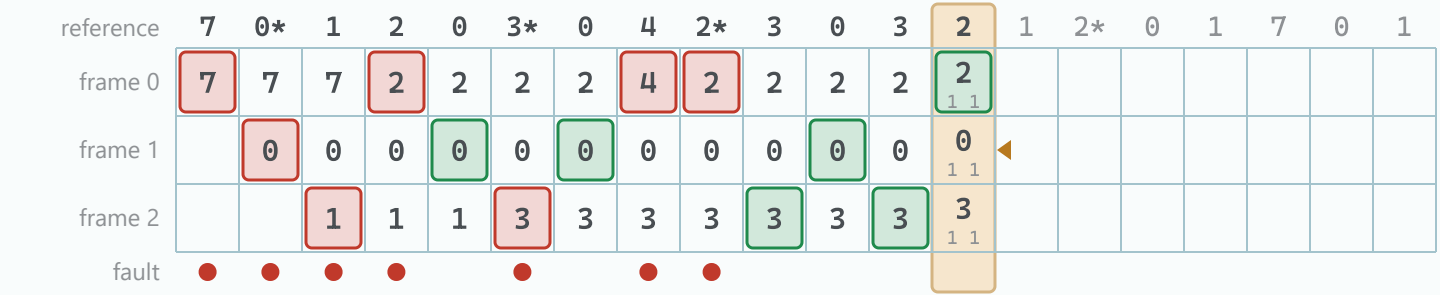
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



12. reference 3: hit; R ← 1.

WSClock on the Common String

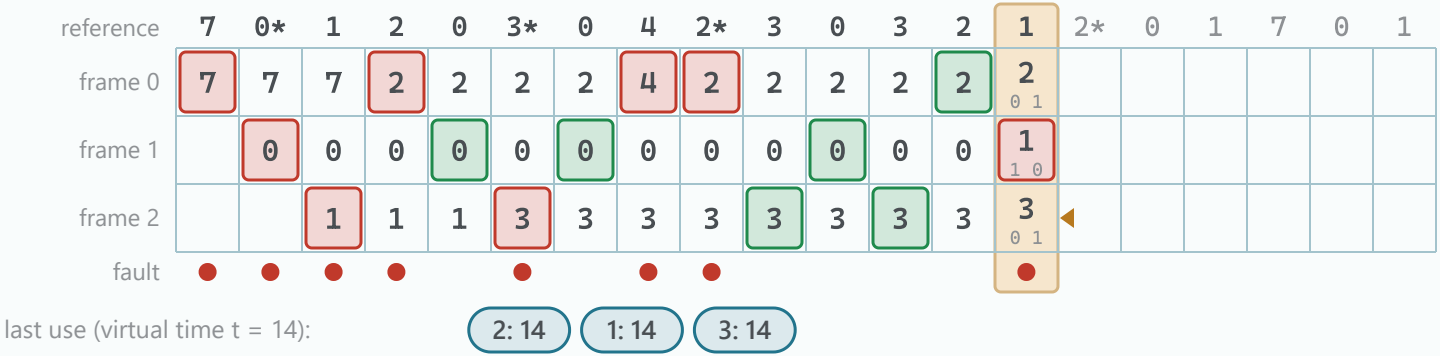
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



13. reference 2: hit; R ← 1.

WSClock on the Common String

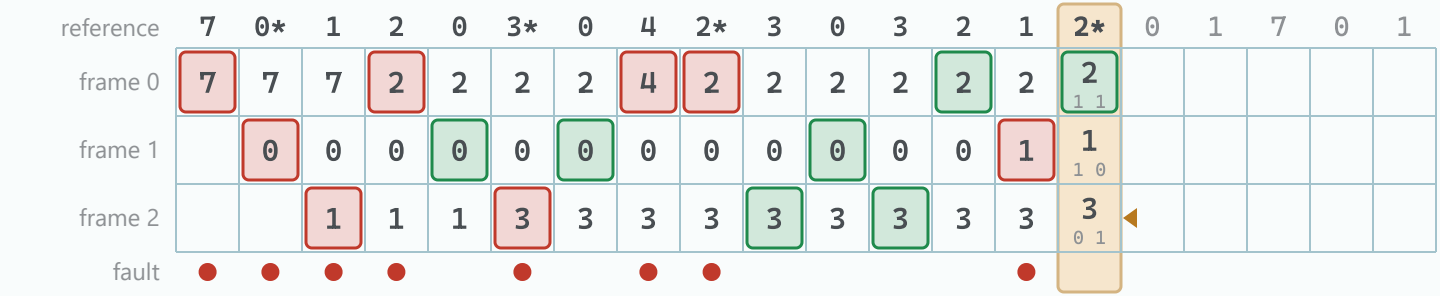
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



14. reference 1: fault; hand scan: 0: R=1 → R=0, time=14; 3: R=1 → R=0, time=14; 2: R=1 → R=0, time=14; pass over, nothing clean: 0 is written and evicted. Faults: 8

WSClock on the Common String

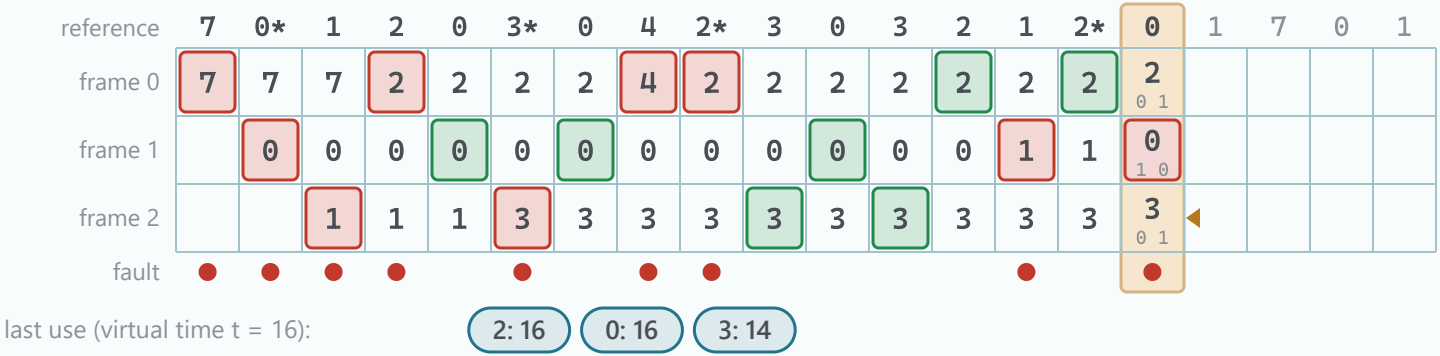
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



15. reference 2*: hit; R ← 1, write → M ← 1.

WSClock on the Common String

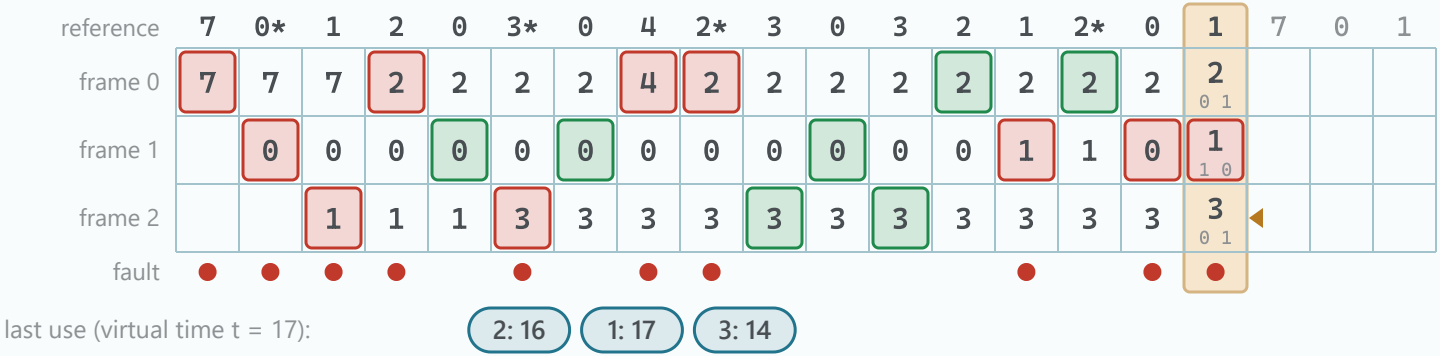
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



16. reference 0: fault; hand scan: 3: R=0, age 2 ≤ τ → in working set; 2: R=1 → R=0, time=16; 1: R=1 → R=0, time=16; pass over, no writes: first clean page 1 → victim. Faults: 9

WSClock on the Common String

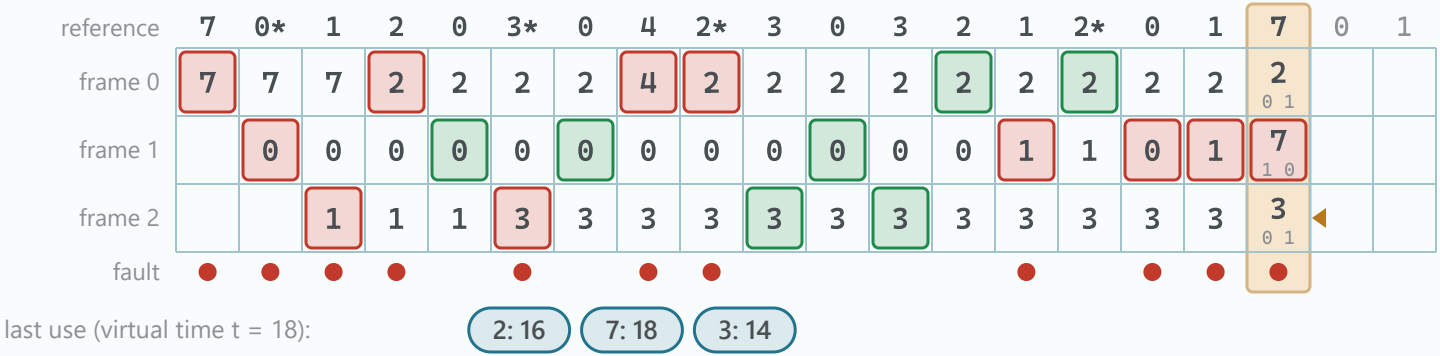
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



17. reference 1: fault; hand scan: 3: R=0, age 3 ≤ τ → in working set; 2: R=0, age 1 ≤ τ → in working set; 0: R=1 → R=0, time=17; pass over, no writes: first clean page 0 → victim. Faults: 10

WSClock on the Common String

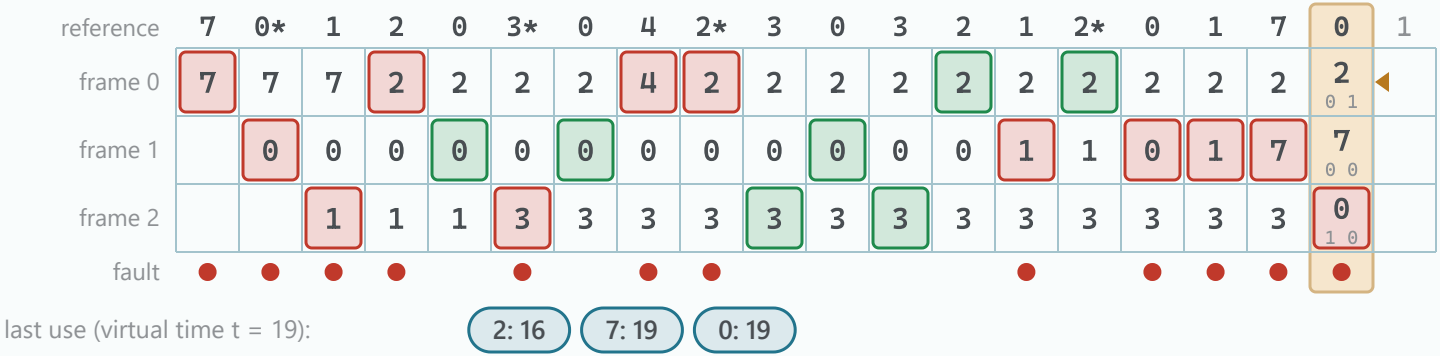
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



18. reference 7: fault; hand scan: 3: R=0, age 4 ≤ τ → in working set; 2: R=0, age 2 ≤ τ → in working set; 1: R=1 → R=0, time=18; pass over, no writes: first clean page 1 → victim. Faults: 11

WSClock on the Common String

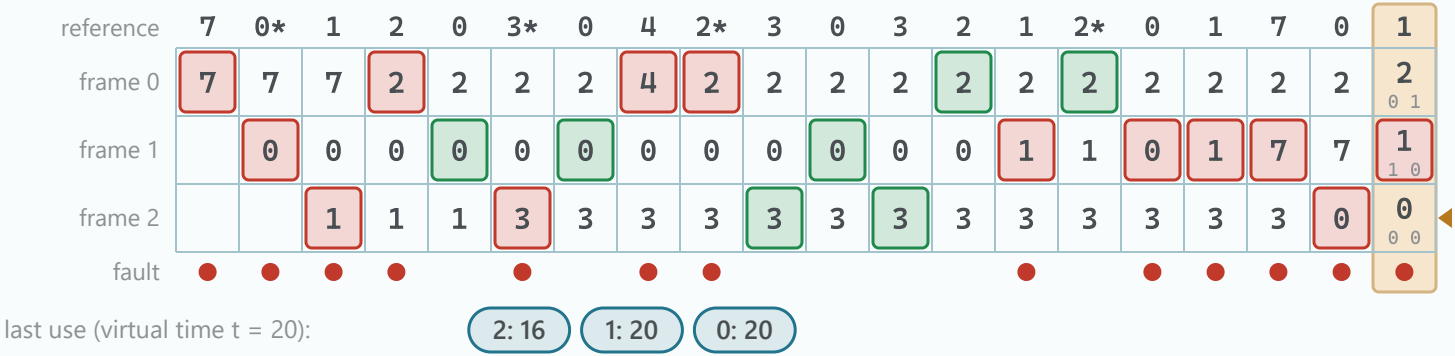
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



19. reference 0: fault; hand scan: 3: R=0, age 5 > τ, dirty → write scheduled; 2: R=0, age 3 ≤ τ → in working set; 7: R=1 → R=0, time=19; pass over: 3, now written, is clean → victim. Faults: 12

WSClock on the Common String

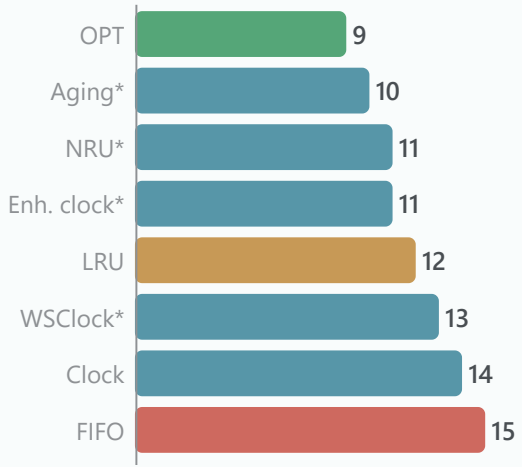
Virtual time = reference number, τ = 4. The string and the writes (*) are the same as for NRU. A page whose write was scheduled counts as cleaned by the end of the lap.



20. reference 1: fault; hand scan: 2: R=0, age 4 ≤ τ → in working set; 7: R=0, age 1 ≤ τ → in working set; 0: R=1 → R=0, time=20; pass over, no writes: first clean page 7 → victim. Faults: 13 **Total: 13 faults, 2 dirty writes.** With only 3 frames and a short string, the working-set idea gives no advantage.

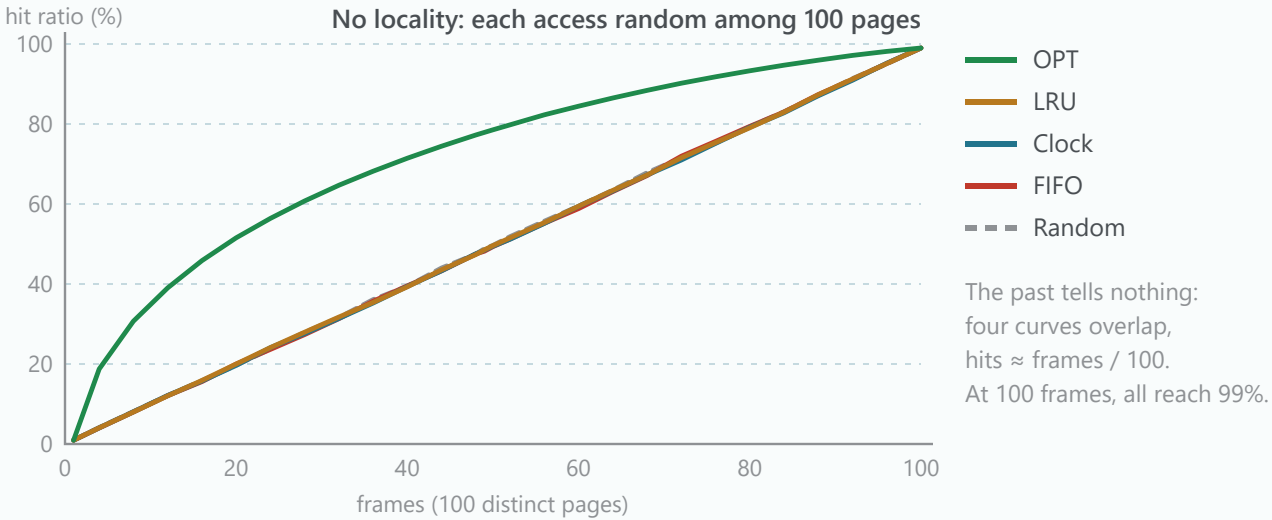
Comparing the Algorithms

Algorithm	Comment
OPT	Not implementable; a lower bound for comparison.
NRU	A very crude approximation of LRU; cheap.
FIFO	May evict important pages; Belady's anomaly.
Second chance	A big improvement over FIFO.
Clock	A realistic implementation of second chance.
LRU	Excellent, but the exact version needs hardware.
NFU	A fairly crude approximation of LRU (never forgets).
Aging	An efficient algorithm that approximates LRU well.
Working set	Good, but expensive to implement.
WSClock	Good and efficient.



For other reference strings, see Tanenbaum MOS §3.4. * Algorithms using R/M bits also use write and tick information; one short string gives no general ranking. NFU never forgets, so it doesn't approximate LRU well; aging does (Tanenbaum).

The Workload Decides the Policy



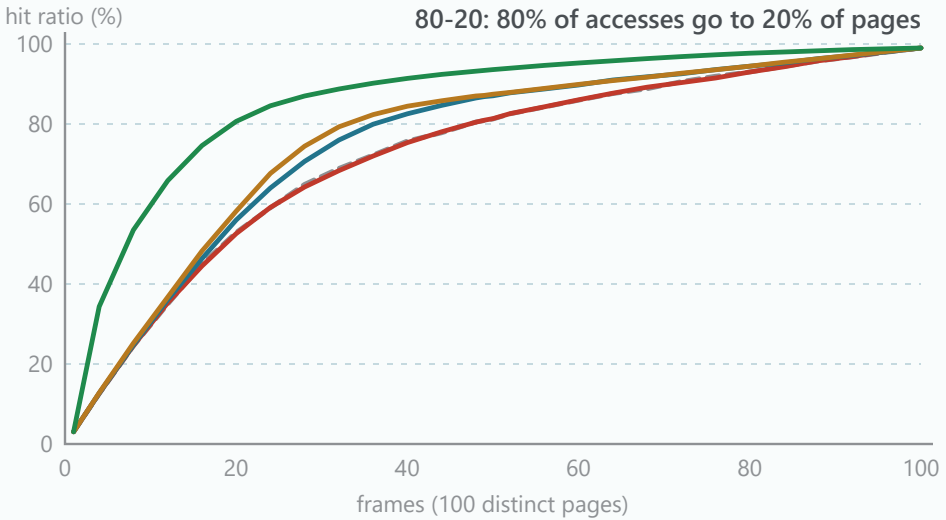
The past tells nothing:
four curves overlap,
hits ≈ frames / 100.
At 100 frames, all reach 99%.

Without locality it doesn't matter which pages are kept: LRU, FIFO, clock, and random trace the same line. Only OPT, which knows the future, is ahead; once memory holds every page, policy is irrelevant.

Step 1 / 3

A 10,000-access simulation (random: average of 5 trials). Sequential scans hit LRU in real systems too: reading a large file once can evict hot pages. That's why Linux first puts a new page on the inactive list and promotes it to the active list only on a second access.

The Workload Decides the Policy

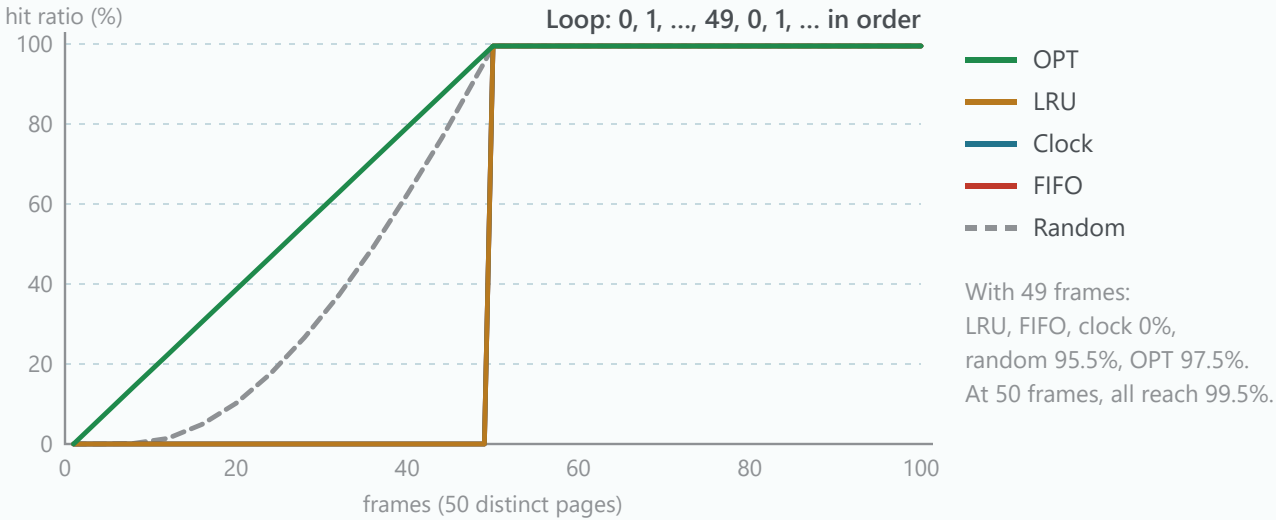


With 20 frames:
OPT 81%, LRU 58%,
clock 56%,
FIFO and random 53%.

With hot pages, policies that use history win: LRU keeps the 20 hot pages in memory. Clock comes very close to LRU; FIFO is no better than random.

A 10,000-access simulation (random: average of 5 trials). Sequential scans hit LRU in real systems too: reading a large file once can evict hot pages. That's why Linux first puts a new page on the inactive list and promotes it to the active list only on a second access.

The Workload Decides the Policy

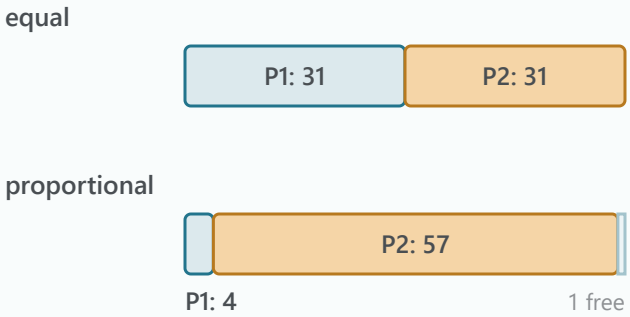


A loop slightly larger than memory is LRU's worst case: the evicted page is always the one needed soonest, so every access misses. Blind random choice crushes LRU here.

A 10,000-access simulation (random: average of 5 trials). Sequential scans hit LRU in real systems too: reading a large file once can evict hot pages. That's why Linux first puts a new page on the inactive list and promotes it to the active list only on a second access.

Frame Allocation: Equal and Proportional

- **Lower bound:** all pages an instruction can touch must be in memory at once (instruction + operands + indirect addresses may cross page boundaries). The *architecture* sets this.
- **Upper bound:** free physical memory.
- **Equal:** m frames, n processes → ⌊m / n⌋ each.
- **Proportional:** $a_i = s_i / S \times m$. Example: 10/137 × 62 ≈ **4**, 127/137 × 62 ≈ **57**.
- **Priority:** the ratio is based on priority instead of (or as well as) size; a high-priority process may take frames from a low-priority one.
- On NUMA, *where* a frame is matters too: a process should be fed from memory close to the node it runs on.

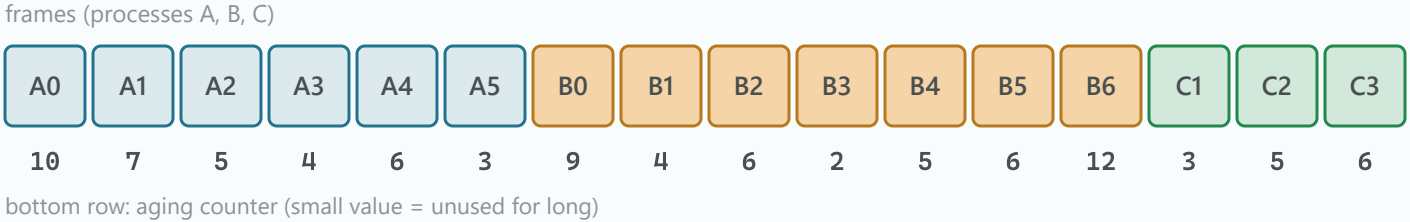


s_1 = 10 pages, s_2 = 127 pages, m = 62

$a_i = s_i / S \times m$ (rounded down)

Local and Global Replacement

- **Global:** the victim is chosen among all frames; a process can take a frame from another. Higher throughput, but a process's run time can vary widely with others' behavior.
- Linux and Windows use global reclaim (bounded by cgroup/job-object limits).
- **Local:** a process chooses only among its own frames; more consistent performance, but idle frames can't pass to others.
- A local policy needs a mechanism that adjusts the frame count as the working set grows and shrinks (PFF).

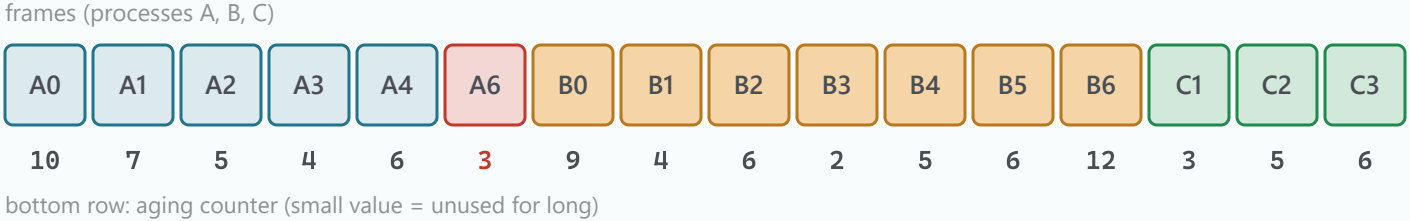


Three processes share 16 frames; the numbers are aging counter values (small = unused for a long time). A faults on page A6.

Tanenbaum MOS Figure 3-22 (redrawn).

Local and Global Replacement

- **Global:** the victim is chosen among all frames; a process can take a frame from another. Higher throughput, but a process's run time can vary widely with others' behavior.
- Linux and Windows use global reclaim (bounded by cgroup/job-object limits).
- **Local:** a process chooses only among its own frames; more consistent performance, but idle frames can't pass to others.
- A local policy needs a mechanism that adjusts the frame count as the working set grows and shrinks (PFF).

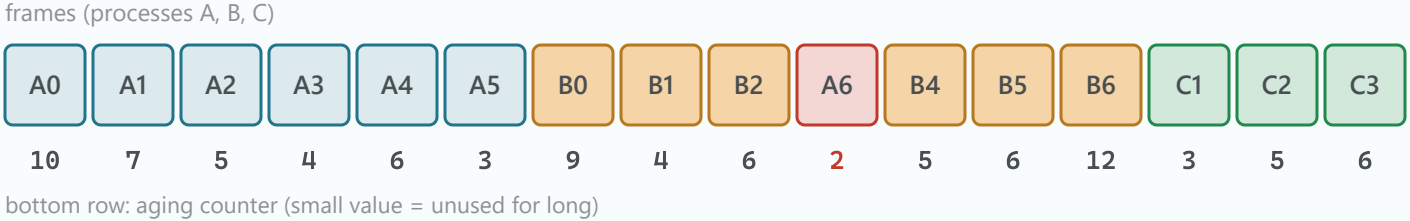


Local: the victim comes only from A's frames → **A5**, with the smallest value. A's frame count stays the same (6).

Tanenbaum MOS Figure 3-22 (redrawn).

Local and Global Replacement

- **Global:** the victim is chosen among all frames; a process can take a frame from another. Higher throughput, but a process's run time can vary widely with others' behavior.
- Linux and Windows use global reclaim (bounded by cgroup/job-object limits).
- **Local:** a process chooses only among its own frames; more consistent performance, but idle frames can't pass to others.
- A local policy needs a mechanism that adjusts the frame count as the working set grows and shrinks (PFF).

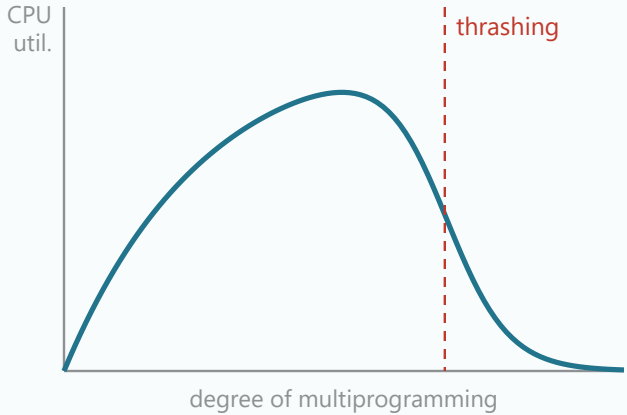


Global: every frame is a candidate → **B3**, with the smallest value, leaves. A grows, B shrinks: allocation changes dynamically.

Tanenbaum MOS Figure 3-22 (redrawn).

Thrashing

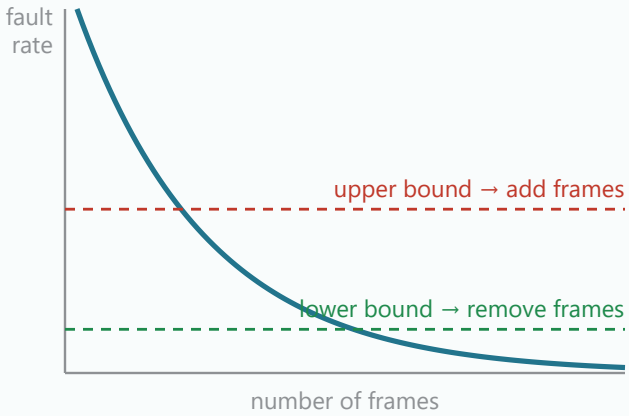
- If processes' active pages (their working sets) don't fit in memory, the page-fault rate climbs: an evicted page is **needed again right away**.
- Processes spend most of their time waiting for pages; the disk queue fills and **CPU utilization drops**.
- Old long-term schedulers read this as "CPU idle" and admitted **more processes** → things got worse (a feedback loop).
- The cause: $\Sigma\ |WS_i| >$ memory. Local replacement *confines* thrashing to one process but doesn't cure it.
- Remedies: lower the degree of multiprogramming (suspend/swap out processes), more RAM, or the system's OOM killer.



Symptoms on Linux: high `si/so` and `wa` in `vmstat 1`; `/proc/pressure/memory` (PSI) reports memory pressure as a percentage.

Preventing Thrashing: Working Set and PFF

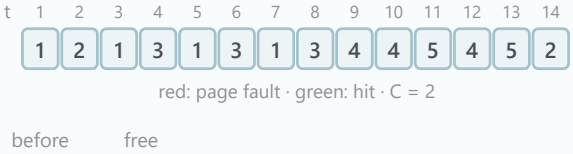
- **Working-set strategy:** $D = \Sigma |WS_i|$ is the total demand. If D > m, a process is **suspended**; if D is small enough, a new process is started.
- For a process returning from suspension, loading its working set first (prepaging) avoids a fault storm.
- **PFF** (page-fault frequency): more direct feedback. If the fault rate exceeds an upper bound the process gets more frames; below a lower bound, frames are taken away.
- If the rate is high and there are no free frames, a process is suspended and its frames redistributed.



Page faults and localities are linked: moving to a new locality causes a **peak** in the fault rate, which drops once the working set is loaded.

PFF: Code Trace

```
▶ void pff(Proc *s, int p, int t) {
▶   if (t - s->last > C) {  /* rare faults */
3     for (int i = s->n - 1; i >= 0; i--)
4       if (s->set[i].R == 0)
5         evict(s, i);    /* unused */
6   }
7   for (int i = 0; i < s->n; i++)
8     s->set[i].R = 0;      /* clear */
9   add(s, p);              /* with R = 1 */
10  s->last = t;
11 }
```



The process starts with no memory; the threshold is C = 2 references. `pff` is called on every fault and checks the time since the last fault.

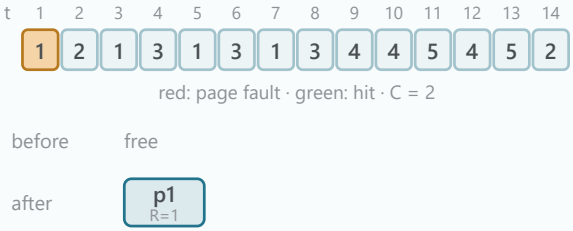
Step 1 / 8

VARIABLES

`t = 1` `p = 1` `last = 0` `t-last = –` `n = 0`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
5          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
▶      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



gap = 1 ≤ C

t = 1, first fault (p1): last = 0, gap 1 ≤ C. Nothing to evict; p1 is added with R = 1.

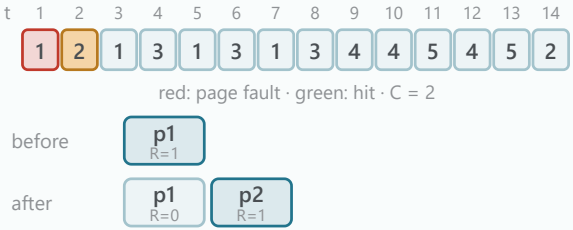
Step 2 / 8

VARIABLES

`t = 1` `p = 1` `last = 0` `t−last = 1` `n = 1`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
5          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
▶      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



gap = 1 ≤ C

t = 2, fault on s2: gap 1 ≤ C, faults are **frequent**. Nothing is evicted, R bits are cleared, and the set grows to 2 pages.

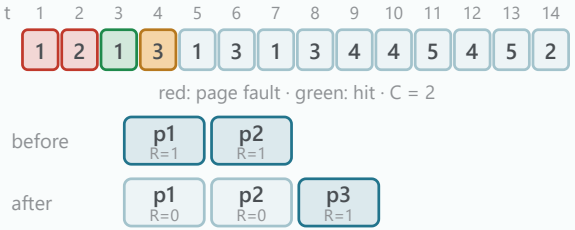
Step 3 / 8

VARIABLES

`t = 2` `p = 2` `last = 1` `t−last = 1` `n = 2`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
5          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
▶      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



gap = 2 ≤ C

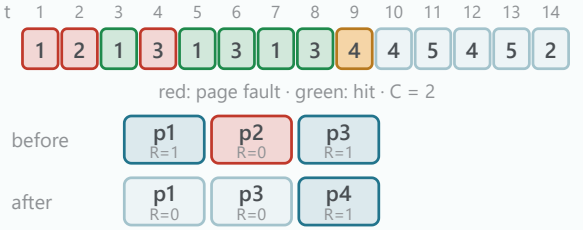
t = 4, fault on s3: gap 2 ≤ C, faults are **frequent**. Nothing is evicted, R bits are cleared, and the set grows to 3 pages.

VARIABLES

`t = 4` `p = 3` `last = 2` `t−last = 2` `n = 3`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
▶          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
8      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



gap = 5 > C

t = 9, fault on s4: gap 5 > C, faults are **rare**. Pages unused since the last fault (R = 0), s2, are evicted; the set shrinks.

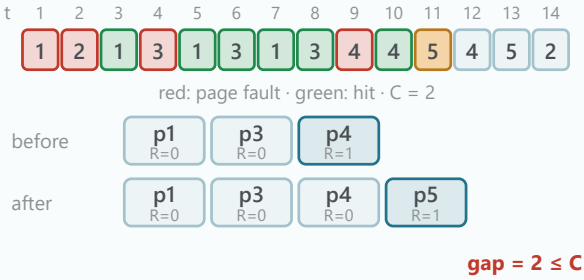
Step 5 / 8

VARIABLES

`t = 9` `p = 4` `last = 4` `t−last = 5` `n = 3`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
5          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
▶      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



t = 11, fault on s5: gap 2 ≤ C, faults are **frequent**. Nothing is evicted, R bits are cleared, and the set grows to 4 pages.

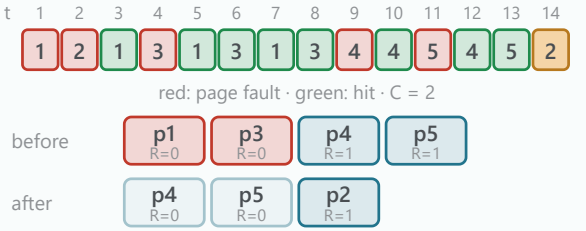
Step 6 / 8

VARIABLES

`t = 11` `p = 5` `last = 9` `t−last = 2` `n = 4`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
▶    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
▶          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
8      s->set[i].R = 0;      /* clear */
▶    add(s, p);              /* with R = 1 */
10   s->last = t;
11 }
```



t = 14, fault on s2: gap 3 > C, faults are **rare**. Pages unused since the last fault (R = 0), s3, s1, are evicted; the set shrinks.

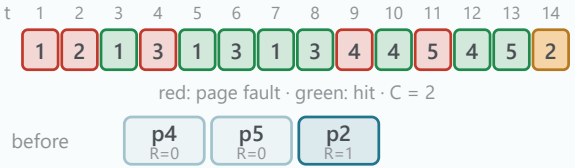
Step 7 / 8

VARIABLES

`t = 14` `p = 2` `last = 11` `t−last = 3` `n = 3`

PFF: Code Trace

```
1  void pff(Proc *s, int p, int t) {
2    if (t - s->last > C) {  /* rare faults */
3      for (int i = s->n - 1; i >= 0; i--)
4        if (s->set[i].R == 0)
5          evict(s, i);    /* unused */
6    }
7    for (int i = 0; i < s->n; i++)
8      s->set[i].R = 0;      /* clear */
9    add(s, p);              /* with R = 1 */
▶    s->last = t;
11 }
```



6 faults in 14 references. When the locality changed (from 1–3 to 4–5), faults first became frequent and the set grew; then the old pages were released on a rare fault.

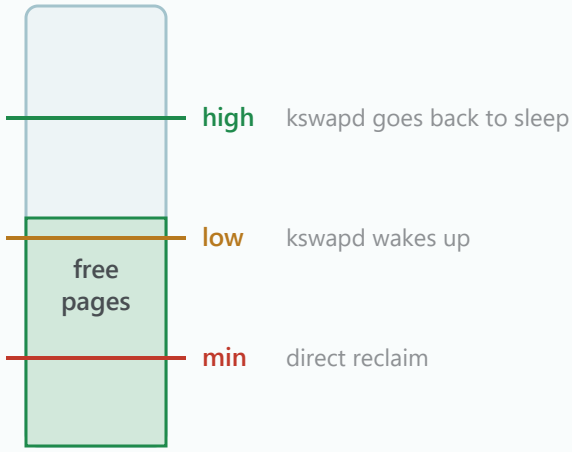
Step 8 / 8

VARIABLES

`t = 14` `p = 2` `last = 14` `t−last = 3` `n = 3`

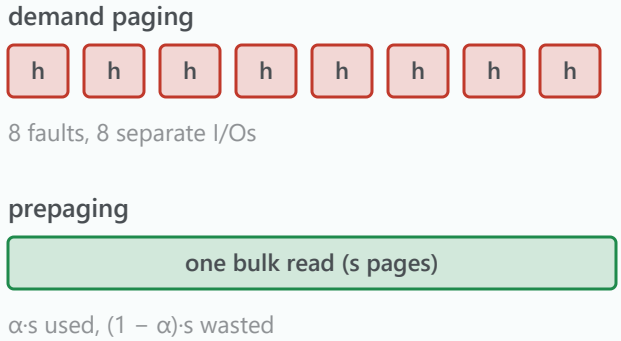
Cleaning Policy and the Paging Daemon

- Instead of searching for a victim at fault time, a **paging daemon** wakes up periodically; if free frames are scarce, it frees pages ahead of time.
- It **writes dirty pages to disk in advance**: at eviction time the page is clean, and a fault waits for only one read.
- A freed frame isn't wiped at once; if needed again it can be **reclaimed** from the pool (a minor fault).
- **Two-handed clock:** the front hand writes dirty pages / clears R, the back hand frees eligible pages.
- Linux: `kswapd` works per zone with *min / low / high* watermarks; `writeback` threads write dirty file pages.



Prepaging

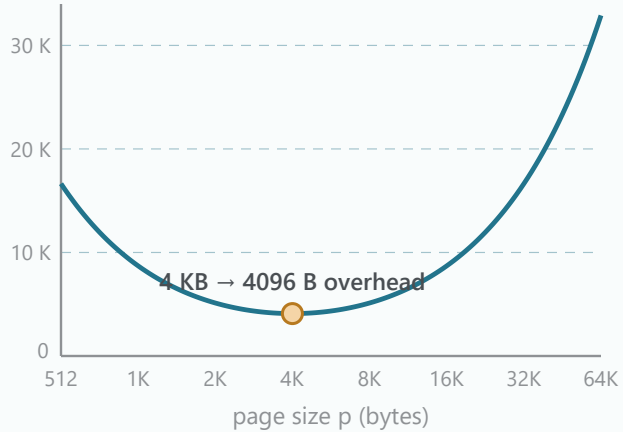
- Demand paging suffers a **fault storm** at startup (and on return from suspension). Prepaging brings needed pages in bulk *ahead of time*.
- s pages are prepaged and a fraction α is used: the gain is **α·s faults**, the cost **(1 − α)·s unneeded pages**. If α is near 0, prepaging loses.
- Remembering a suspended process's **working set** and loading it on resumption is the classic example.
- Linux: *readahead* for files, `page-cluster` for swap; applications give hints with `madvise(MADV_WILLNEED)` or `mmap(MAP_POPULATE)`.
- Windows: the Prefetcher and SysMain (formerly SuperFetch) load frequently used pages while idle.



Page Size

- Overhead = **se/p** (page table) + **p/2** (internal fragmentation in the last page); s: process size, e: entry size.
- Setting the derivative to zero: $-se/p^2 + 1/2 = 0 \rightarrow$ **$p = \sqrt{(2se)}$**. s = 1 MB, e = 8 B → p = $\sqrt{(2^{24})}$ = **4 KB**.

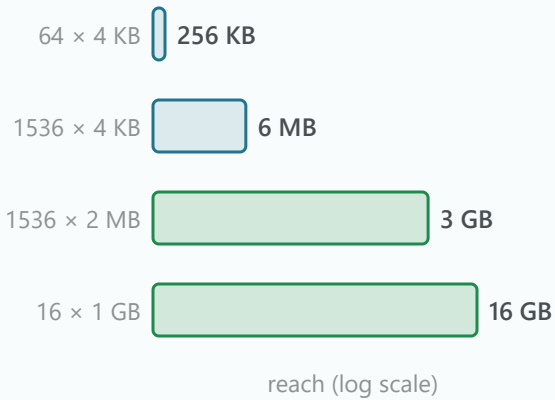
Small pages	Large pages
less internal fragmentation	smaller page table
track locality finely, less unneeded data	fewer faults, efficient I/O (seek/setup cost paid once)
many TLB misses	large TLB reach



2026: x86-64 4 KB (+2 MB, 1 GB); Apple Silicon 16 KB; Android 15+ supports 16 KB kernels; some ARM64 servers use 64 KB. 1 KB pages appear only on old systems.

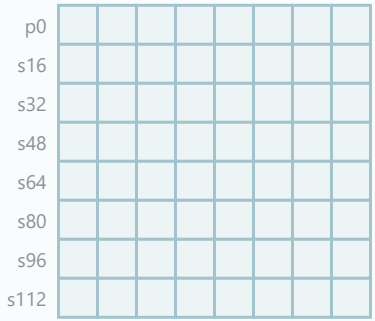
TLB Reach and Huge Pages

- **TLB reach** = number of TLB entries × page size: the amount of memory accessible from the TLB without a miss.
- If the working set doesn't fit within the reach, time goes to **page walks**. Adding entries is expensive (associative memory); larger pages are more effective.
- **Huge pages:** 2 MB and 1 GB on x86-64. Linux: `hugetlbfs` and *transparent huge pages* (THP, `khugepaged`); multi-size *folio*/mTHP support in 6.x kernels. Windows: the large-page privilege.
- The cost: internal fragmentation, finding large contiguous regions (compaction), longer copying/zeroing per page.
- Databases and virtual machines benefit most from huge pages.



Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
5     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
9     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

128 × 128 `int`, pages of 128 words: each **row** is a separate page. The process has 64 frames, none in use.

Step 1 / 8

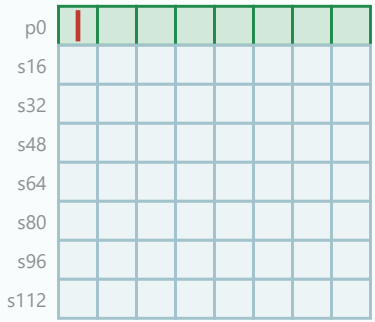
VARIABLES

`faults = 0`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
▶     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
9     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

A, in column 0: data[0][0] → a fault on page 0.

Step 2 / 8

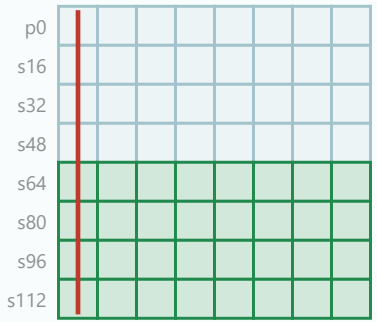
VARIABLES

`j = 0` `i = 0` `page = 0` `faults = 1`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1  int data[128][128];   /* row = 1 page */
2  /* A: column by column */
3  for (int j = 0; j < 128; j++)
4    for (int i = 0; i < 128; i++)
▶      data[i][j] = 0;
6  /* B: row by row */
7  for (int i = 0; i < 128; i++)
8    for (int j = 0; j < 128; j++)
9      data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

Along column 0 every `i` is a different page: 128 faults; they don't fit in 64 frames, so pages 0–63 were evicted long ago.

Step 3 / 8

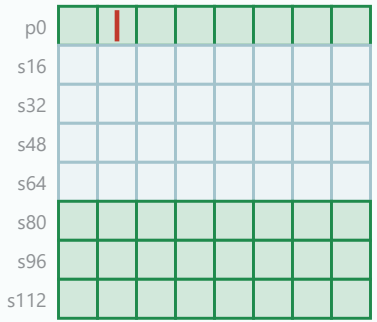
VARIABLES

`j = 0` `i = 127` `page = 127` `faults = 128`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
▶     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
9     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

Column 1: data[0][1] is again on page 0, but it was **evicted** → another fault.

Step 4 / 8

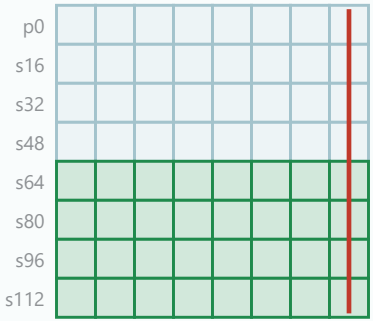
VARIABLES

`j = 1` `i = 0` `page = 0` `faults = 129`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
▶ for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
5     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
9     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

Every access faults: 128 × 128 = **16,384 faults**.

Step 5 / 8

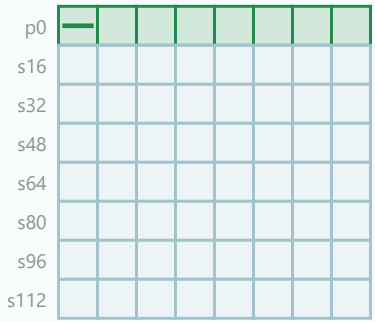
VARIABLES

`j = 128` `faults = 16384`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
5     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
▶     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

B, row 0: data[0][0] → a fault on page 0, then data[0][1…127] are on the same page: **hits**.

Step 6 / 8

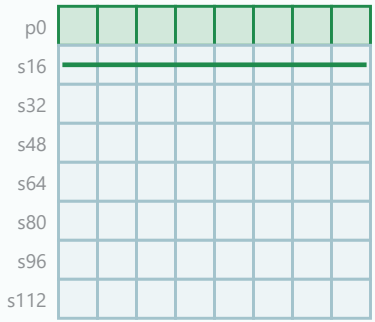
VARIABLES

`i = 0` `j = 0..127` `page = 0` `faults = 1`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
5     data[i][j] = 0;
6 /* B: row by row */
7 for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
▶     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

Row 1 → a single fault on page 1; one fault per row.

Step 7 / 8

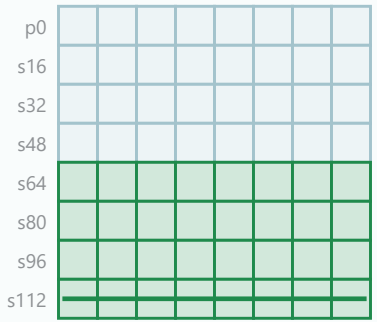
VARIABLES

`i = 1` `j = 0` `page = 1` `faults = 2`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

Program Structure and Locality

```
1 int data[128][128];   /* row = 1 page */
2 /* A: column by column */
3 for (int j = 0; j < 128; j++)
4   for (int i = 0; i < 128; i++)
5     data[i][j] = 0;
6 /* B: row by row */
▶ for (int i = 0; i < 128; i++)
8   for (int j = 0; j < 128; j++)
9     data[i][j] = 0;
```



green: rows (pages) in memory

64
frame
LRU
128 pages

128 faults in total: the same work, 128 times fewer faults.

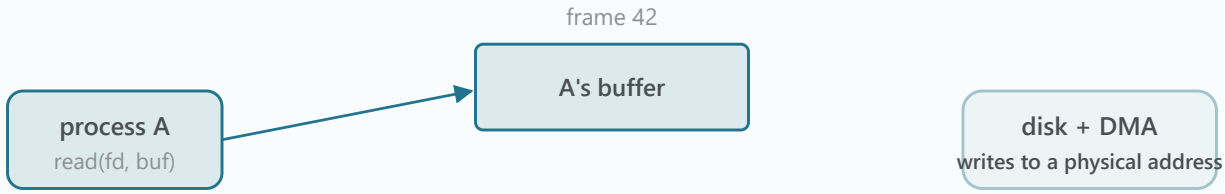
Step 8 / 8

VARIABLES

`i = 128` `faults = 128`

C stores arrays in row-major order, Fortran in column-major. Stacks have good locality; hash tables and pointer-chasing structures have poor locality; compilers can perform loop interchange.

I/O Interlock and Page Pinning

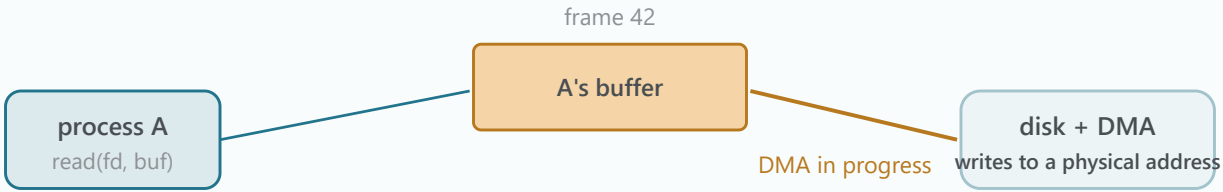


A requests a read from disk; the kernel programs the DMA with the physical address of **frame 42**, which holds the buffer.

Step 1 / 4

- Other pinned pages: parts of kernel code and data, and the memory of real-time processes (`mlock()` , `mlockall()`).
- Linux drivers pin user pages with `pin_user_pages()` (RDMA, GPUs, io_uring fixed buffers). An IOMMU lets devices use translated addresses too.
- A page of a newly loaded low-priority process that hasn't run yet can also be protected until it is used once.
- The risk: a locked page that is forgotten makes its frame unusable forever; so locking is limited (`ulimit -l`).

I/O Interlock and Page Pinning

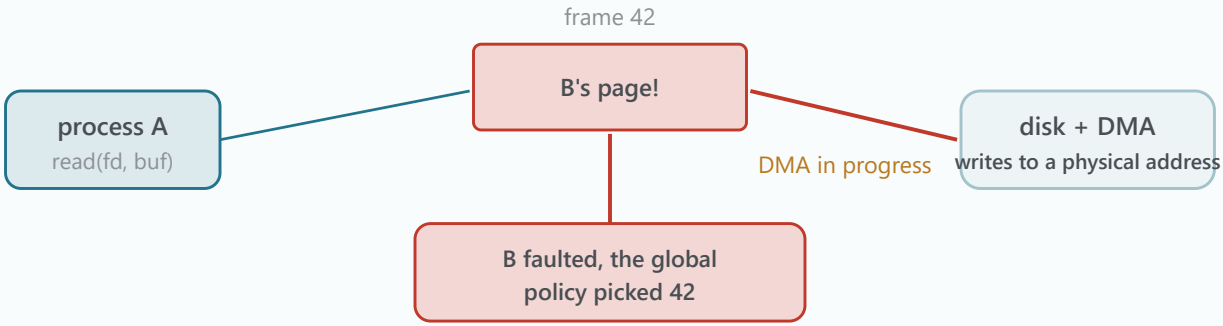


A blocks while the device queues/works. Meanwhile frame 42 looks like an ordinary page.

Step 2 / 4

- Other pinned pages: parts of kernel code and data, and the memory of real-time processes (`mlock()` , `mlockall()`).
- Linux drivers pin user pages with `pin_user_pages()` (RDMA, GPUs, io_uring fixed buffers). An IOMMU lets devices use translated addresses too.
- A page of a newly loaded low-priority process that hasn't run yet can also be protected until it is used once.
- The risk: a locked page that is forgotten makes its frame unusable forever; so locking is limited (`ulimit -l`).

I/O Interlock and Page Pinning

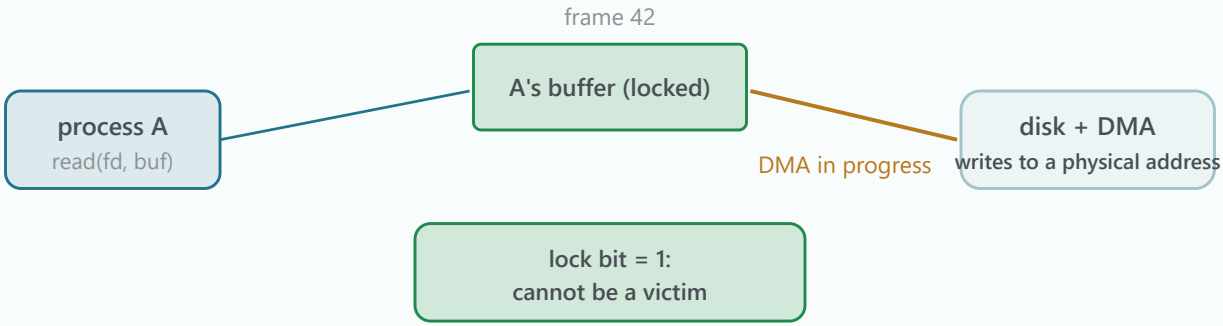


B page-faults; the global policy chooses 42 as the victim and loads B's page. When the DMA finishes, it writes **over B's page**: corruption.

Step 3 / 4

- Other pinned pages: parts of kernel code and data, and the memory of real-time processes (`mlock()` , `mlockall()`).
- Linux drivers pin user pages with `pin_user_pages()` (RDMA, GPUs, io_uring fixed buffers). An IOMMU lets devices use translated addresses too.
- A page of a newly loaded low-priority process that hasn't run yet can also be protected until it is used once.
- The risk: a locked page that is forgotten makes its frame unusable forever; so locking is limited (`ulimit -l`).

I/O Interlock and Page Pinning



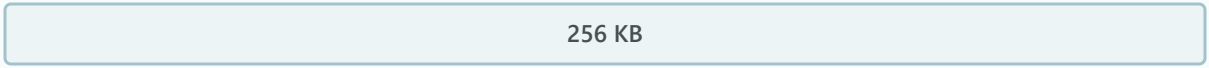
The fix: the frame is **locked** (pinned) for the duration of the I/O and never chosen as a victim; or the I/O goes to kernel buffers and is copied afterward.

Step 4 / 4

- Other pinned pages: parts of kernel code and data, and the memory of real-time processes (`mlock()` , `mlockall()`).
- Linux drivers pin user pages with `pin_user_pages()` (RDMA, GPUs, io_uring fixed buffers). An IOMMU lets devices use translated addresses too.
- A page of a newly loaded low-priority process that hasn't run yet can also be protected until it is used once.
- The risk: a locked page that is forgotten makes its frame unusable forever; so locking is limited (`ulimit -l`).

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.



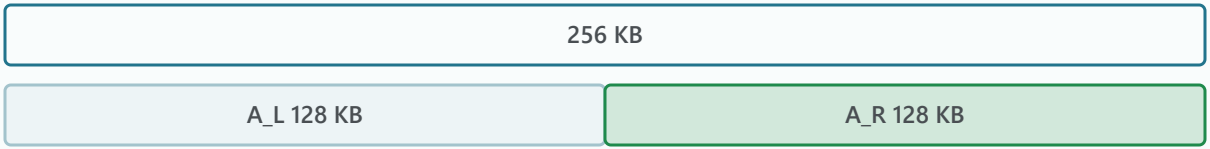
The kernel has a physically **contiguous** 256 KB region; a 21 KB request arrives.

Step 1 / 6

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.

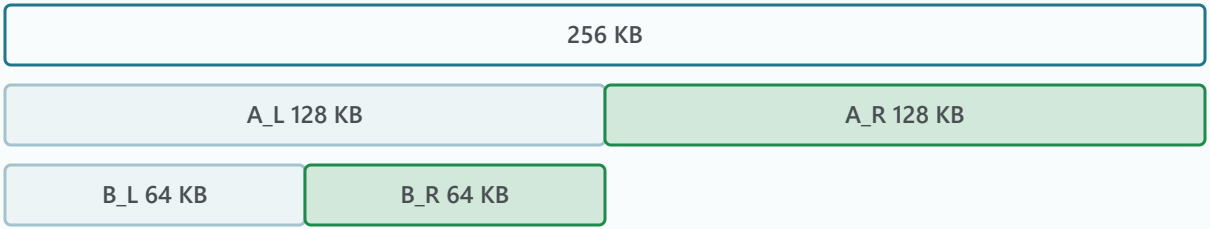


The block is split into two equal **buddies**: A_L and A_R, 128 KB each.

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.

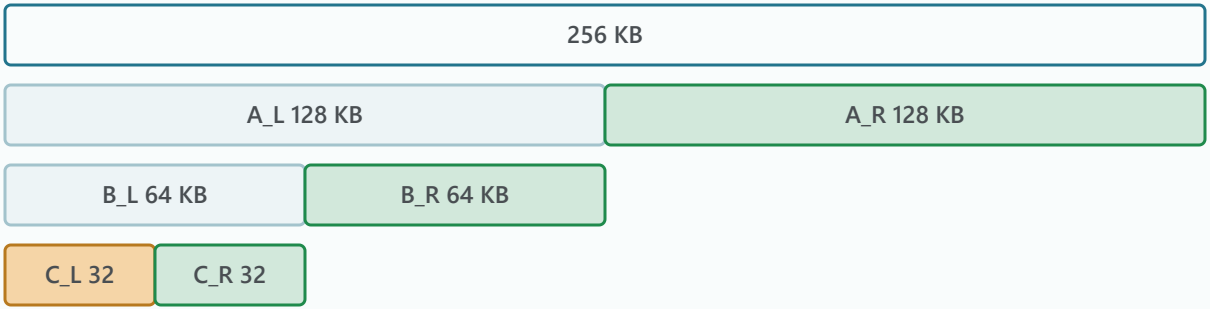


A_L is split again: B_L, B_R (64 KB).

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.

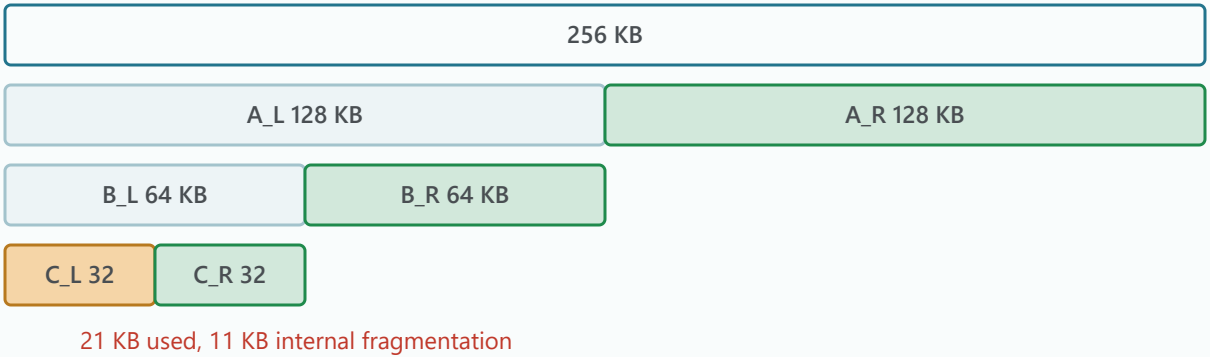


B_L is split: C_L, C_R (32 KB). The smallest power of 2 that holds 21 KB is 32 KB → C_L is allocated.

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.



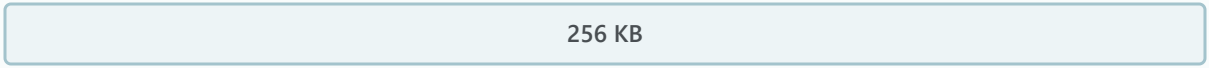
The cost is internal fragmentation: 21 requested, 32 KB given.

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

KERNEL MEMORY

Kernel Memory: The Buddy System

Kernel memory differs from user memory: it is often **not paged**, needs structures of various sizes, and must be **physically contiguous** for DMA.



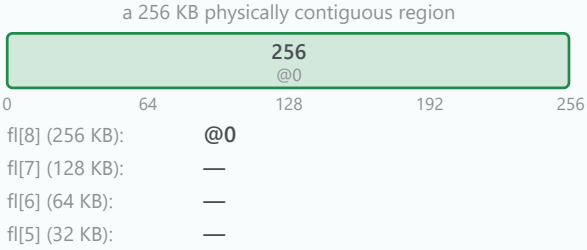
C_L + C_R → B_L, B_L + B_R → A_L, A_L + A_R → 256 KB

On release, free buddies **coalesce quickly**: a buddy's address is found by flipping a single bit.

The Linux page allocator is a buddy allocator: lists of orders 0 to 10 (4 KB … 4 MB), `cat /proc/buddyinfo`.

The Buddy Allocator: Code Trace

```
▶ int alloc(int k) {          /* 2^k KB */
▶   int j = k;
3   while (fl[j] == NULL) j++;
4   int a = pop(j);
5   while (j > k) {   /* split, right half free
  */
6     j--; push(j, a + (1 << j));
7   }
8   return a;
9 }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



A single free block: 256 KB at @0 (order 8). The 21 KB request rounds up to 32 KB (2^5): `A = alloc(5)`.

Step 1 / 12

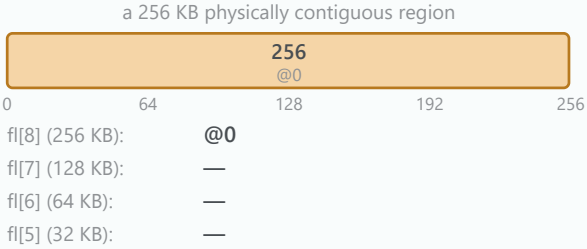
VARIABLES

`k = 5` `j = 5` `a = –`

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
▶   while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free
   */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



fl[5], fl[6], fl[7] are empty; the first nonempty list is fl[8]: j = 8. A larger block will be split.

Step 2 / 12

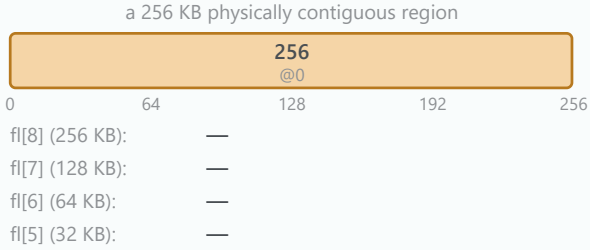
VARIABLES

k = 5 | j = 8 | a = –

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
▶    int a = pop(j);
5    while (j > k) {   /* split, right half free
   */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



The 256 KB block is taken from the list: a = 0.

Step 3 / 12

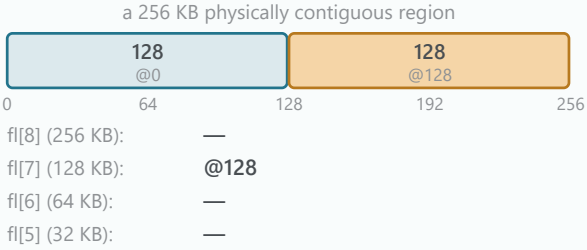
VARIABLES

`k = 5` `j = 8` `a = 0`

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
▶    while (j > k) {   /* split, right half free
   */
▶      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



j = 7: the block is split; the right half, @128 (128 KB), goes on fl[7] and the left half is split further.

Step 4 / 12

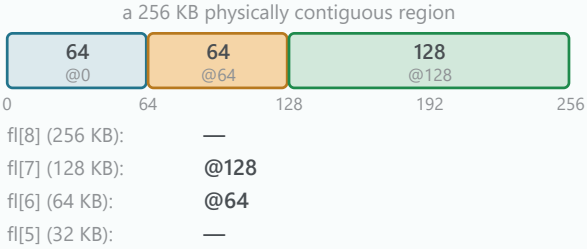
VARIABLES

`k = 5` `j = 7` `a = 0`

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
▶    while (j > k) {   /* split, right half free
   */
▶      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



j = 6: the block is split; the right half, @64 (64 KB), goes on fl[6] and the left half is split further.

Step 5 / 12

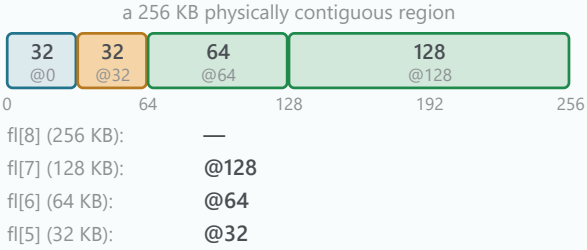
VARIABLES

`k = 5` `j = 6` `a = 0`

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
▶    while (j > k) {   /* split, right half free
   */
▶      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



j = 5: the last split; @32 goes to fl[5]. Since j = k the loop ends, and the left half (@0, 32 KB) is given to **A**.

Step 6 / 12

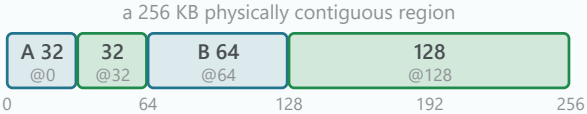
VARIABLES

`k = 5` `j = 5` `a = 0`

OUTPUT

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
▶    while (fl[j] == NULL) j++;
▶    int a = pop(j);
5    while (j > k) {   /* split, right half free */
6      j--; push(j, a + (1 << j));
7    }
▶    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



fl[8] (256 KB): —
fl[7] (128 KB): @128
fl[6] (64 KB): —
fl[5] (32 KB): @32

`B = alloc(6)` : @64 is ready on fl[6]; no split needed, it returns at once. Internal fragmentation: 32 − 21 = 11 KB in A.

Step 7 / 12

VARIABLES

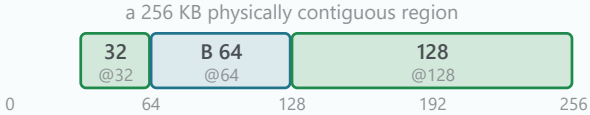
`k = 6` `j = 6` `a = 64`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
▶ void release(int a, int k) {
▶   int b = a ^ (1 << k);     /* buddy */
12   while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



fl[8] (256 KB): —
fl[7] (128 KB): @128
fl[6] (64 KB): —
fl[5] (32 KB): @32

`release(0, 5)` : A is being returned. The buddy's address is found by flipping one bit: 0 ⊕ 32 = **32**.

Step 8 / 12

VARIABLES

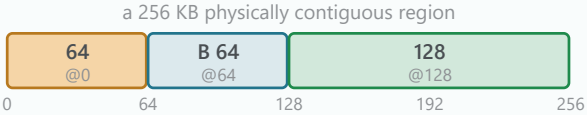
`a = 0` `k = 5` `b = 32`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
▶    while (k < MAX && remove(k, b)) {
▶      a = a < b ? a : b;      /* merge */
▶      k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



fl[8] (256 KB): —
fl[7] (128 KB): @128
fl[6] (64 KB): —
fl[5] (32 KB): —

@32 is on fl[5]: the buddy is free, so they **merge** → 64 KB at @0. The new buddy is 0 ⊕ 64 = 64.

Step 9 / 12

VARIABLES

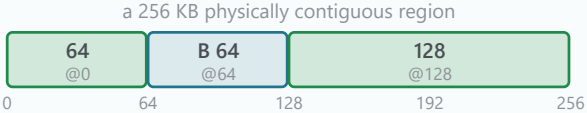
`a = 0` `k = 6` `b = 64`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free
   */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
▶    while (k < MAX && remove(k, b)) {
13     a = a < b ? a : b;      /* merge */
14     k++; b = a ^ (1 << k);
15   }
▶    push(k, a);
17 }
```



fl[8] (256 KB): —
fl[7] (128 KB): @128
fl[6] (64 KB): @0
fl[5] (32 KB): —

@64 isn't on fl[6] (B is using it): merging stops, and @0 (64 KB) goes on fl[6].

Step 10 / 12

VARIABLES

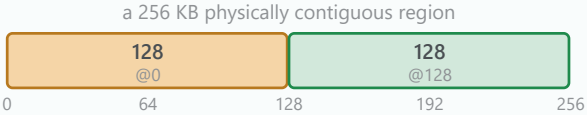
`a = 0` `k = 6` `b = 64`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
▶ void release(int a, int k) {
▶   int b = a ^ (1 << k);     /* buddy */
▶   while (k < MAX && remove(k, b)) {
▶     a = a < b ? a : b;      /* merge */
▶     k++; b = a ^ (1 << k);
15   }
16   push(k, a);
17 }
```



fl[8] (256 KB): —
fl[7] (128 KB): @128
fl[6] (64 KB): —
fl[5] (32 KB): —

`release(64, 6)` : its buddy 64 ⊕ 64 = 0 is free → a 128 KB block at @0 forms; the new buddy is @128.

Step 11 / 12

VARIABLES

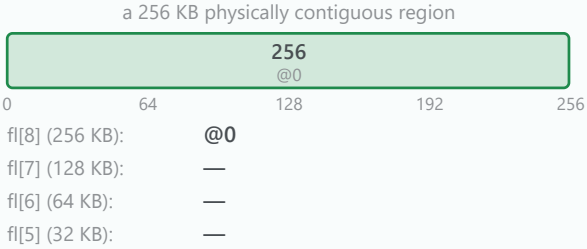
`a = 0` `k = 7` `b = 128`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

The Buddy Allocator: Code Trace

```
1  int alloc(int k) {          /* 2^k KB */
2    int j = k;
3    while (fl[j] == NULL) j++;
4    int a = pop(j);
5    while (j > k) {   /* split, right half free
   */
6      j--; push(j, a + (1 << j));
7    }
8    return a;
9  }
10 void release(int a, int k) {
11   int b = a ^ (1 << k);     /* buddy */
▶    while (k < MAX && remove(k, b)) {
▶      a = a < b ? a : b;      /* merge */
▶      k++; b = a ^ (1 << k);
15   }
▶    push(k, a);
17 }
```



@128 is free too: they merge into 256 KB; k = MAX, so it stops. The region is back to **one piece**, as at the start.

Step 12 / 12

VARIABLES

`a = 0` `k = 8` `b = –`

OUTPUT

```
A = @0 (32 KB)
B = @64 (64 KB)
```

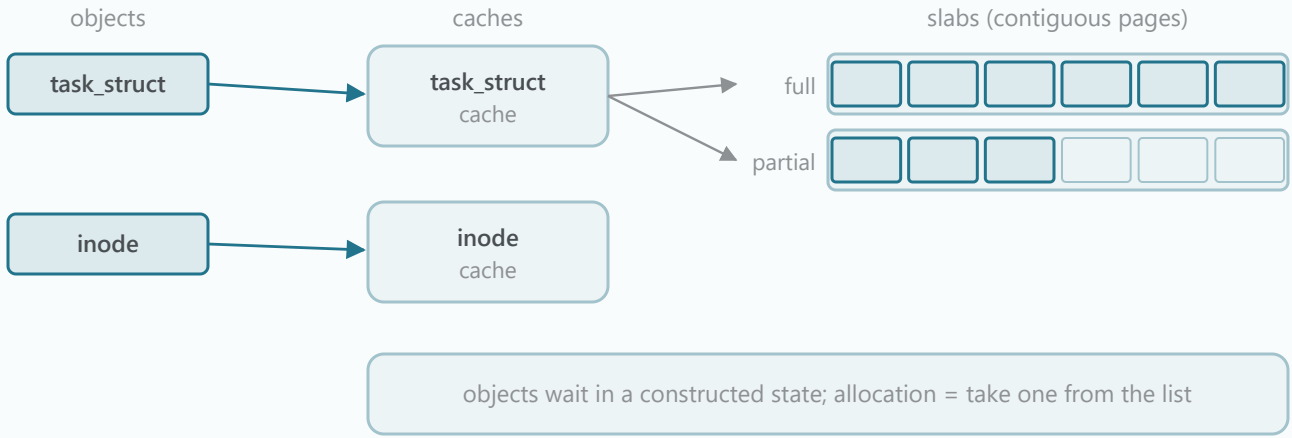
Buddy: An Example Request Sequence

Start with a single free 1 MiB (1024 K) block; each request rounds up to the next power of 2. A buddy's address is *address XOR size*.

Event	Block	Split or merge	Internal fragmentation
A = 70 K	128 K	1024 → 512 + 512 → 256 + 256 → 128 + 128	58 K
B = 35 K	64 K	128 → 64 + 64	29 K
C = 80 K	128 K	256 → 128 + 128	48 K
A released	—	its buddy (128–256) is split and holds B → no merge	—
D = 60 K	64 K	B's free buddy is given; no split needed	4 K
B released	—	its buddy D is in use → no merge	—
D released	—	64 + 64 → 128, with A's free 128 → 256	—
C released	—	128 + 128 → 256 → 512 → 1024: memory is a single block again	—

Total internal fragmentation for A, B, and C: 58 + 29 + 48 = 135 K. Buddy allocation and coalescing are fast (a free list per size, O(log n)), but in the worst case about half of a block is wasted.

The Slab Allocator

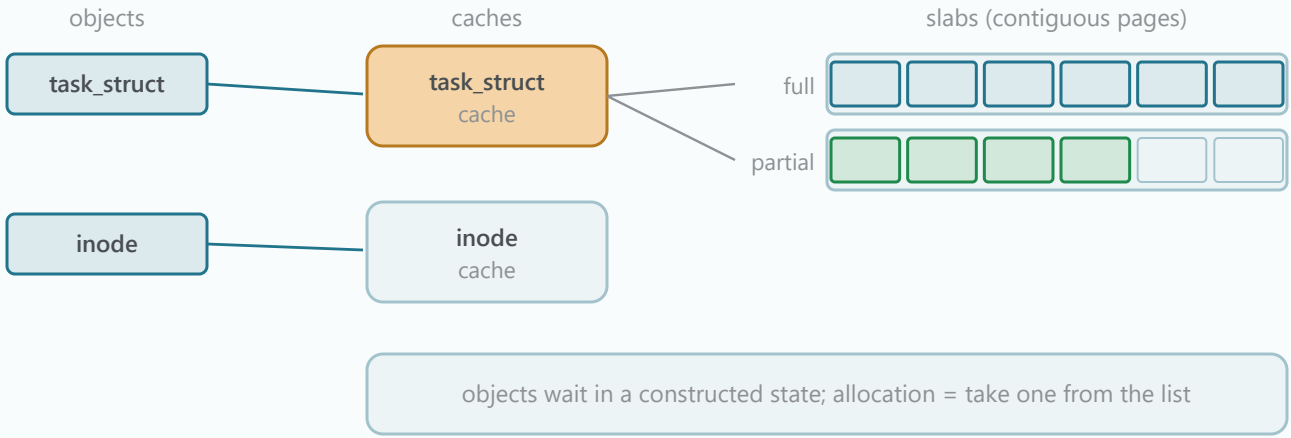


One **cache** for each frequently used kernel object; a cache consists of one or more **slabs**, and a slab of contiguous pages.

Step 1 / 4

- Bonwick (Solaris, 1994). Benefits: no internal fragmentation, O(1) allocation/freeing, hot ready-made objects in the cache, and a lock-free fast path with per-CPU lists.
- Linux: SLOB was removed in 6.4 and classic SLAB in 6.8 (2024); today the only allocator is **SLUB**. `kmalloc()` draws from size-class slab caches; monitor with `slabtop`, `/proc/slabinfo`.

The Slab Allocator

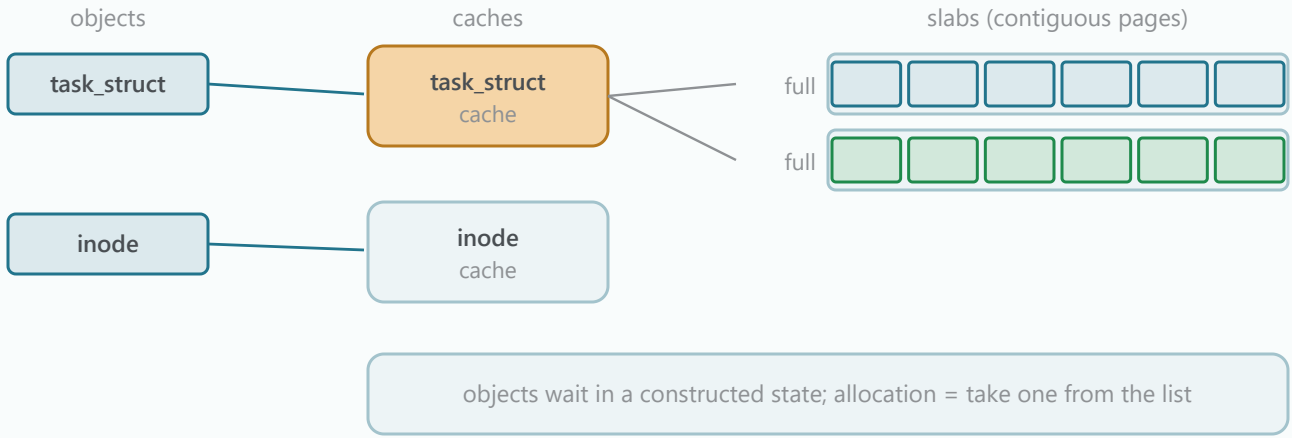


A request for a new `task_struct` (e.g., `fork()`): first look for a free object in a **partial** slab; one is found and returned.

Step 2 / 4

- Bonwick (Solaris, 1994). Benefits: no internal fragmentation, O(1) allocation/freeing, hot ready-made objects in the cache, and a lock-free fast path with per-CPU lists.
- Linux: SLOB was removed in 6.4 and classic SLAB in 6.8 (2024); today the only allocator is **SLUB**. `kmalloc()` draws from size-class slab caches; monitor with `slabtop`, `/proc/slabinfo`.

The Slab Allocator

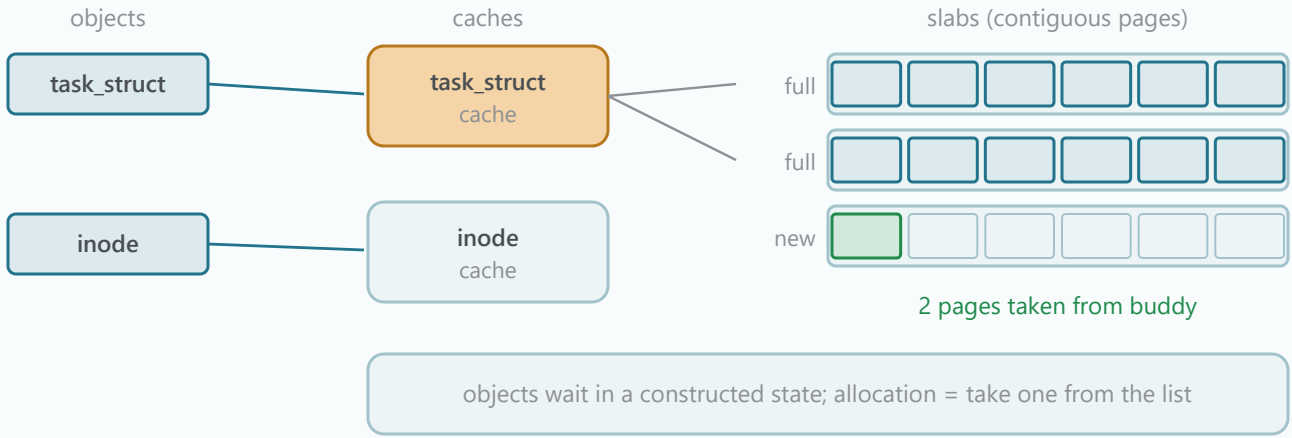


The partial slab is now full. No fragmentation: objects are exactly sized and packed side by side within the page.

Step 3 / 4

- Bonwick (Solaris, 1994). Benefits: no internal fragmentation, O(1) allocation/freeing, hot ready-made objects in the cache, and a lock-free fast path with per-CPU lists.
- Linux: SLOB was removed in 6.4 and classic SLAB in 6.8 (2024); today the only allocator is **SLUB**. `kmalloc()` draws from size-class slab caches; monitor with `slabtop`, `/proc/slabinfo`.

The Slab Allocator



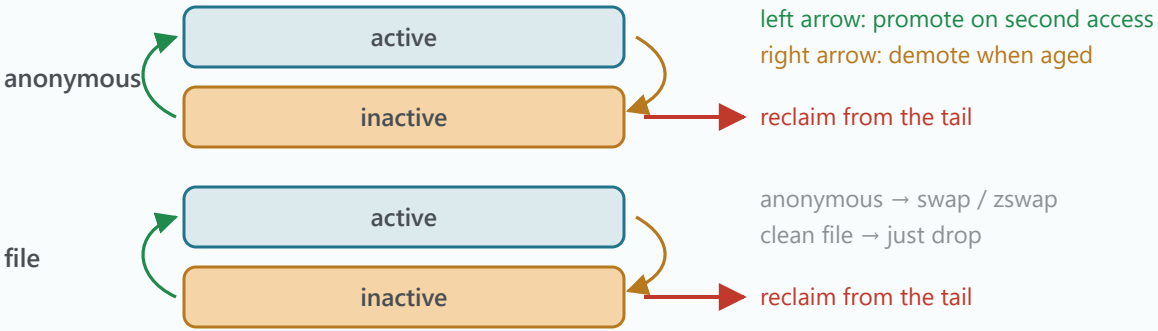
No partial or empty slab is left: new pages come from the buddy allocator and a new slab is created. Empty slabs are returned under memory pressure.

Step 4 / 4

- Bonwick (Solaris, 1994). Benefits: no internal fragmentation, O(1) allocation/freeing, hot ready-made objects in the cache, and a lock-free fast path with per-CPU lists.
- Linux: SLOB was removed in 6.4 and classic SLAB in 6.8 (2024); today the only allocator is **SLUB**. `kmalloc()` draws from size-class slab caches; monitor with `slabtop`, `/proc/slabinfo`.

Linux: Page Reclaim

- When free pages drop below the *low* watermark, **kswapd** wakes; below *min*, the allocating process does **direct reclaim** (latency!).
- The classic LRU is a pair of clock-like lists: **active / inactive**, separately for anonymous and file pages (plus *unevictable* for locked pages).
- `vm.swappiness` (0–200) sets the preference between anonymous and file pages. When memory runs out, the **OOM killer** kills a process.



Linux: MGLRU, zswap, and zram

MGLRU (6.1, 2022)

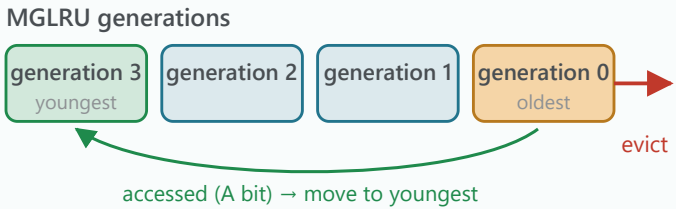
Multi-generational LRU: pages are divided into 4 generations; access bits are collected by walking page tables, and the victim comes from the oldest generation. Less CPU, better victim choice. `/sys/kernel/mm/lru_gen/enabled`

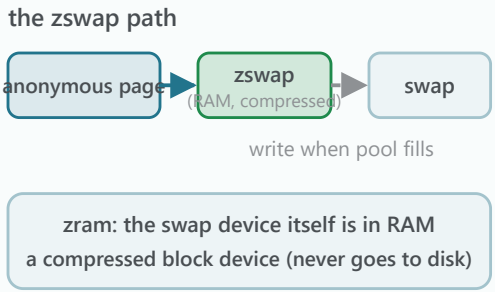
zswap

A *compressed cache* in front of swap: an anonymous page is compressed in RAM before going to disk; when the pool fills, the oldest entries are written to real swap.

zram

A compressed block device in RAM, used as swap. Fedora (since 33), Android, and ChromeOS use it by default.

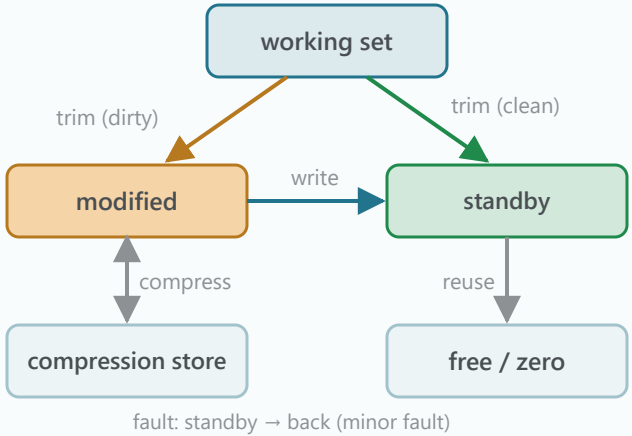




MGLRU is the default on ChromeOS and Android; many distribution kernels enable it as a build option. Compression costs CPU, but it is hundreds of times cheaper than going to disk.

Windows: Working Sets and Compression

- Each process has a **working set** (with minimum/maximum limits); a fault either makes room in the set or grows it.
- When memory runs low, the **working-set manager** (balance set manager) *trims* sets; a trimmed page isn't discarded at once but goes to the **standby** or **modified** list and can be recovered with a minor fault.
- **Memory compression** (Windows 10, 2015): trimmed pages are kept in a compression store in a system process; writing to `pagefile.sys` is the last resort.
- Prefetching: the Prefetcher and SysMain; large pages via `VirtualAlloc(MEM_LARGE_PAGES)`.
- macOS/iOS are similar: compressed memory, no swap on iOS, and app termination under pressure (Jetsam).



Summary

Mechanism

Demand paging: invalid entry → page fault → bring in the page → update the table → restart the instruction. COW and mmap use the same mechanism.

Performance

EAT is extremely sensitive to the fault rate: on an HDD, p = 0.001 slows memory down 251×. Fast SSDs and compression shrink the penalty but don't eliminate it.

Policy

On the common string (3 frames): OPT 9, LRU 12, clock 14, FIFO 15 faults. In practice: variants of clock, aging, and WSClock; MGLRU on Linux.

System

Allocation (equal/proportional, local/global), the working set and PFF against thrashing, page size and TLB reach, pinning, buddy + slab.

Thanks to locality, only processes' **working sets** need to be in memory; the OS's job is to keep those sets resident with little I/O.

SUMMARY

Review Questions

1. How many faults does FIFO make on `1 2 3 4 1 2 5 1 2 3 4 5` with 3 and 4 frames? Why doesn't LRU show this anomaly?
2. Memory access 200 ns, fault service 8 ms: EAT for p = 0.001? Largest p for a 10% slowdown?
3. 62 frames, two processes of 10 and 127 pages: what does proportional allocation give?
4. What is the TLB reach of a 64-entry TLB with 4 KB and with 2 MB pages?
5. After `fork()` the child writes 3 of 100 pages, then calls `exec()`. How many pages are copied?
6. Distinguish minor and major page faults with an example of each.
7. Why does thrashing lower CPU utilization? How does the working-set model prevent it?
8. What is the optimal page size for s = 4 MB, e = 8 B?
9. Why must a buffer page be locked during I/O?
10. In a 1 MiB buddy system, what block does a 70 K request get? How much internal fragmentation?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapter 10: Virtual Memory).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson (Chapter 3: Memory Management).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org: "Swapping: Mechanisms", "Swapping: Policies", "Complete VM Systems".
- L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", *IBM Systems Journal* 5(2), 1966; P. J. Denning, "The Working Set Model for Program Behavior", *CACM* 11(5), 1968.
- R. W. Carr, J. L. Hennessy, "WSCLOCK: A Simple and Effective Algorithm for Virtual Memory Management", SOSP 1981; J. Bonwick, "The Slab Allocator", USENIX Summer 1994.
- Linux kernel documentation: `admin-guide/mm/multigen_lru.rst`, `admin-guide/mm/zswap.rst`, `admin-guide/mm/transhuge.rst`, `admin-guide/blockdev/zram.rst`.
- P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7th ed., Part 1, Chapter 5: Memory management.

The End

Chapter 9: Virtual Memory

Previous: Chapter 8: Memory Management · Next: Chapter 10: File Systems