CI583: Data Structures and Operating Systems
Memory management part 4

Outline

Placement policies

Once a page is loaded into a page frame, exactly where it was loaded may not be much of an issue, since page frames are held in RAM.

Some systems however, such as Linux, use the placement policy to reserve *lower* addresses for Direct Memory Access (DMA) devices, many of which can only handle 24-bit addresses (recall our discussion of IO modules, DMA and PIO).

Placement policies

An area of primary storage is also reserved for unpageable memory (e.g. those pages containing parts of the OS, especially its modules such as the page fault handler).

Linux reserves 1GB of the address space (much of which is pageable), whereas Windows divides the address space equally between user processes and the OS.

Replacement policies

The replacement policy needs to decide which page frames can be reused as more page faults occur, and should be able to anticipate the need for more free frames.

The pages to remove will certainly include those belonging to processes that are no longer running.

Other situations are less clear cut though – replacement policies are, from an algorithmic point of view, the most complex aspect of managing virtual memory.

Replacement policies

If we have no free page frames, it doesn't make much sense to wait for a fault before we make room, potentially incurring an IO cost.

Given a replacement policy, we can run it in a separate thread page-out thread.

Replacement policies

As a starting point for a replacement policy, we might consider a first-in-first-out (FIFO) approach.

When we run out of page frames and need to make room, we remove the page that has been in primary memory for the longest time.

This is not likely to work well though: the page that has been in memory for the longest time might also be the most frequently accessed.

Replacement policies

In an ideal world, we would like to remove the pages whose *next access is at the farthest point in the future*.

In practice, this is impossible to know, but the idealised goal is known as Belady's optimal replacement algorithm (1966) and provided motivation for the algorithms that have actually been implemented.

Replacement policies

To simulate Belady's algorithm, we can guess the page whose access point is the farthest in the future by finding the page whose last access is the farthest in the past.

This doesn't follow, logically speaking, but is a decent starting point.

This is the least-recently used (LRU) algorithm.

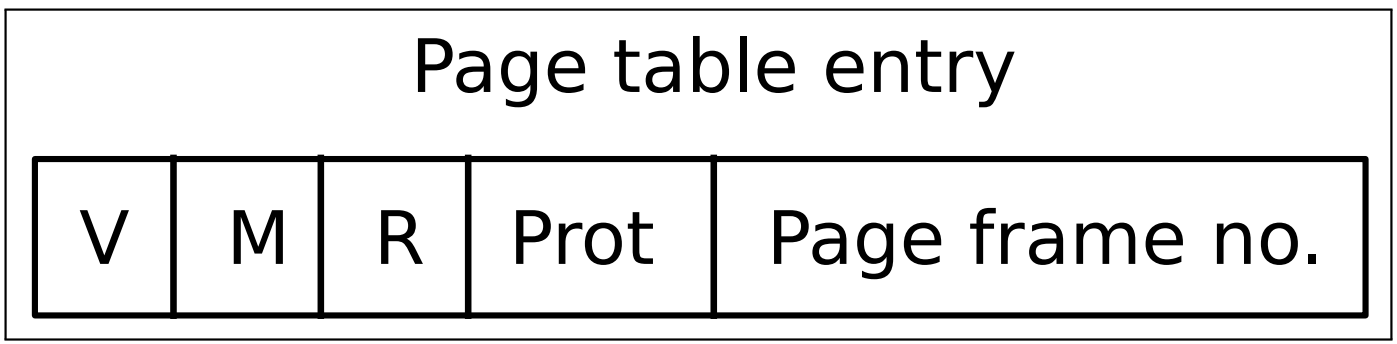
Replacement policies

Keeping a list of all pages ordered by access time is much too expensive to consider doing, so how to proceed?

The way that LRU is implemented is to define a time period (long enough for thousands of page frame references), and consider any page which hasn't been accessed in that period to be a candidate for removal.

Replacement policies

So, we keep a reference bit in page table entries and set it to 1 every time the entry is used to look up a frame. At the end of the time period, we inspect all reference bits. If it is 0, this frame hasn't been referenced and can be removed. If it is 1, we set it to 0, ready for the next cycle.

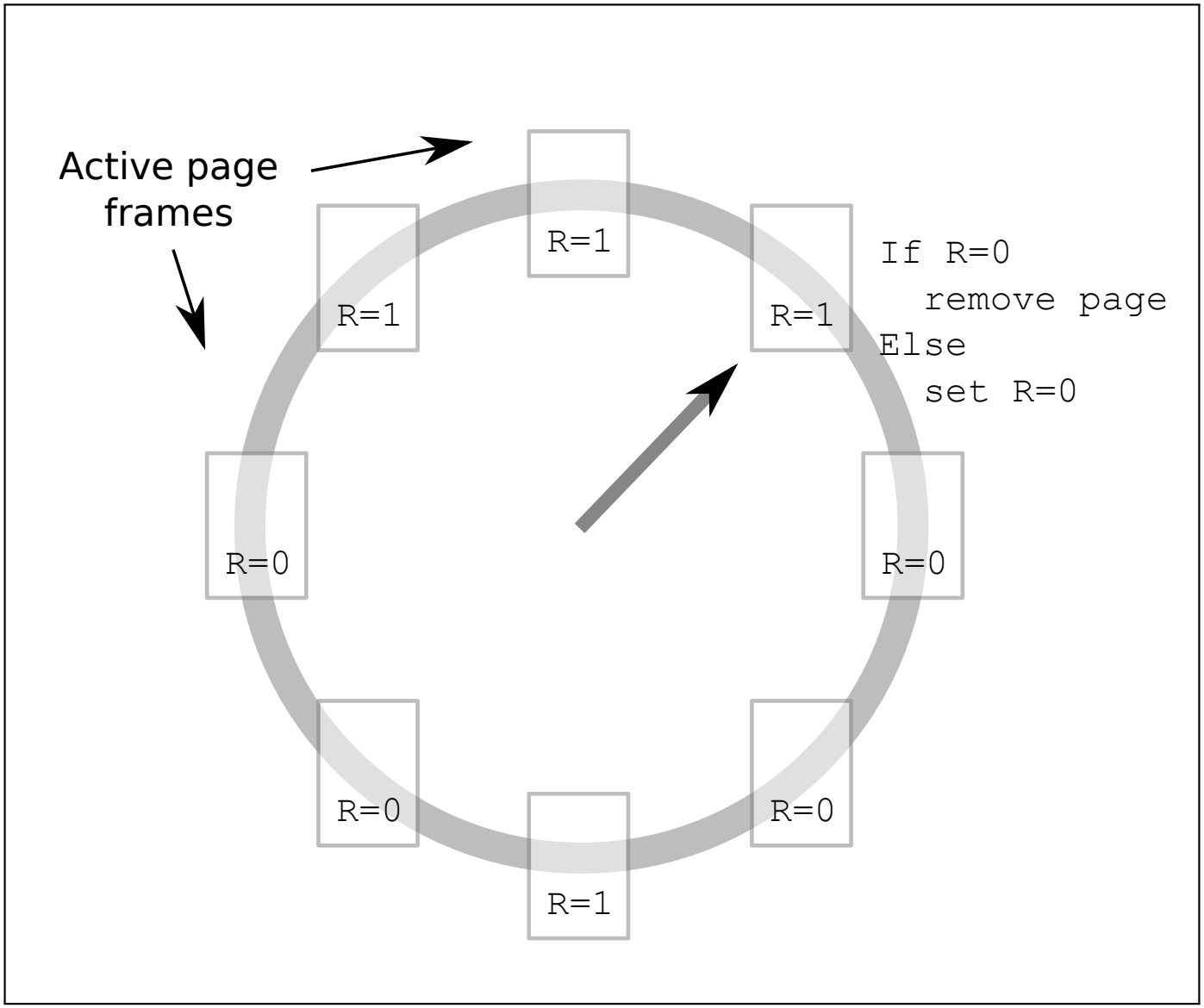


Replacement policies

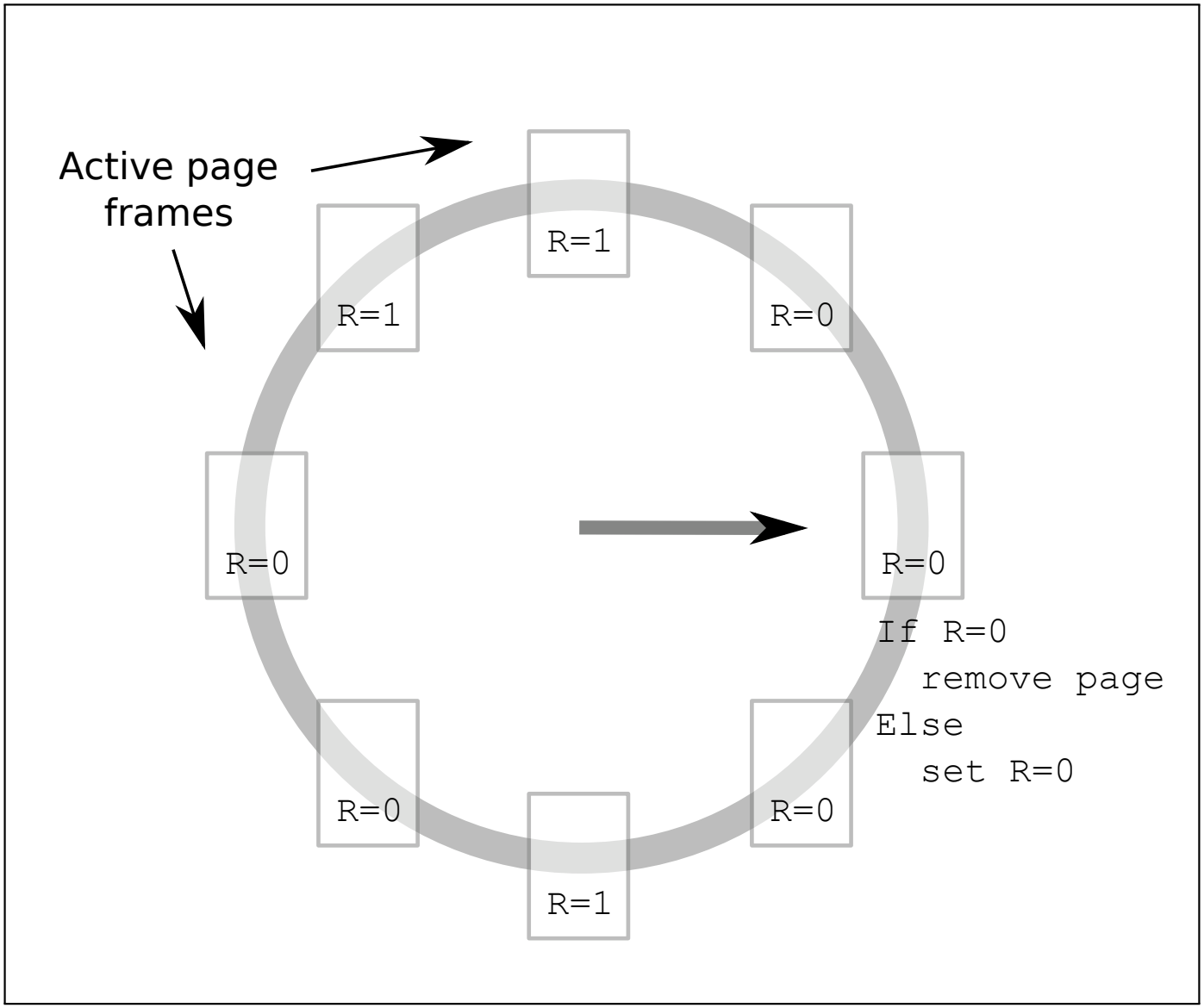
Simple LRU is a relatively coarse approach – apart from anything else, it will take some variable amount of time to inspect the page table (depending on what is in it). An alternative is to use a *continuous* approach – the clock algorithm.

- Active page frames are gathered in a circularly linked list.
- The page-out thread traverses the list.
- For each frame, if $R = 0$, add the page to the list of free frames (including writing changes to the backing store if necessary).
- If $R = 1$, set $R = 0$ and continue.

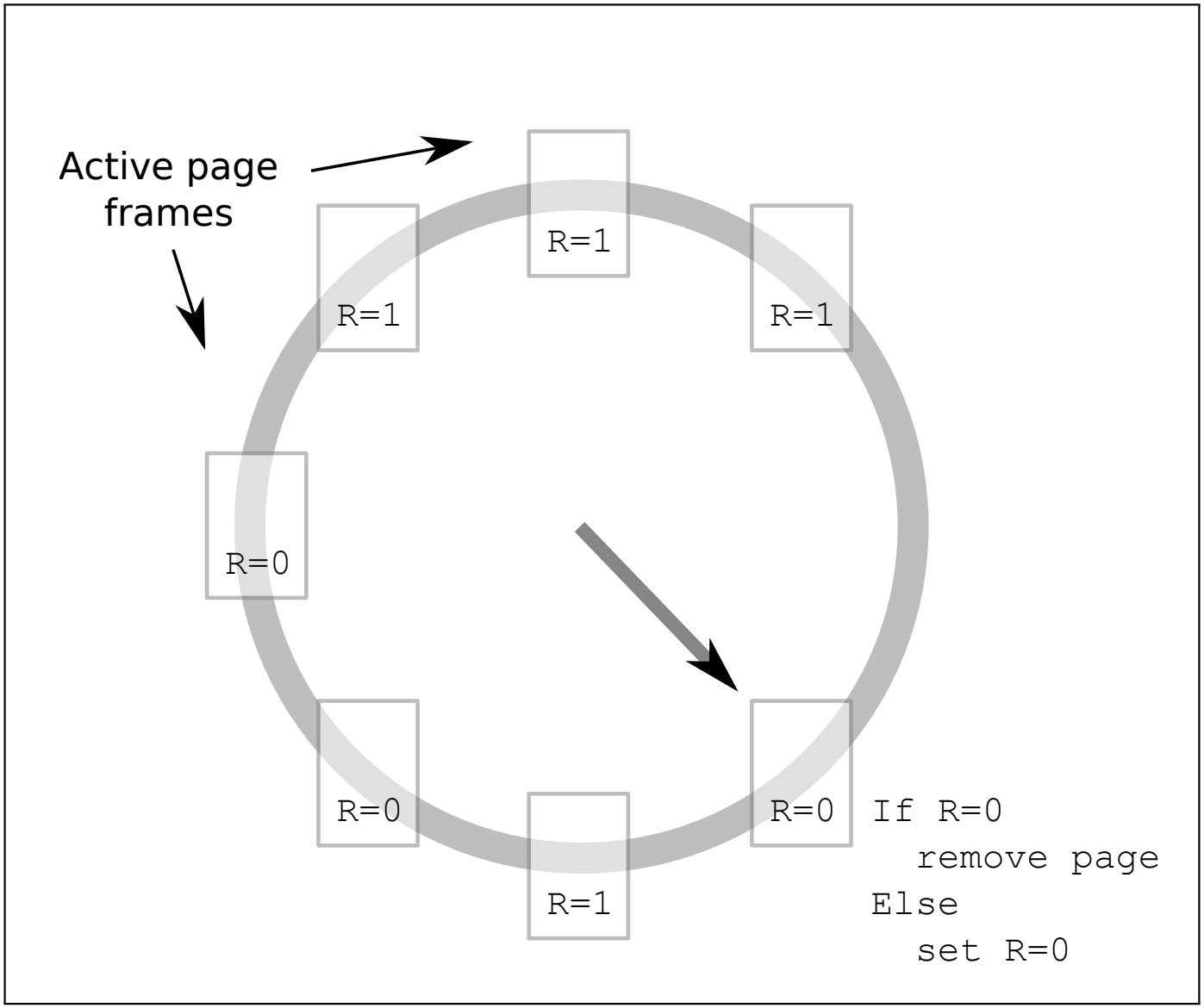
Replacement policies



Replacement policies



Replacement policies



Replacement policies

The clock algorithm depends on the page-out thread traversing the list faster or more slowly depending on how many pages are in the list of free frames.

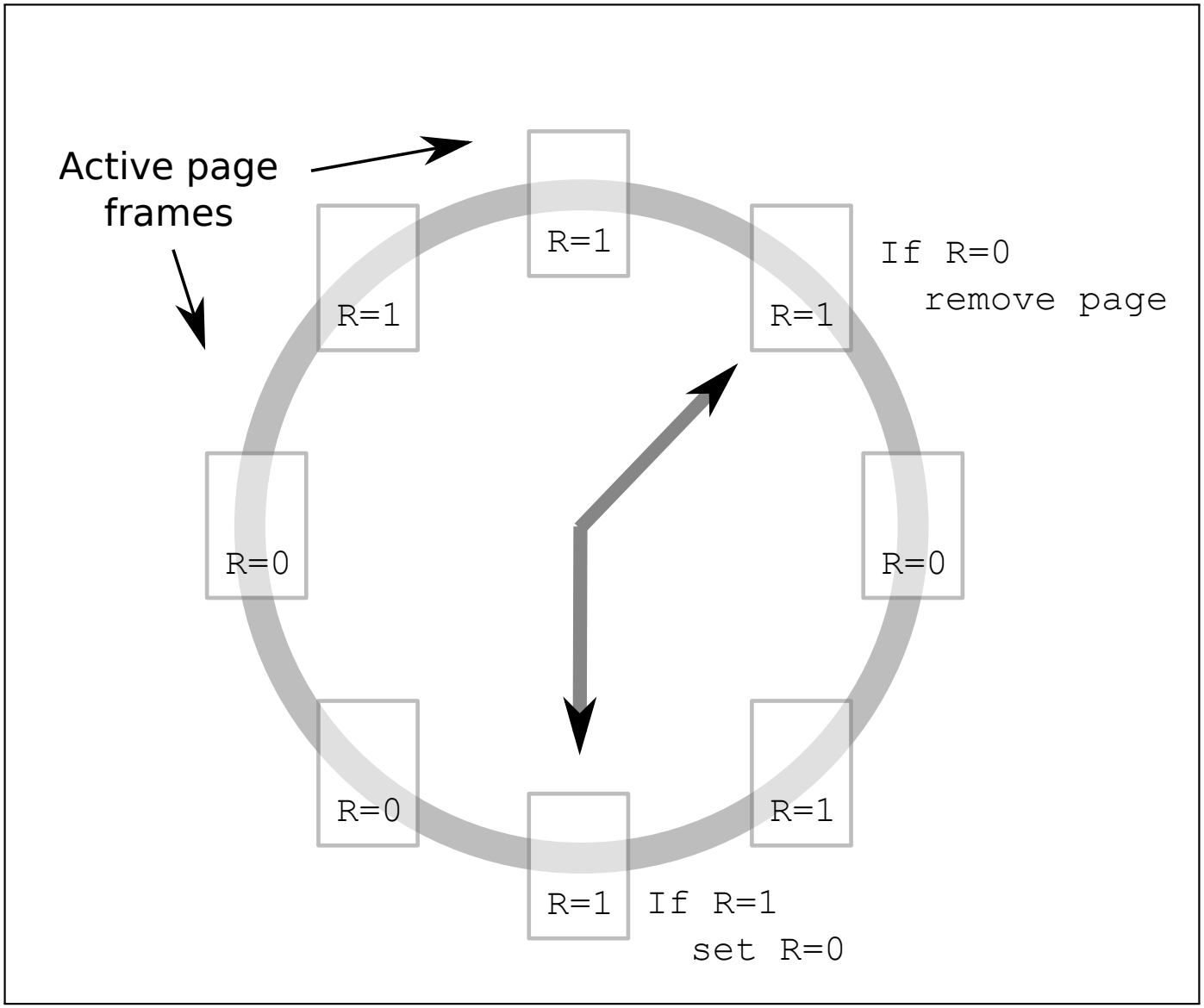
Accessing one page at a time in this way might take too long if we have a large number of them.

Replacement policies

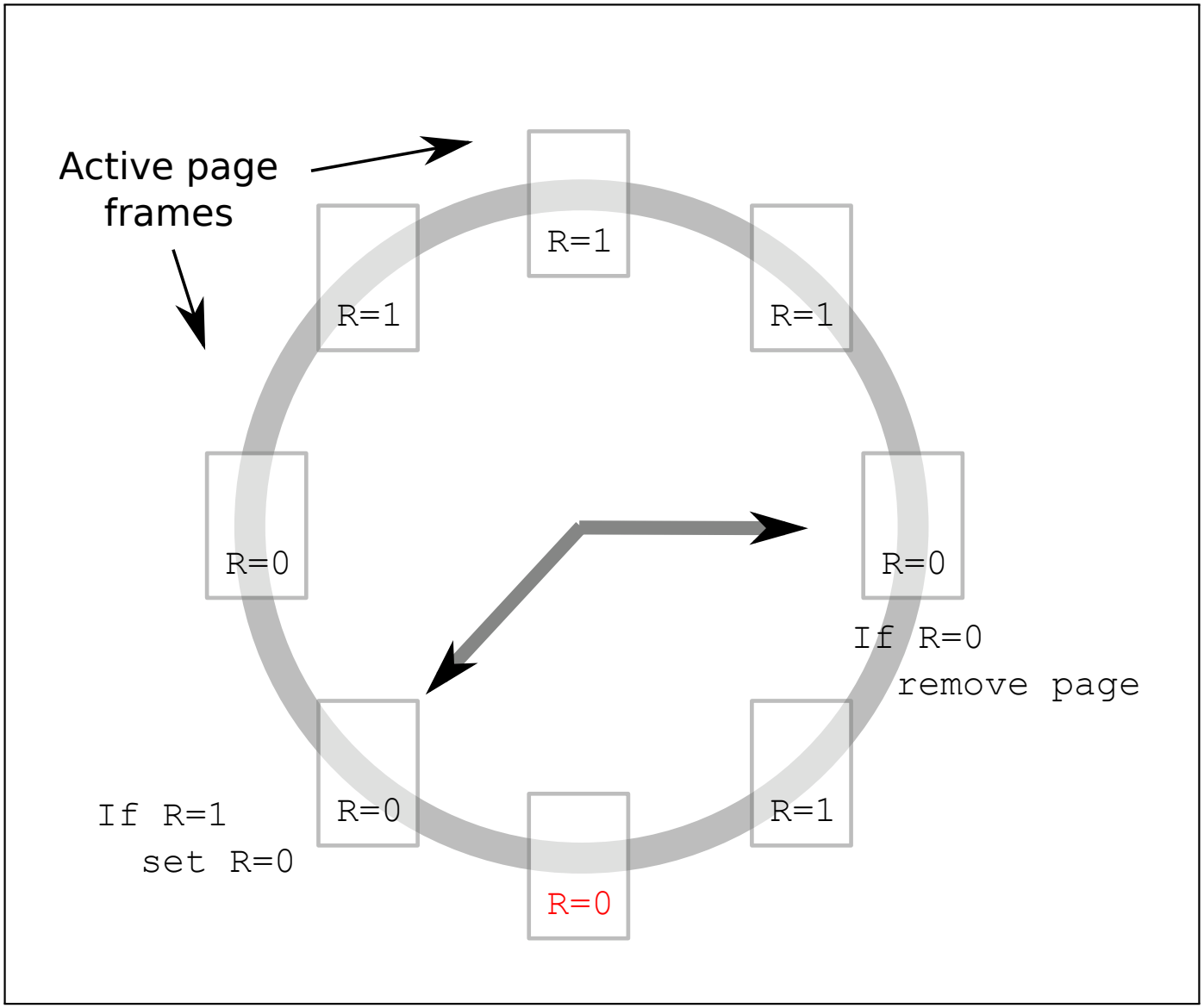
The two-handed clock algorithm was developed to solve this issue. Two threads traverse the list out of step with each other.

The first thread (or *hand*) clears the reference bits, whilst the second frees the pages whose reference bits haven't been set to 1 by the time it arrives.

Replacement policies



Replacement policies



Replacement policies