Overview of Memory Reclaim in the Current Upstream Kernel

Vlastimil Babka
Linux Kernel Developer, SUSE Labs
vbabka@suse.cz
LPC 2021, 21 September 2021 (r2)

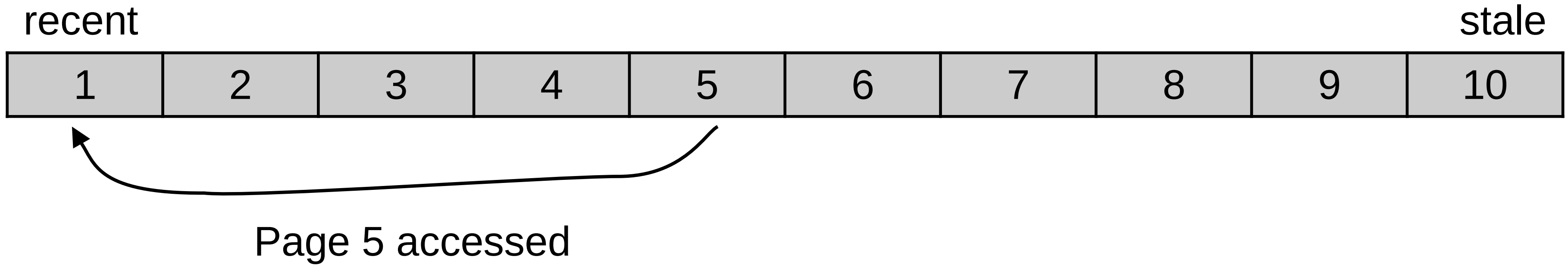
Introduction

- Unused memory is wasted memory – the kernel will keep cached everything that userspace touches, so eventually the RAM will get (almost) full
- Memory reclaim evicts the existing data to make room for new data
- Two distinct types of userspace pages
  - Anonymous pages allocated by `mmap(MAP_PRIVATE)` and populated by page fault, must be swapped out first (if at all possible) to reclaim
  - File pages (a.k.a. page cache) created by file operations or `mmap(…, fd)` – can be immediately discarded when clean, or after write-out when dirty
- Disk IO is costly, so we would like to keep pages that will be accessed again soon, and reclaim those that will not, but we cannot predict the future
  - Instead we can look at the past and assume temporal locality – pages accessed recently are more likely to be accessed again in near future
  - So we put (struct) pages on Least Recently Used (LRU) list, ordered by their last access time from most recent (head) to least recent (tail)

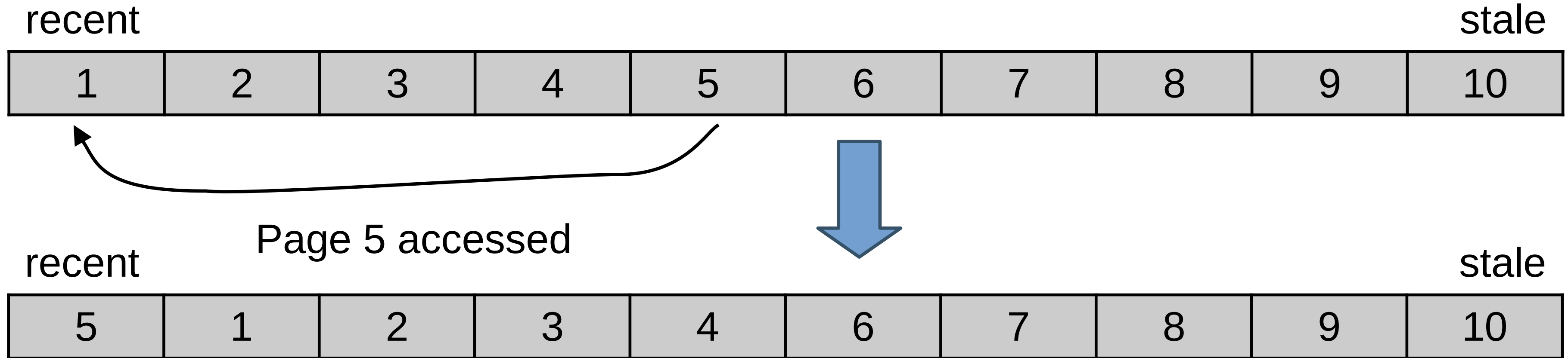
LRU list – ideal model

recent									stale
1	2	3	4	5	6	7	8	9	10

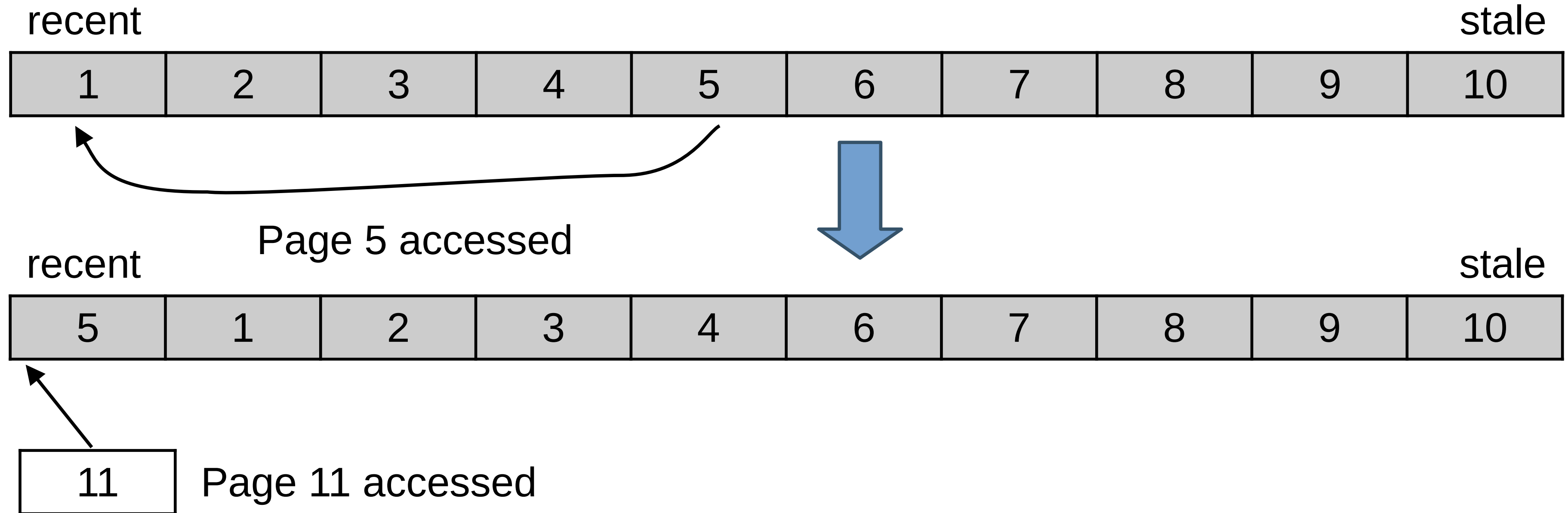
LRU list – ideal model



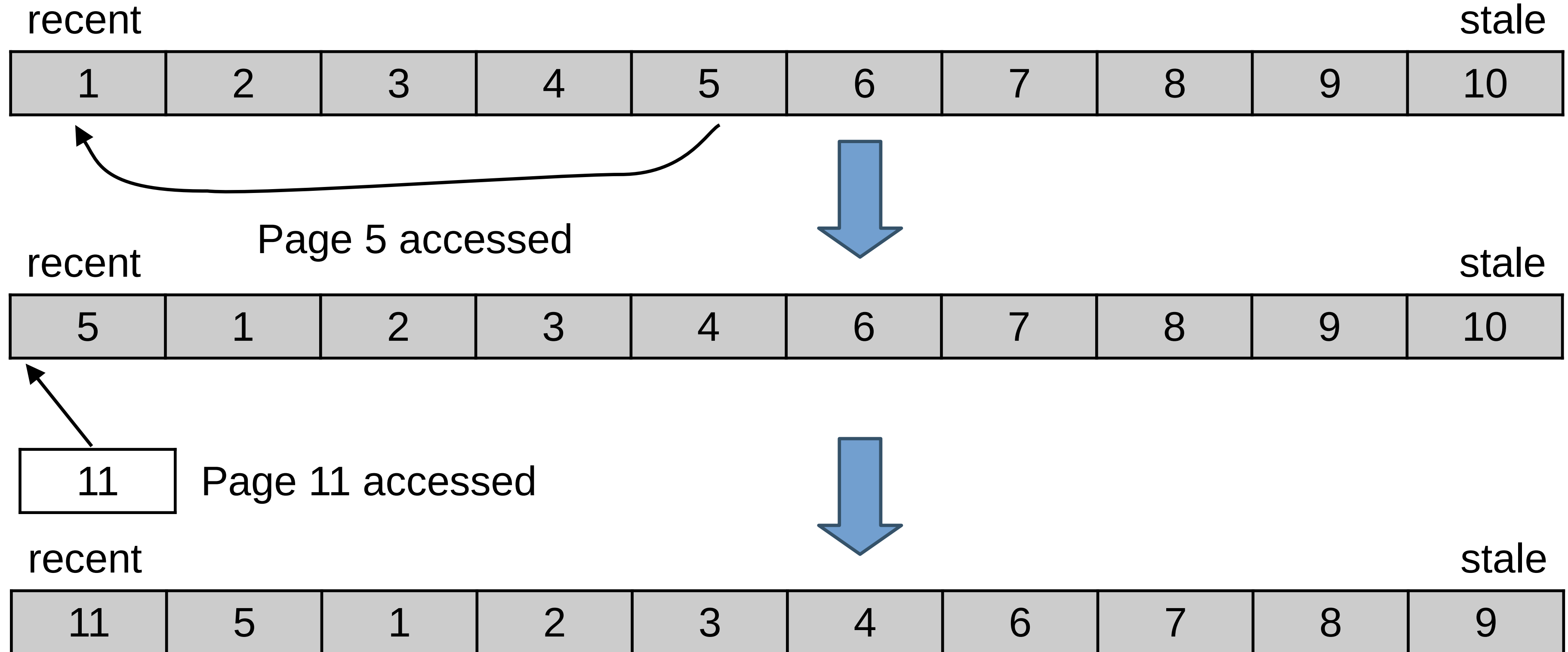
LRU list – ideal model



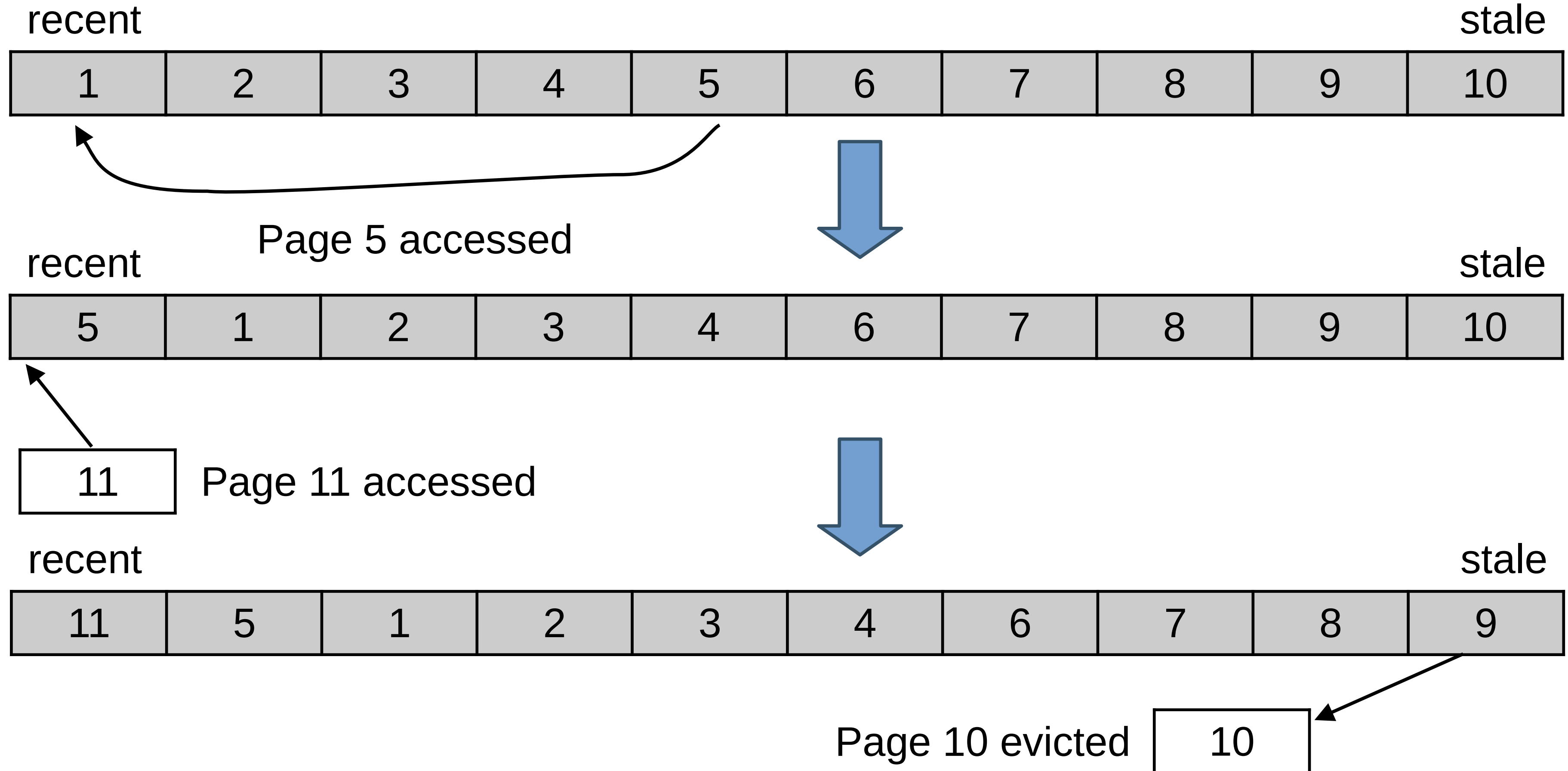
LRU list – ideal model



LRU list – ideal model



LRU list – ideal model

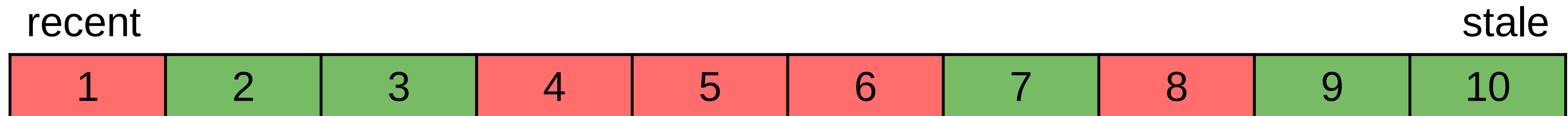


LRU – anonymous/file split

- Anonymous and file pages have distinct properties
  - Clean file pages can be just evicted, anonymous have to be swapped out at least once...
  - Historically, reclaim has been biased towards file pages more than anonymous
- Single list would be ineffective when reclaiming just one type
- Hence separate anon and file LRU lists
  - But now we **have** to choose which one (or both) to reclaim, and balance their sizes

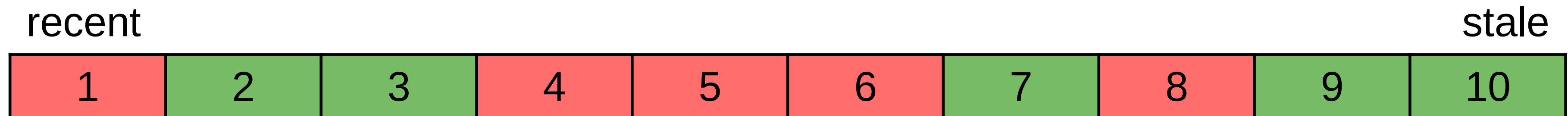
LRU – anonymous/file split

- Anonymous and file pages have distinct properties
  - Clean file pages can be just evicted, anonymous have to be swapped out at least once...
  - Historically, reclaim has been biased towards file pages more than anonymous
- Single list would be ineffective when reclaiming just one type
- Hence separate anon and file LRU lists
  - But now we **have** to choose which one (or both) to reclaim, and balance their sizes



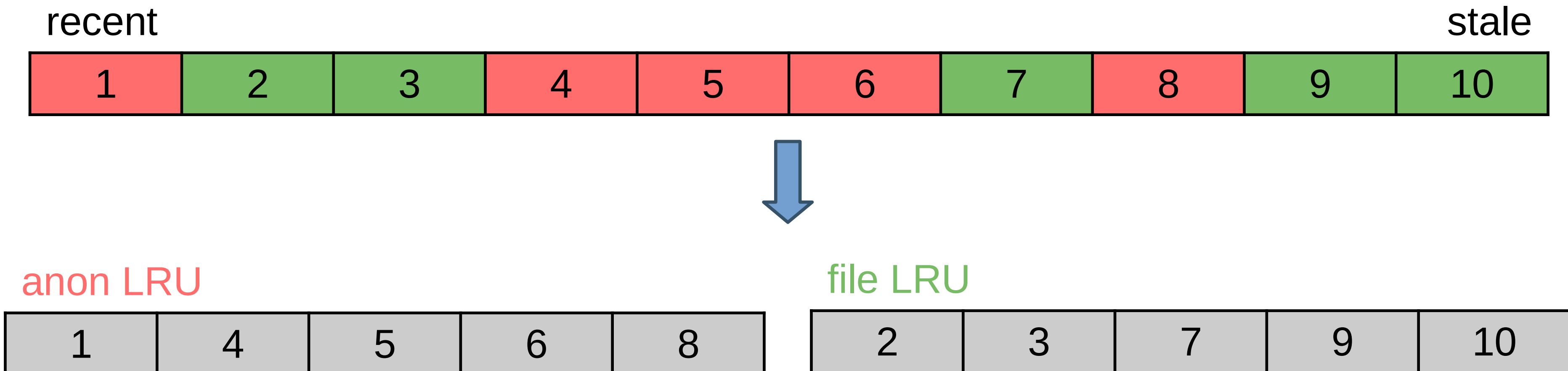
LRU – anonymous/file split

- Anonymous and file pages have distinct properties
  - Clean file pages can be just evicted, anonymous have to be swapped out at least once...
  - Historically, reclaim has been biased towards file pages more than anonymous
- Single list would be ineffective when reclaiming just one type
- Hence separate anon and file LRU lists
  - But now we **have** to choose which one (or both) to reclaim, and balance their sizes



LRU – anonymous/file split

- Anonymous and file pages have distinct properties
  - Clean file pages can be just evicted, anonymous have to be swapped out at least once...
  - Historically, reclaim has been biased towards file pages more than anonymous
- Single list would be ineffective when reclaiming just one type
- Hence separate anon and file LRU lists
  - But now we **have** to choose which one (or both) to reclaim, and balance their sizes



LRU – active/inactive split

- Ideal LRU model not achievable in practice
  - Capturing each memory access for precise tracking would be prohibitively slow
  - Approximated by detecting if page has been accessed since last check
  - More effective if we track more and less actively pages separately
- Hence separate active and inactive LRU lists for each type
  - Also fifth list for unevictable pages (not relevant to reclaim)
  - All together that’s called `lruvec`

LRU – active/inactive split

- Ideal LRU model not achievable in practice
  - Capturing each memory access for precise tracking would be prohibitively slow
  - Approximated by detecting if page has been accessed since last check
  - More effective if we track more and less actively pages separately
- Hence separate active and inactive LRU lists for each type
  - Also fifth list for unevictable pages (not relevant to reclaim)
  - All together that's called `lruvec`

anon LRU

1	4	5	6	8

file LRU

2	3	7	9	10

LRU – active/inactive split

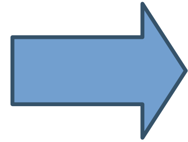
- Ideal LRU model not achievable in practice
  - Capturing each memory access for precise tracking would be prohibitively slow
  - Approximated by detecting if page has been accessed since last check
  - More effective if we track more and less actively pages separately
- Hence separate active and inactive LRU lists for each type
  - Also fifth list for unevictable pages (not relevant to reclaim)
  - All together that’s called `lruvec`

anon LRU

1	4	5	6	8

file LRU

2	3	7	9	10



LRU – active/inactive split

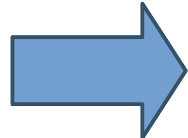
- Ideal LRU model not achievable in practice
  - Capturing each memory access for precise tracking would be prohibitively slow
  - Approximated by detecting if page has been accessed since last check
  - More effective if we track more and less actively pages separately
- Hence separate active and inactive LRU lists for each type
  - Also fifth list for unevictable pages (not relevant to reclaim)
  - All together that’s called `lruvec`

anon LRU

1	4	5	6	8

file LRU

2	3	7	9	10



LRU – node/memcg lruvecs

- Four reclaimable LRU lists per lruvec
  - Large part of reclaim magic is to decide how many pages to scan and try to reclaim in each one (*shrink* the list)
    - Pages are taken from the tail of each list, can be moved to the head of another list (activated/deactivated), back to head of the same list (kept), or evicted entirely (reclaimed)

LRU – node/memcg lruvecs

- Four reclaimable LRU lists per lruvec
  - Large part of reclaim magic is to decide how many pages to scan and try to reclaim in each one (*shrink* the list)
    - Pages are taken from the tail of each list, can be moved to the head of another list (activated/deactivated), back to head of the same list (kept), or evicted entirely (reclaimed)
- In practice, there are many lruvecs
  - Different memory cgroups have distinct lruvecs, for memcg reclaim
    - Global memory reclaim has to iterate over all memcgs
  - Different NUMA nodes have distinct lruvecs, as nodes are reclaimed separately
    - Each node has own kswapd daemon, memory pressure can differ due to e.g. mempolicies

LRU – node/memcg lruvecs

- Four reclaimable LRU lists per lruvec
  - Large part of reclaim magic is to decide how many pages to scan and try to reclaim in each one (*shrink* the list)
    - Pages are taken from the tail of each list, can be moved to the head of another list (activated/deactivated), back to head of the same list (kept), or evicted entirely (reclaimed)
- In practice, there are many lruvecs
  - Different memory cgroups have distinct lruvecs, for memcg reclaim
    - Global memory reclaim has to iterate over all memcgs
  - Different NUMA nodes have distinct lruvecs, as nodes are reclaimed separately
    - Each node has own kswapd daemon, memory pressure can differ due to e.g. mempolicies
- Summary: each userspace page placed on a LRU list in one of many lruvecs:

LRU – node/memcg lruvecs

- Four reclaimable LRU lists per lruvec
  - Large part of reclaim magic is to decide how many pages to scan and try to reclaim in each one (*shrink* the list)
    - Pages are taken from the tail of each list, can be moved to the head of another list (activated/deactivated), back to head of the same list (kept), or evicted entirely (reclaimed)
- In practice, there are many lruvecs
  - Different memory cgroups have distinct lruvecs, for memcg reclaim
    - Global memory reclaim has to iterate over all memcgs
  - Different NUMA nodes have distinct lruvecs, as nodes are reclaimed separately
    - Each node has own kswapd daemon, memory pressure can differ due to e.g. mempolicies
- Summary: each userspace page placed on a LRU list in one of many lruvecs:

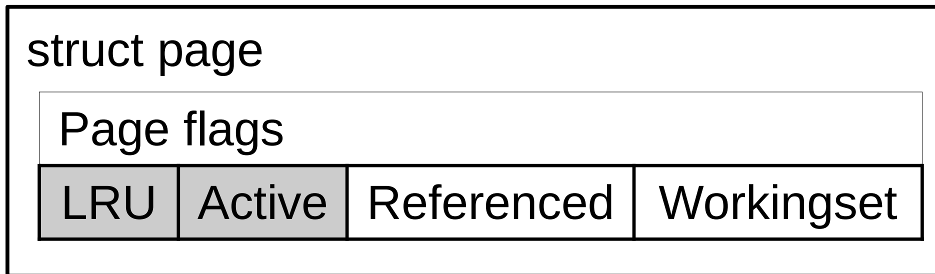
	Root memcg	Memcg1	Memcg2	Memcg3	Memcg4	Memcg5
Node 0	lruvec	lruvec	lruvec	lruvec	lruvec	lruvec
Node 1	lruvec	lruvec	lruvec	lruvec	lruvec	lruvec

Page States Relevant to Reclaim

- Determined by page flags, mainly the following:
  - LRU – page is on any LRU list, Active – page is on active list
  - Referenced – inactive page has been accessed “recently”
  - Workingset – page is considered part of active userspace’s workingset
- Affected by Accessed bit in page tables entries (PTE’s) that map this page
  - `page_referenced()` counts them (via a *rmap walk*) and resets them to zero

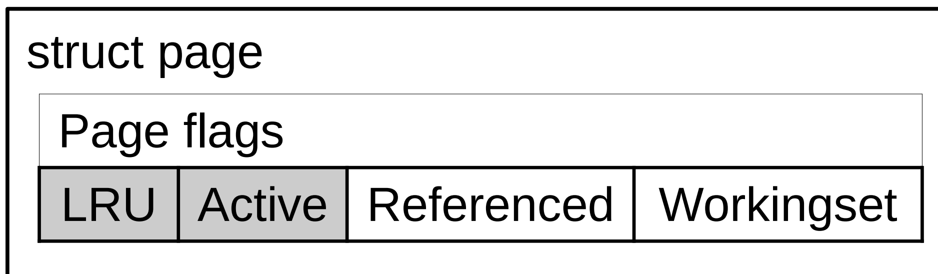
Page States Relevant to Reclaim

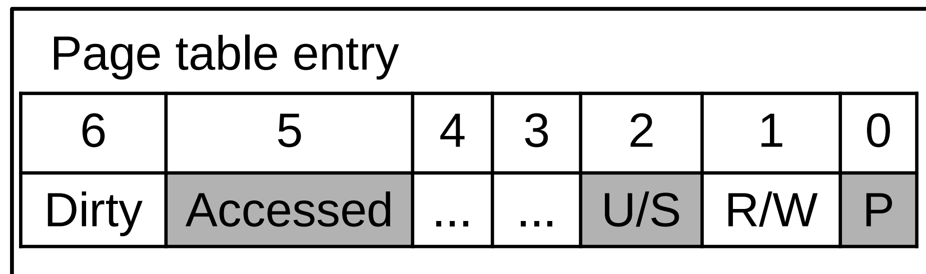
- Determined by page flags, mainly the following:
  - LRU – page is on any LRU list, Active – page is on active list
  - Referenced – inactive page has been accessed “recently”
  - Workingset – page is considered part of active userspace’s workingset
- Affected by Accessed bit in page tables entries (PTE’s) that map this page
  - `page_referenced()` counts them (via a *rmap walk*) and resets them to zero



Page States Relevant to Reclaim

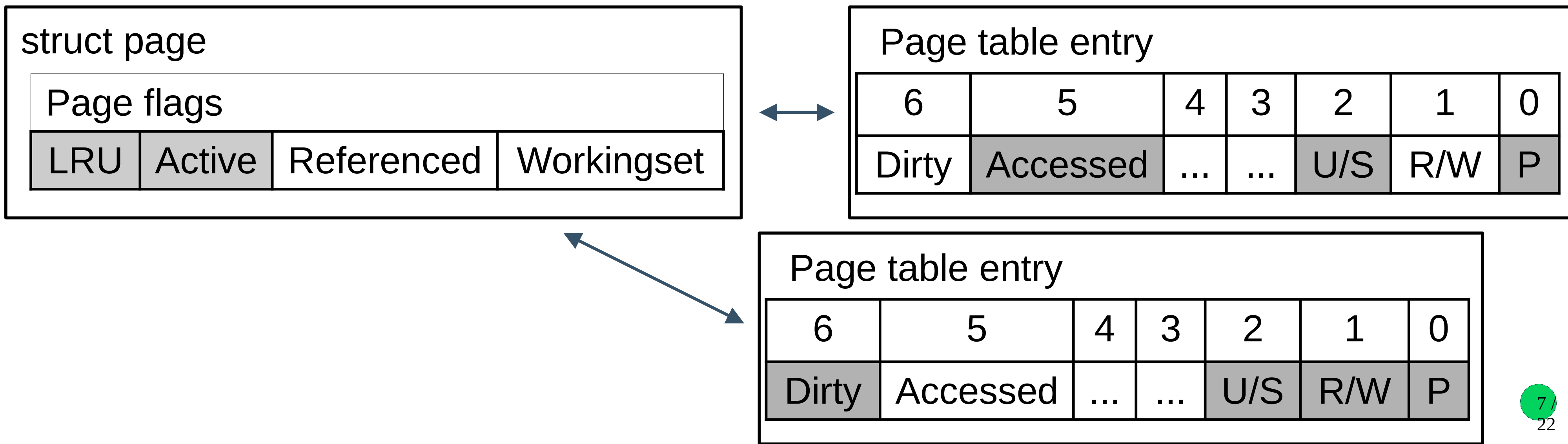
- Determined by page flags, mainly the following:
  - LRU – page is on any LRU list, Active – page is on active list
  - Referenced – inactive page has been accessed “recently”
  - Workingset – page is considered part of active userspace’s workingset
- Affected by Accessed bit in page tables entries (PTE’s) that map this page
  - `page_referenced()` counts them (via a *rmap walk*) and resets them to zero





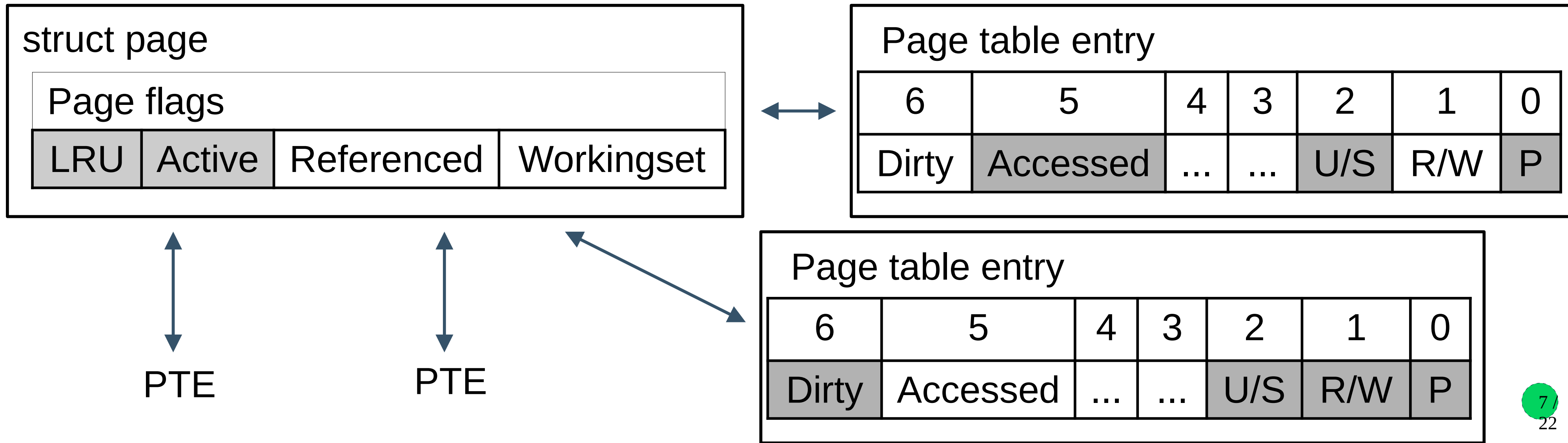
Page States Relevant to Reclaim

- Determined by page flags, mainly the following:
  - LRU – page is on any LRU list, Active – page is on active list
  - Referenced – inactive page has been accessed “recently”
  - Workingset – page is considered part of active userspace’s workingset
- Affected by Accessed bit in page tables entries (PTE’s) that map this page
  - `page_referenced()` counts them (via a *rmap walk*) and resets them to zero



Page States Relevant to Reclaim

- Determined by page flags, mainly the following:
  - LRU – page is on any LRU list, Active – page is on active list
  - Referenced – inactive page has been accessed “recently”
  - Workingset – page is considered part of active userspace’s workingset
- Affected by Accessed bit in page tables entries (PTE’s) that map this page
  - `page_referenced()` counts them (via a *rmap walk*) and resets them to zero



Not present

Not present

initial page fault

!active

!referenced

#PTE.A=1

After fault is handled,
the userspace access
is restarted and sets
PTE Accessed
bit immediately

kern/usr
access

Not present

initial page fault

!active
!referenced
#PTE.A=1

kern/usr access

Not present

initial page fault

!active

referenced

#PTE.A=0

!active

!referenced

#PTE.A=1

Reclaim filters out the initial access by only setting the referenced Page flag, but keeping Page on inactive list

kern/usr access

reclaim keeps

Not present

initial page fault

!active
referenced
#PTE.A=0

!active
!referenced
#PTE.A=1

kern/usr access

reclaim keeps

Not present

initial page fault

!active

!referenced

#PTE.A=1

!active

referenced

#PTE.A=0

userspace

!active

referenced

#PTE.A>0

Another access sets PTE active bit

kern/usr access

reclaim keeps

Not present

initial page fault

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

userspace

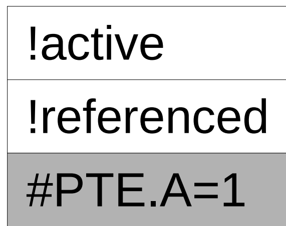
!active
referenced
#PTE.A>0

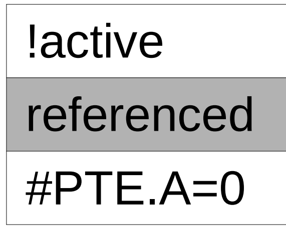
kern/usr
access

reclaim
keeps

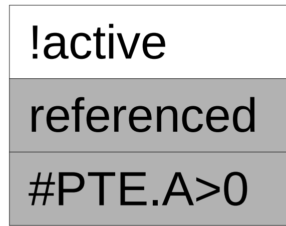
Not present

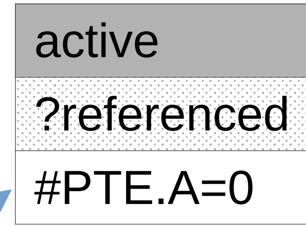
initial page fault





userspace





Reclaim sees both referenced flag and PTE active, so page was accessed multiple times, activate it

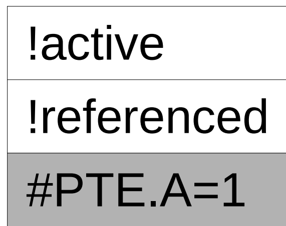
kern/usr access

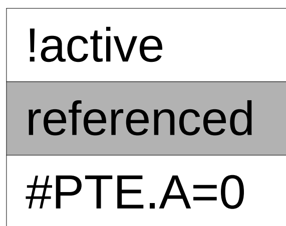
reclaim keeps

reclaim promotes

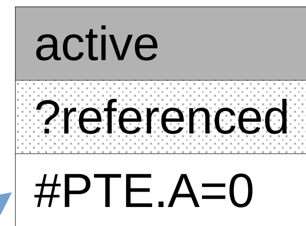
Not present

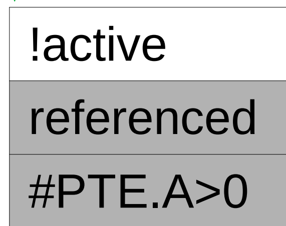
initial page fault





userspace

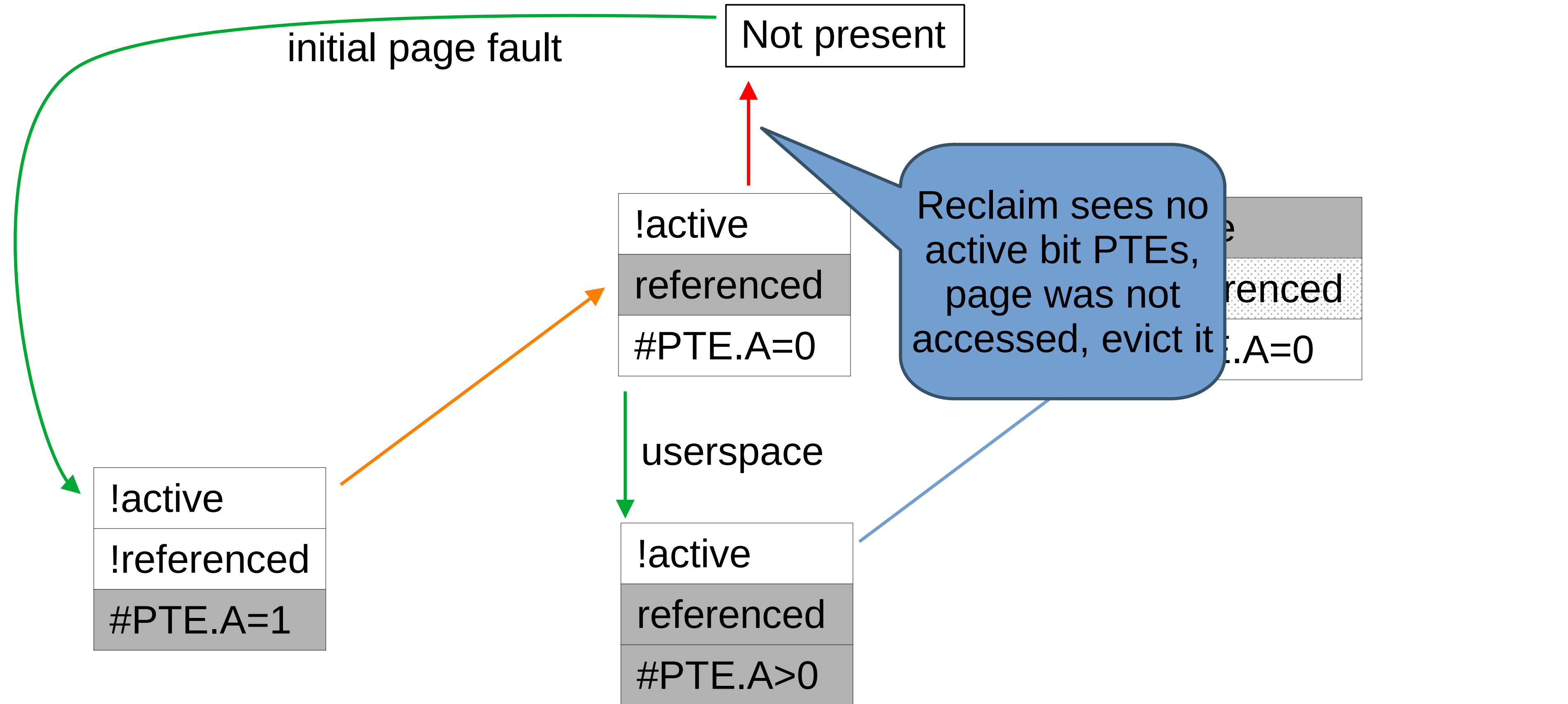




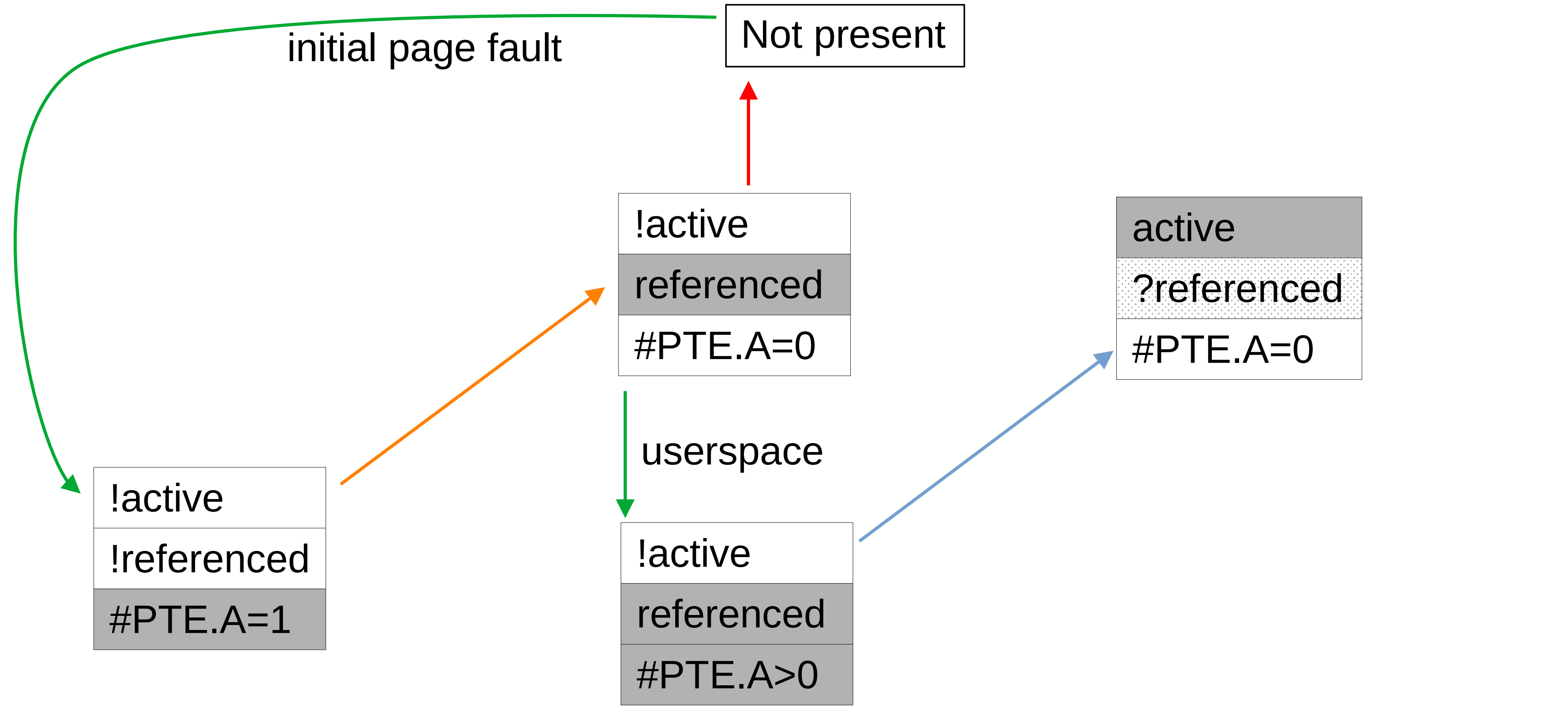
kern/usr access

reclaim keeps

reclaim promotes



kern/usr access | reclaim demotes | reclaim keeps | reclaim promotes

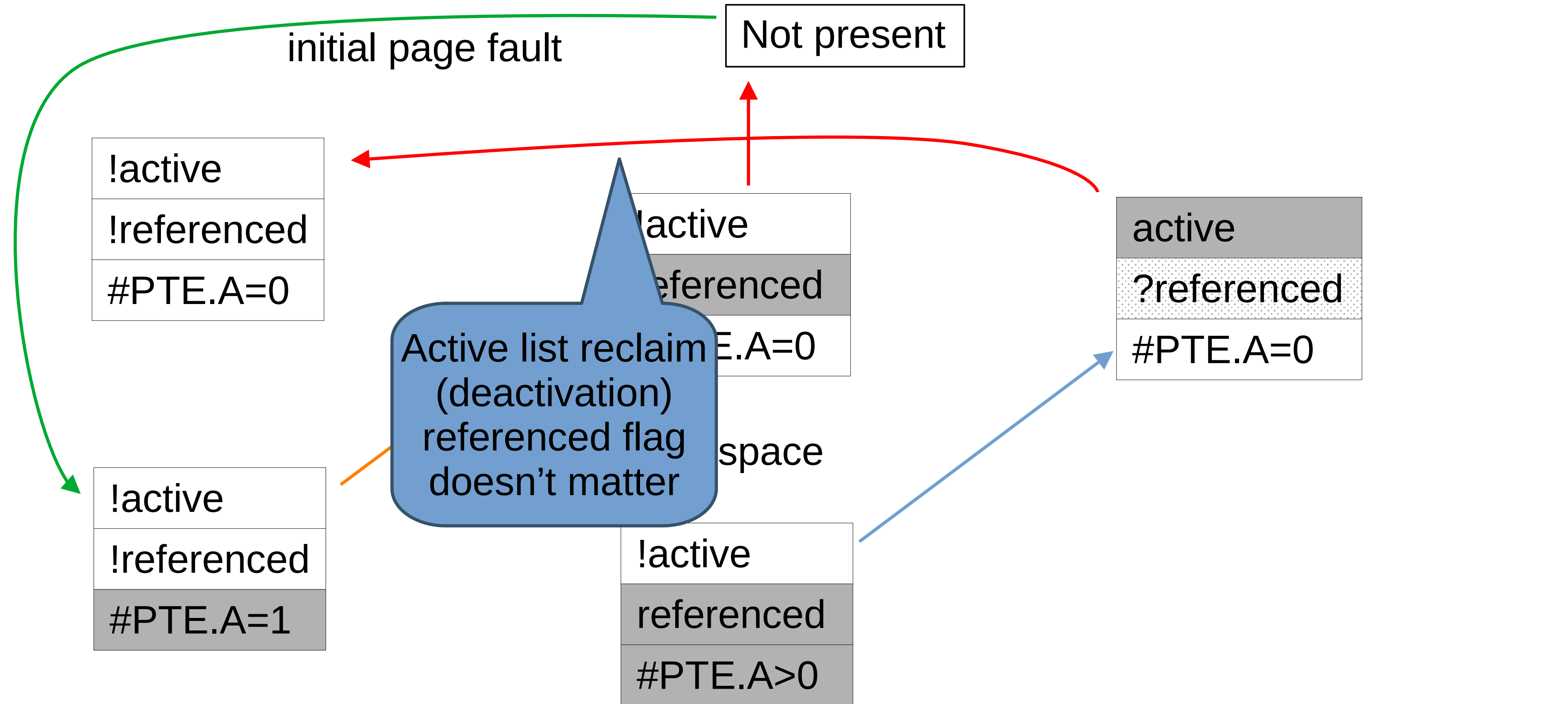


kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes

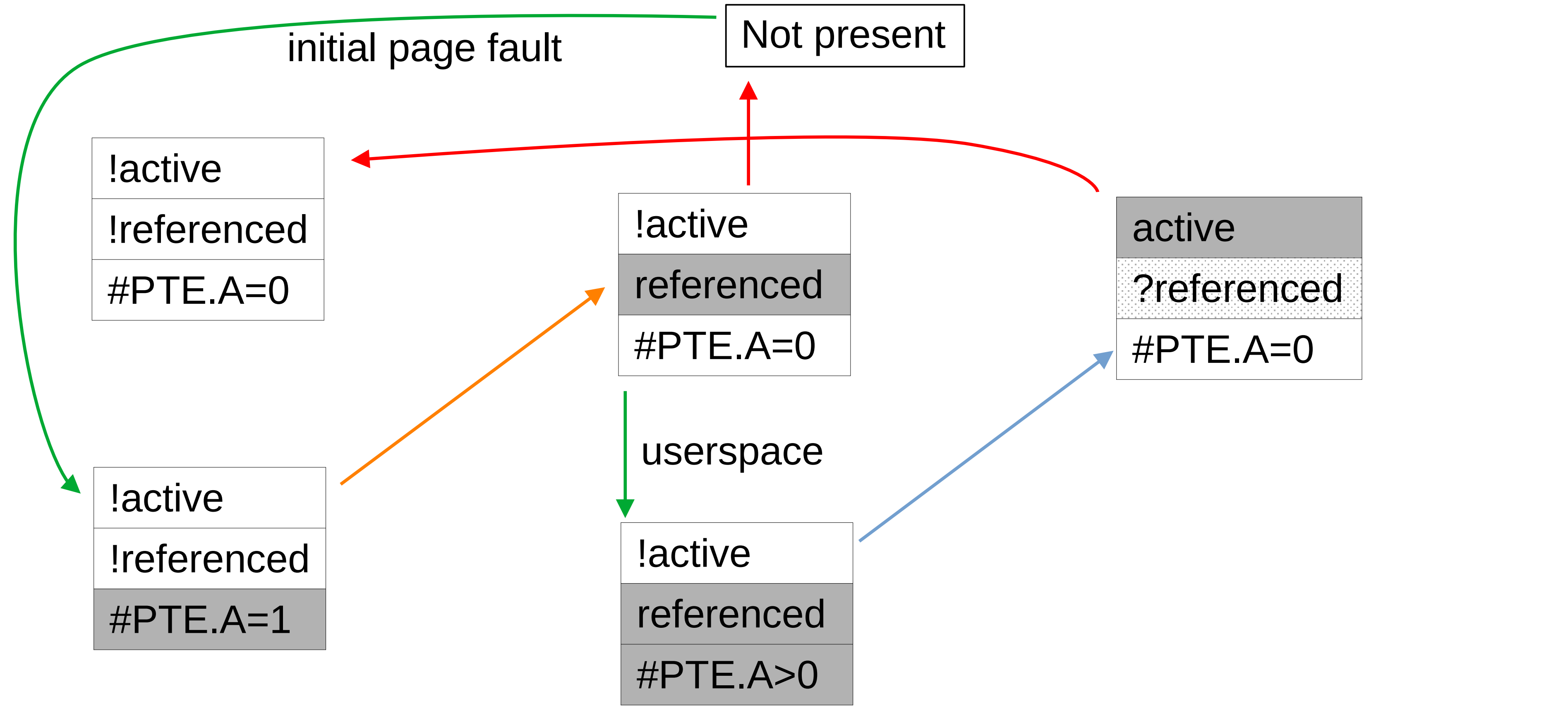


kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes

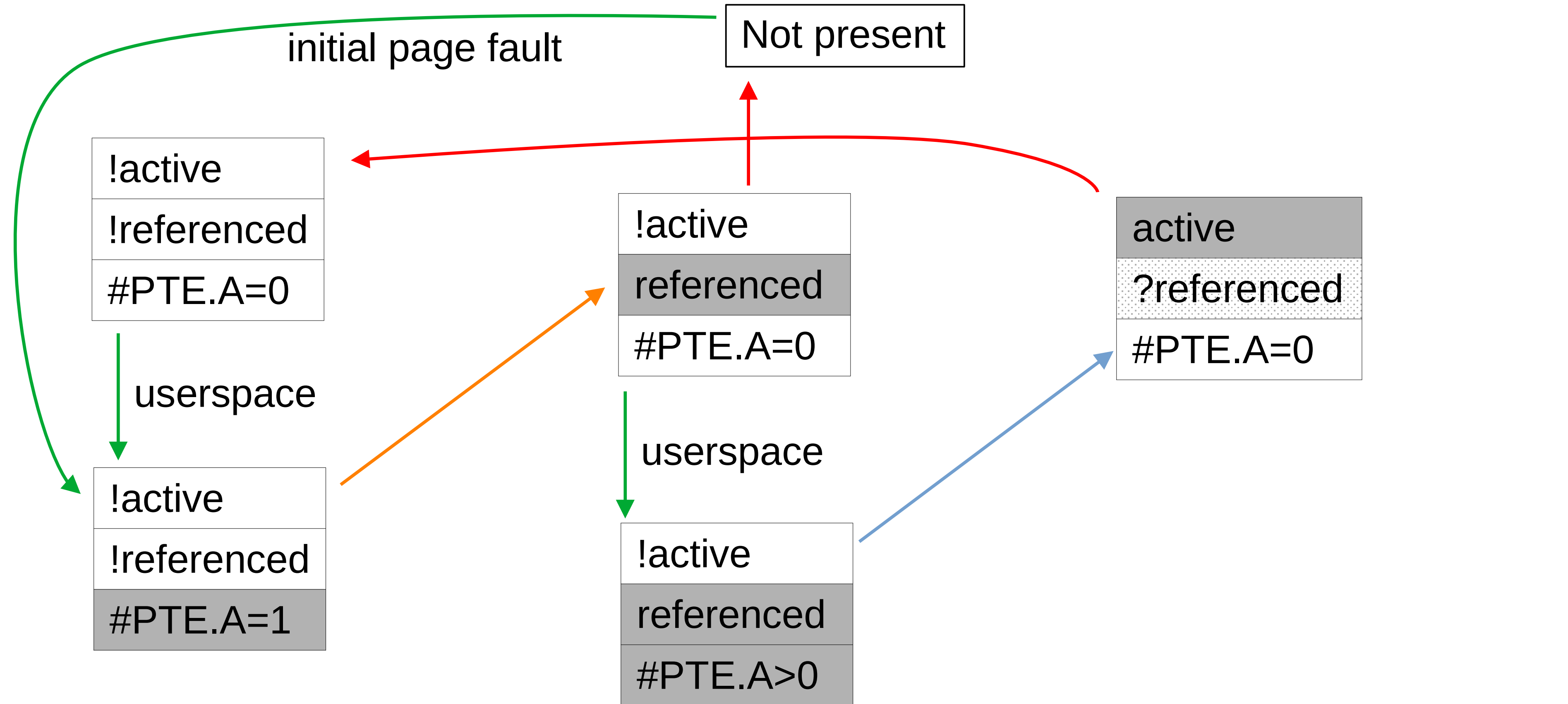


kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes

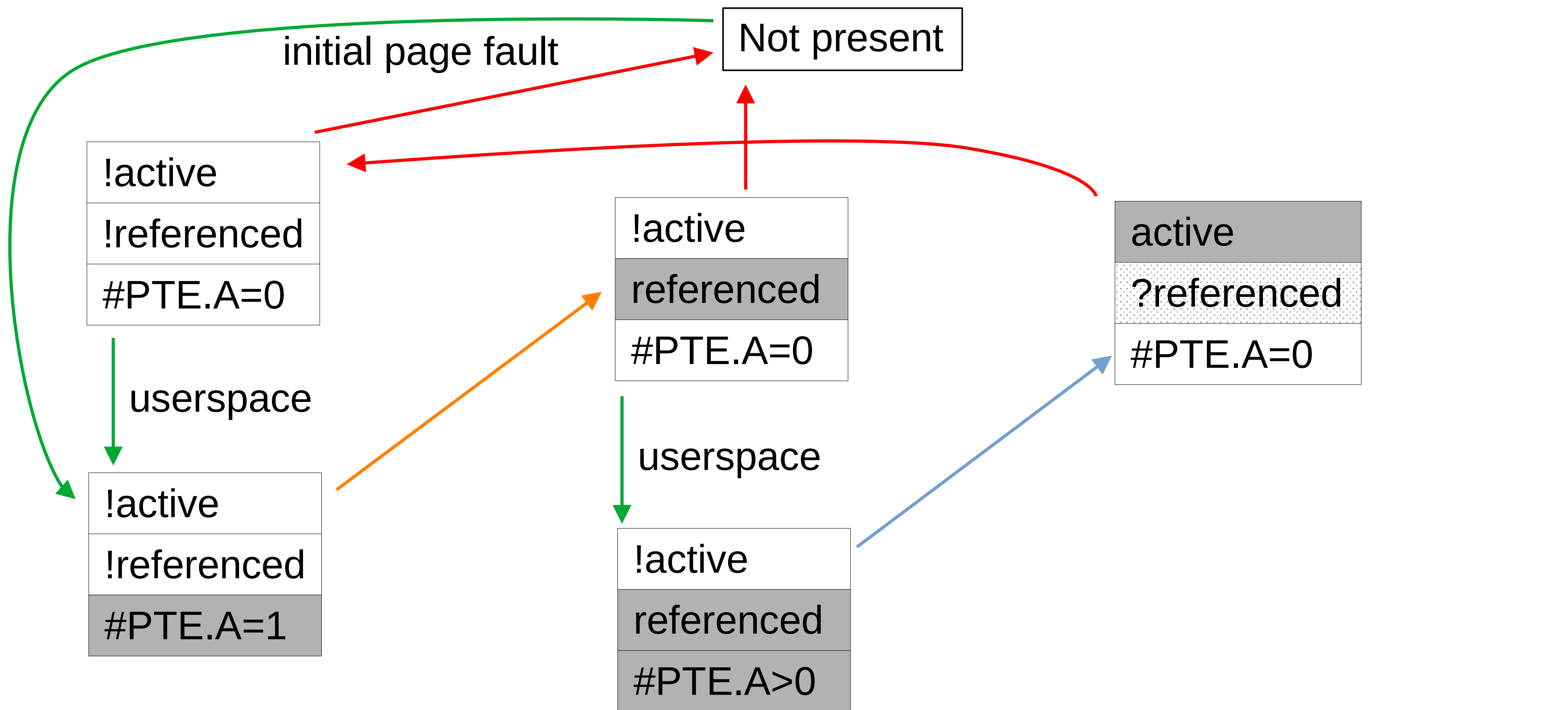


kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes



kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes

Not present

initial page fault

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

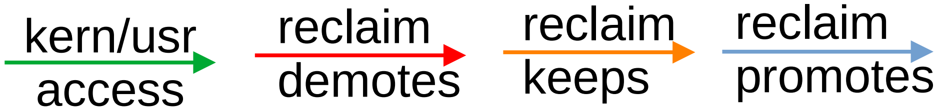
userspace

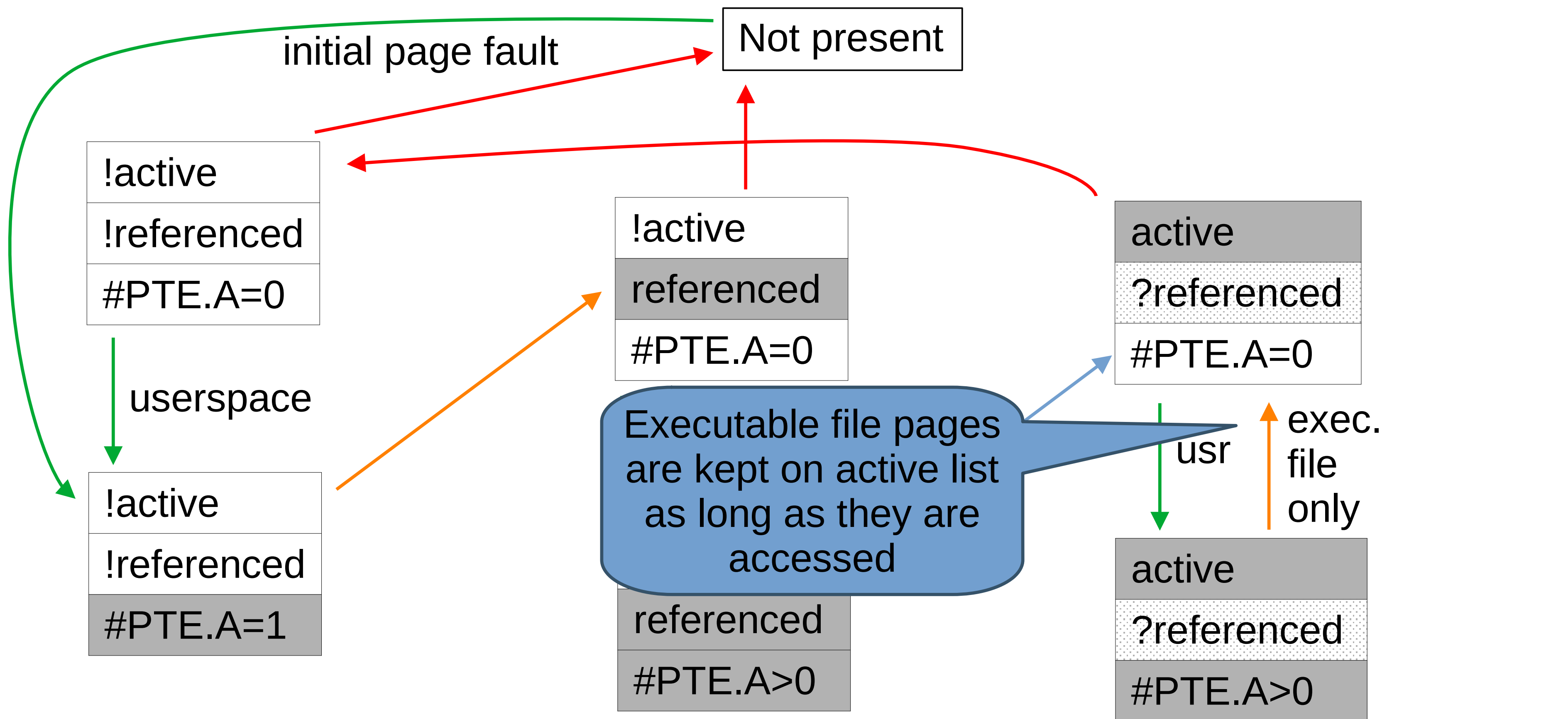
!active
referenced
#PTE.A>0

active
?referenced
#PTE.A=0

usr

active
?referenced
#PTE.A>0





kern/usr access
reclaim demotes
reclaim keeps
reclaim promotes

Not present

initial page fault

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

userspace

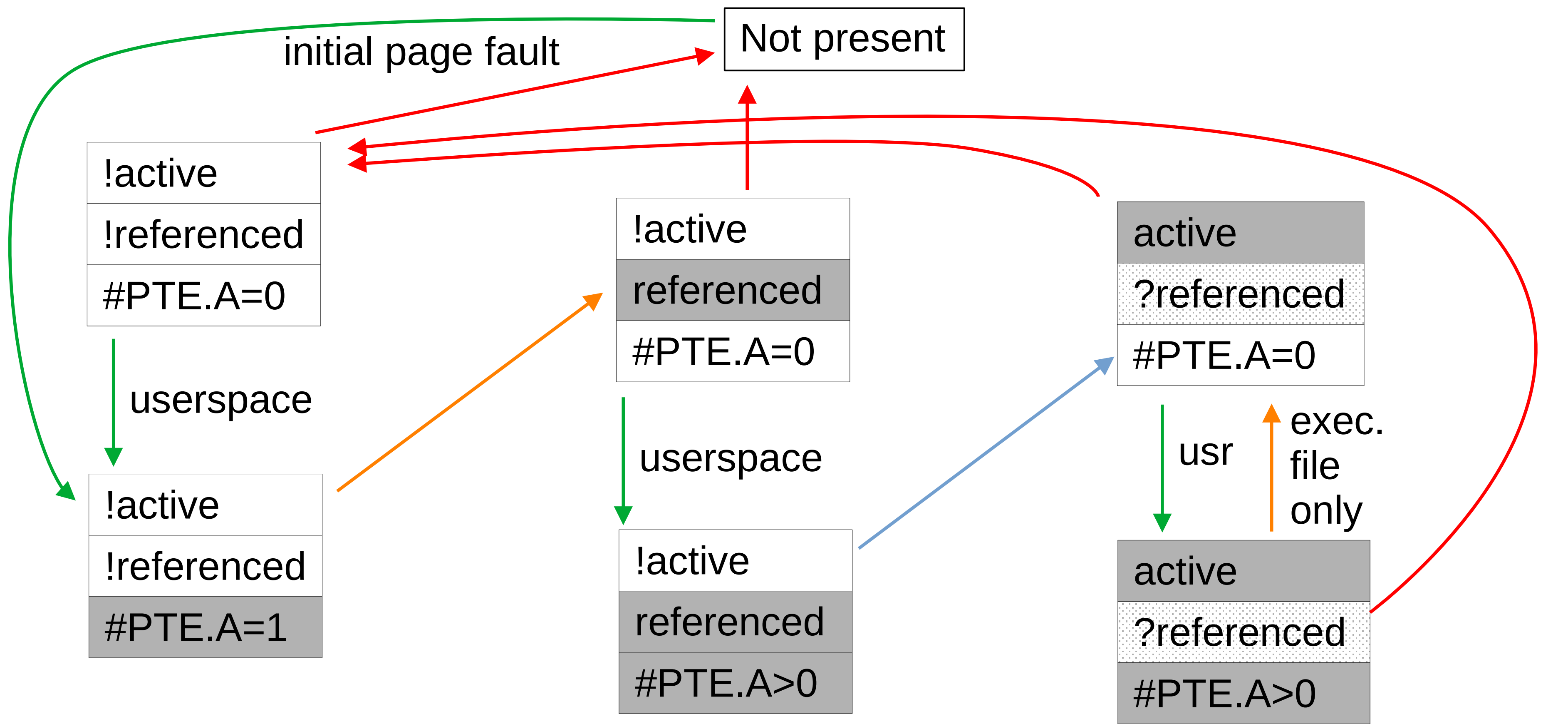
!active
referenced
#PTE.A>0

active
?referenced
#PTE.A=0

usr

exec.
file
only

active
?referenced
#PTE.A>0



Not present
initial page fault
!active
!referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
!active
referenced
#PTE.A=0
userspace
!active
referenced
#PTE.A>0
active
?referenced
#PTE.A=0
usr
exec.
file
only
active
?referenced
#PTE.A>0
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes

Not present

initial page fault

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

userspace

!active
referenced
#PTE.A>0

active
?referenced
#PTE.A=0

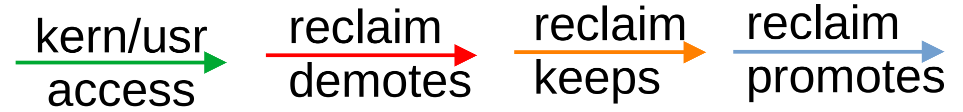
usr

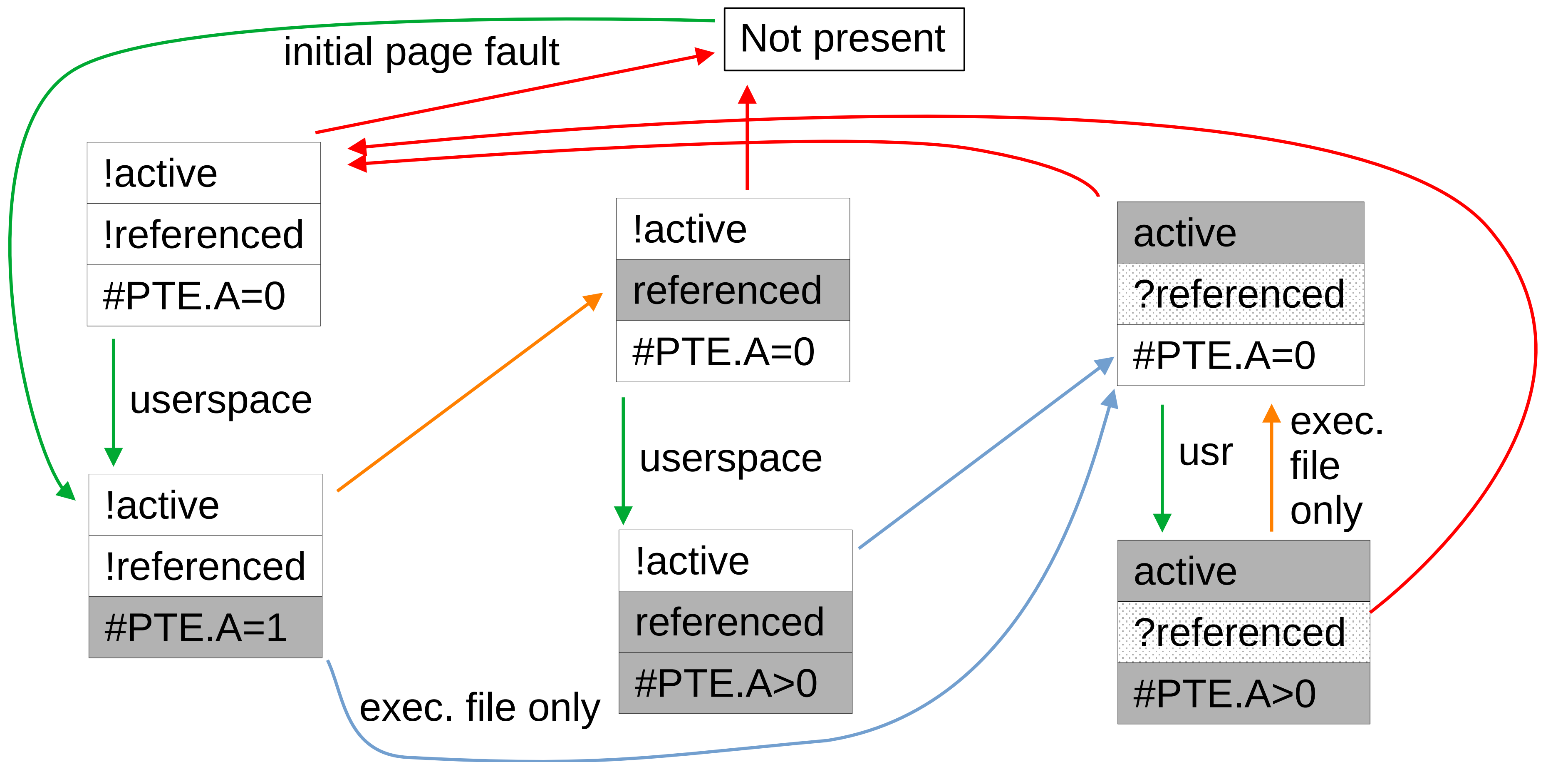
exec.
file
only

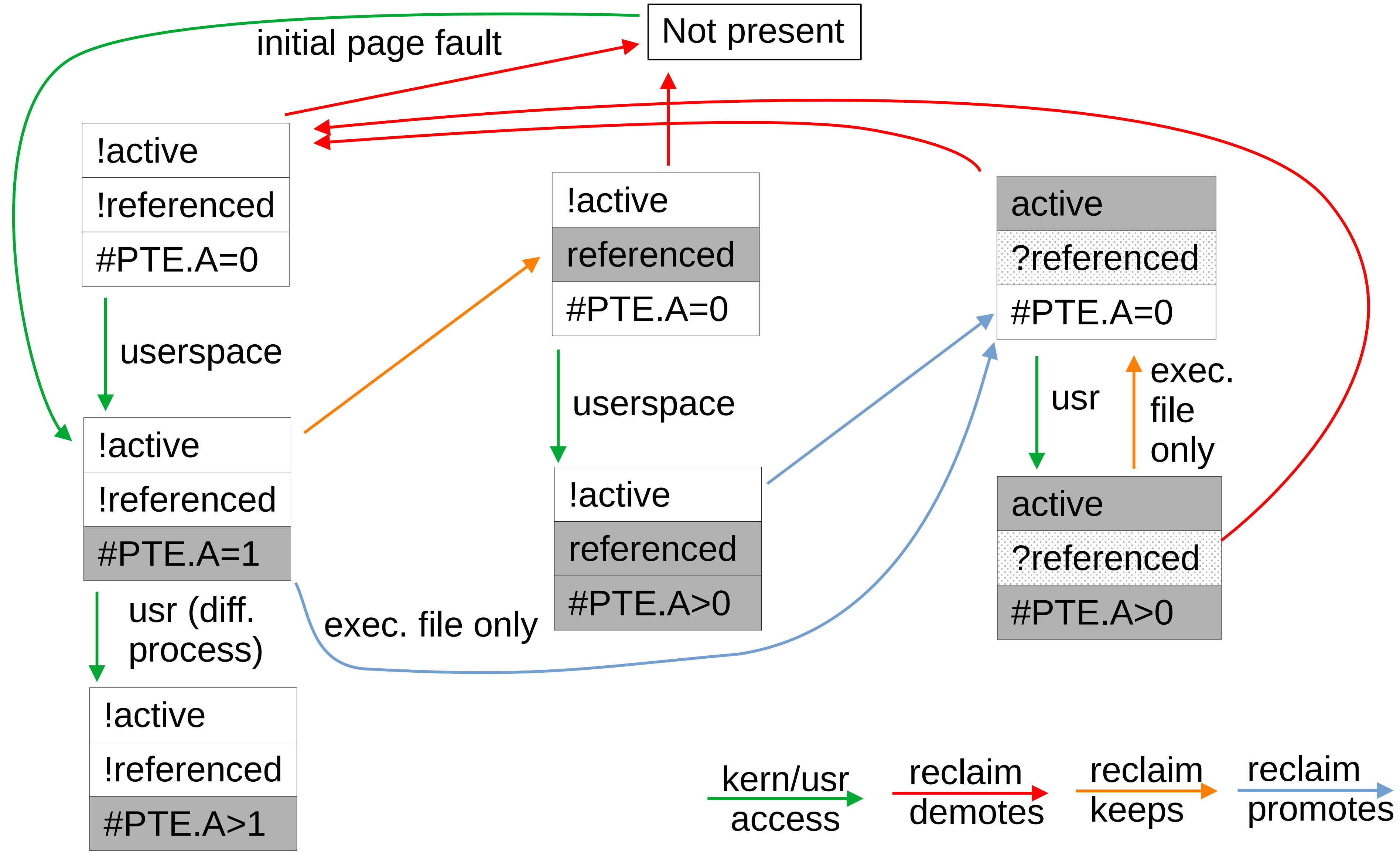
active
?referenced
#PTE.A>0

exec. file only

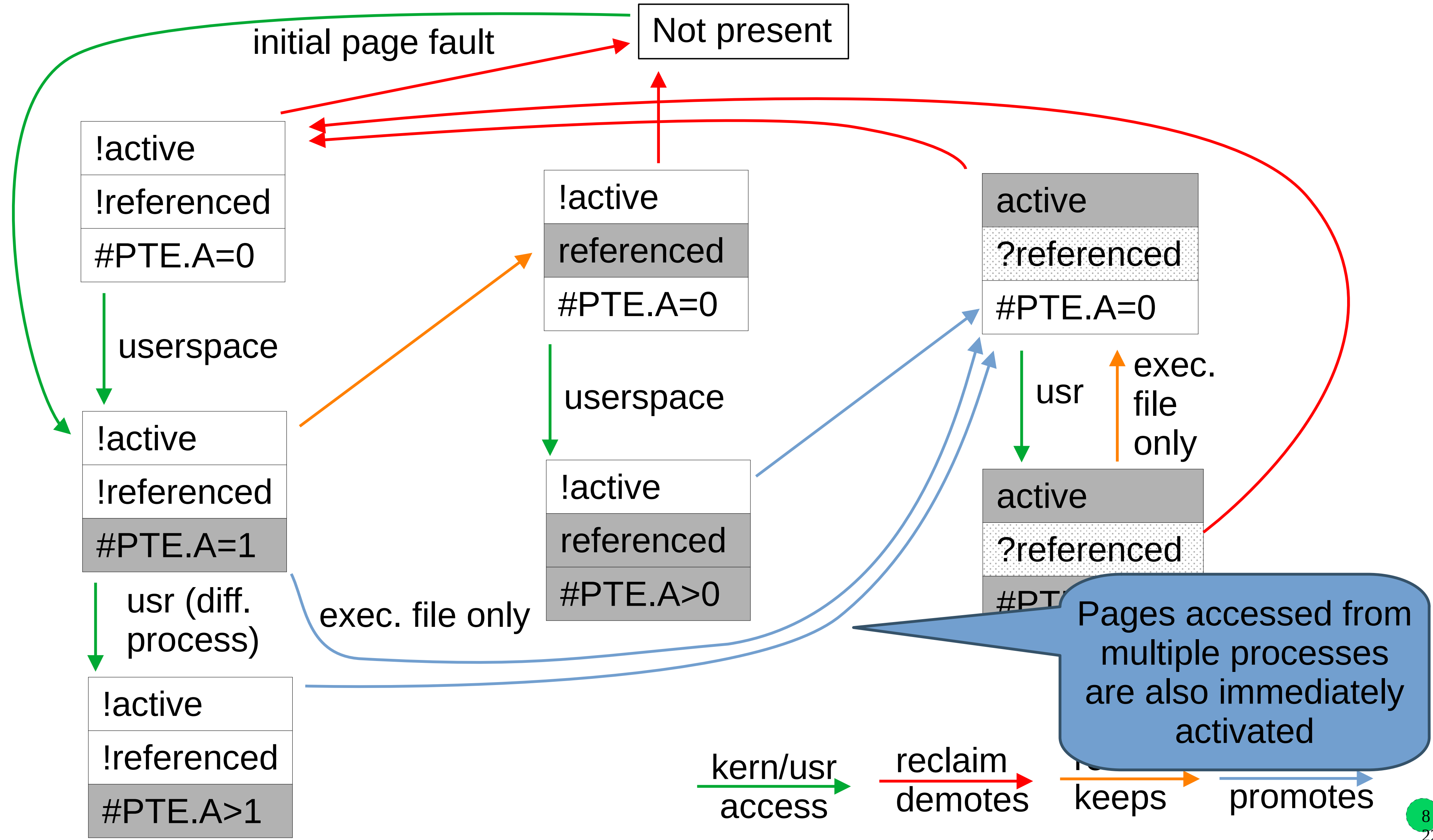
Executable file pages are also immediately activated

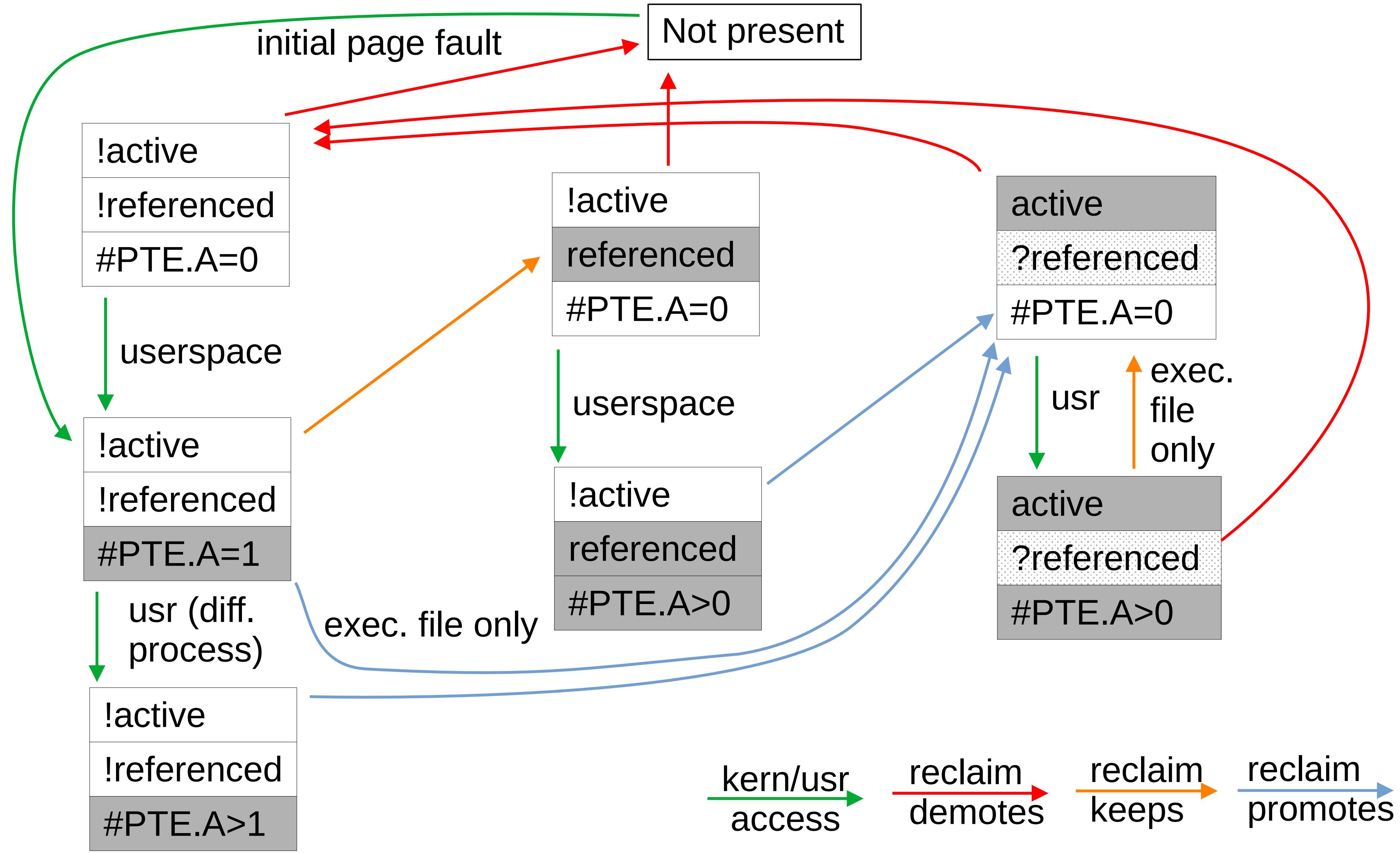


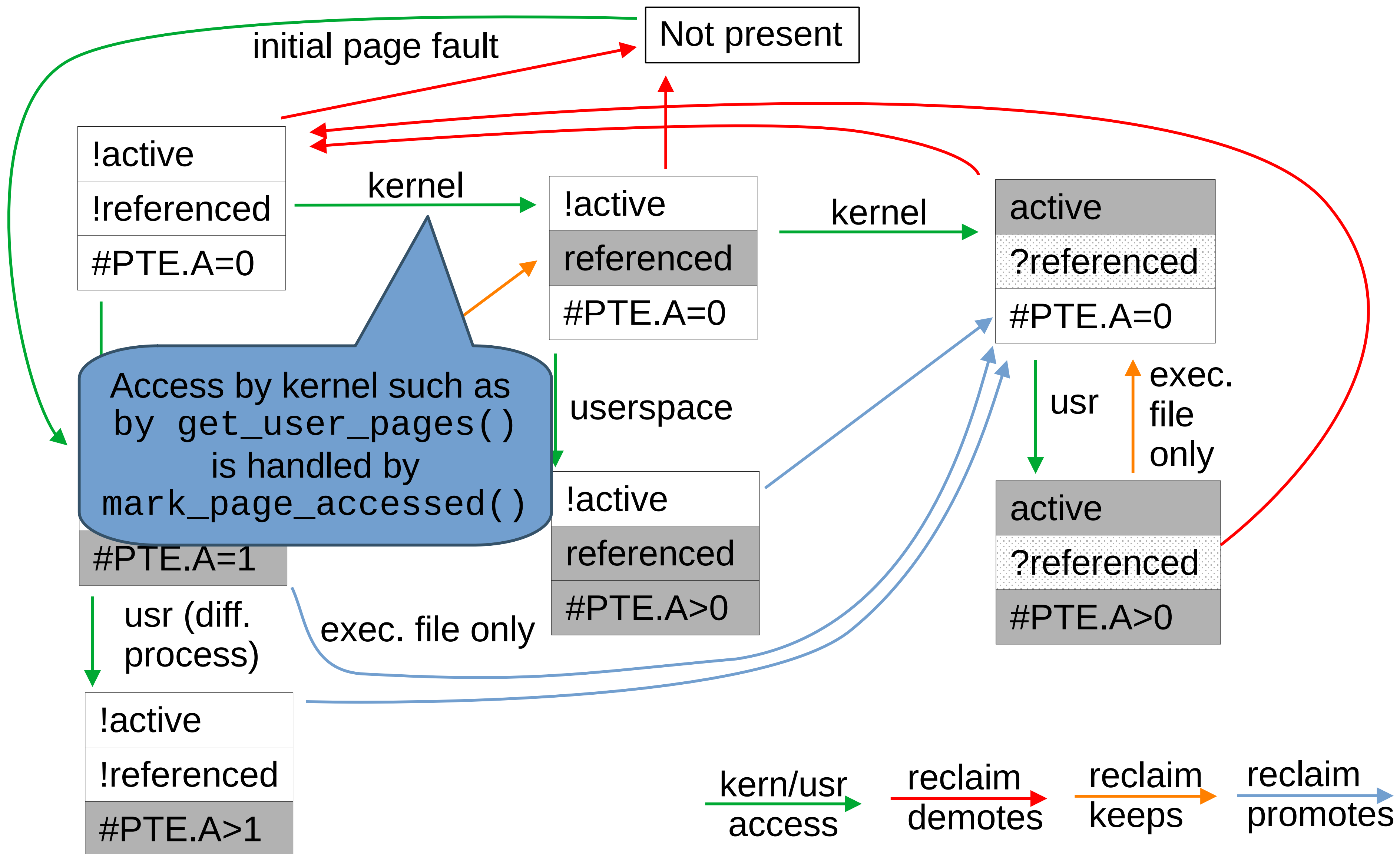

Not present
initial page fault
!active
!referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
exec. file only
!active
referenced
#PTE.A=0
userspace
!active
referenced
#PTE.A>0
active
?referenced
#PTE.A=0
usr
exec.
file
only
active
?referenced
#PTE.A>0
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes

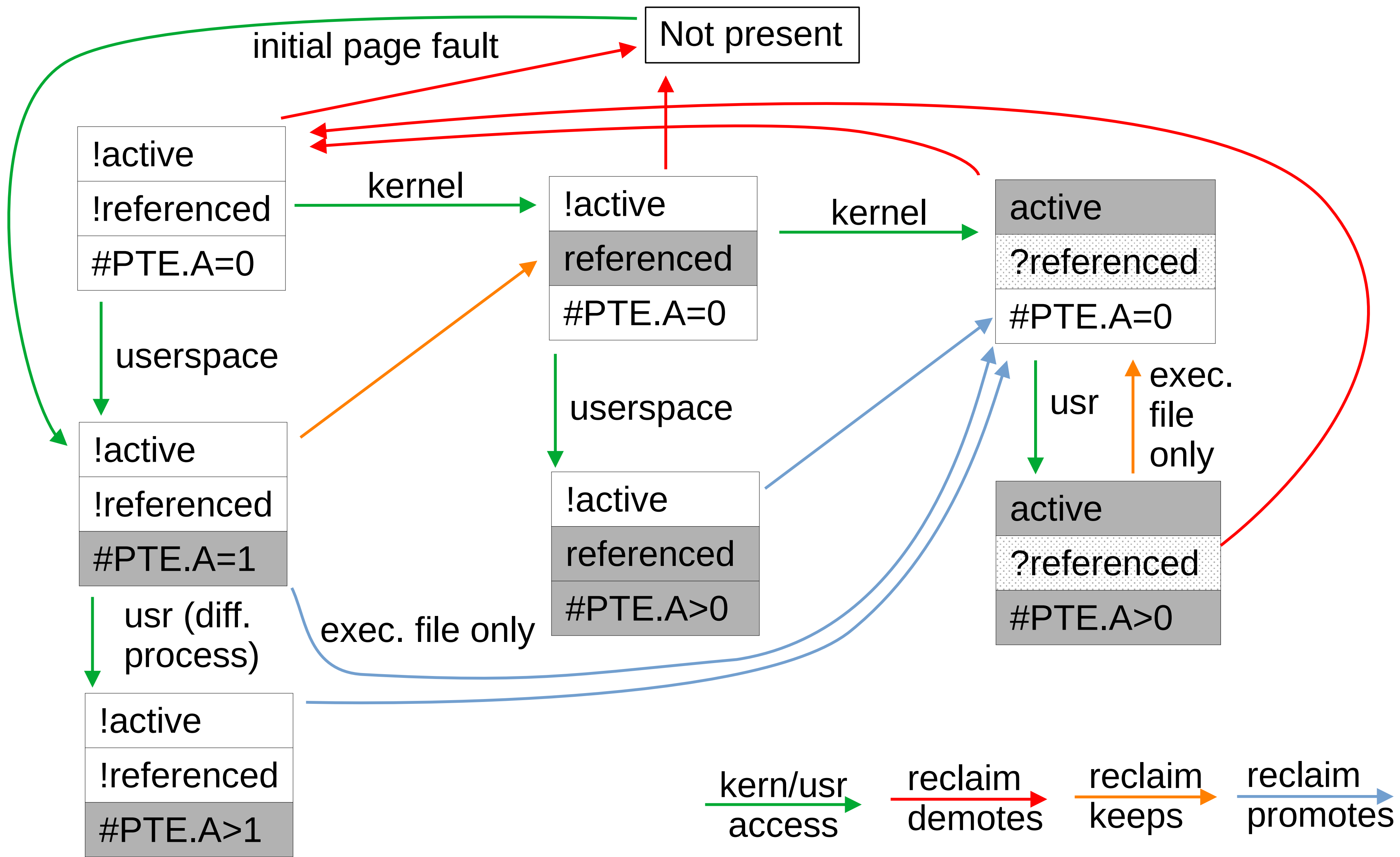


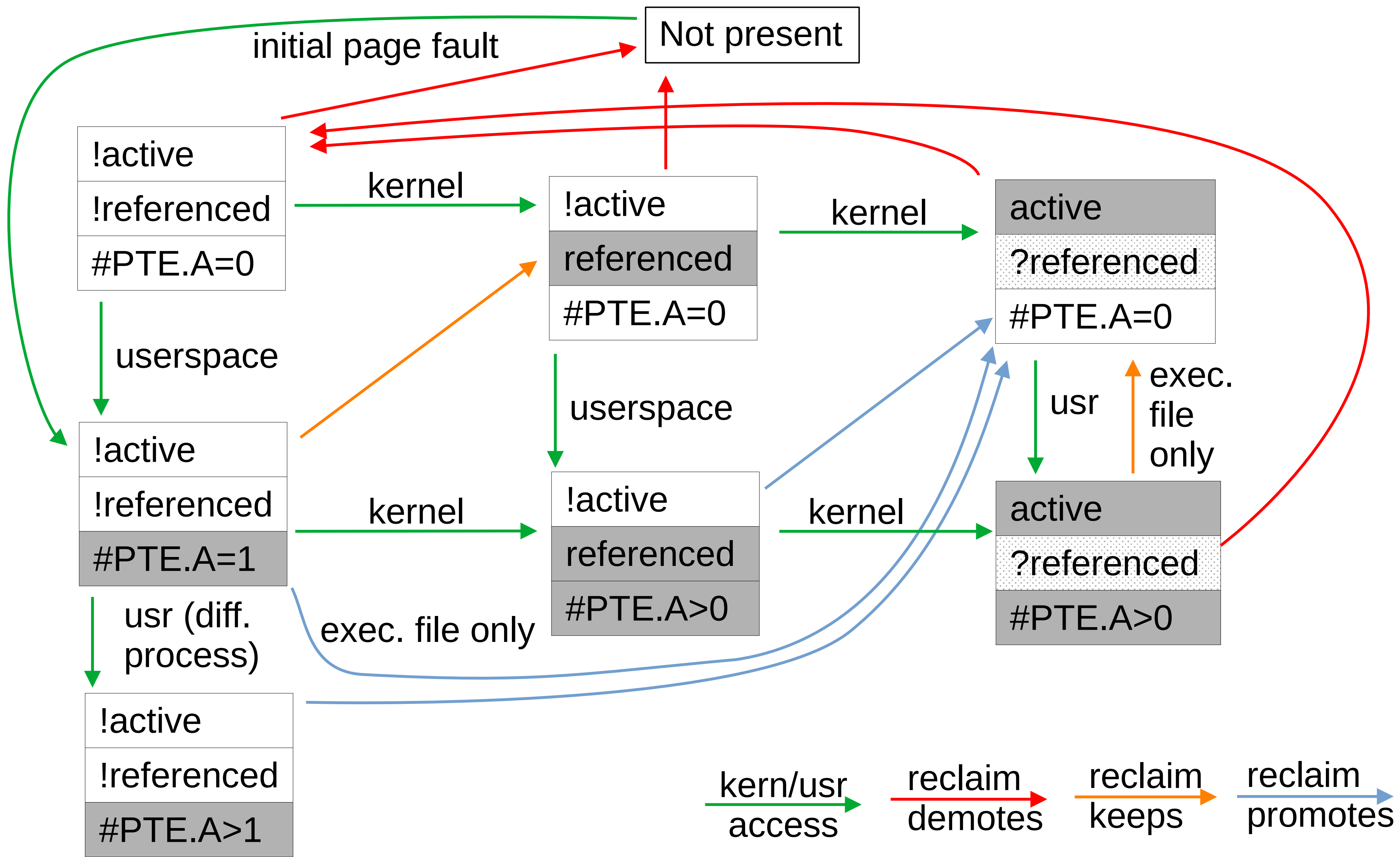
Not present
initial page fault
!active
!referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
usr (diff. process)
!active
!referenced
#PTE.A>1
!active
referenced
#PTE.A=0
userspace
!active
referenced
#PTE.A>0
exec. file only
active
?referenced
#PTE.A=0
usr
exec. file only
active
?referenced
#PTE.A>0
kern/usr access
reclaim demotes
reclaim keeps
reclaim promotes

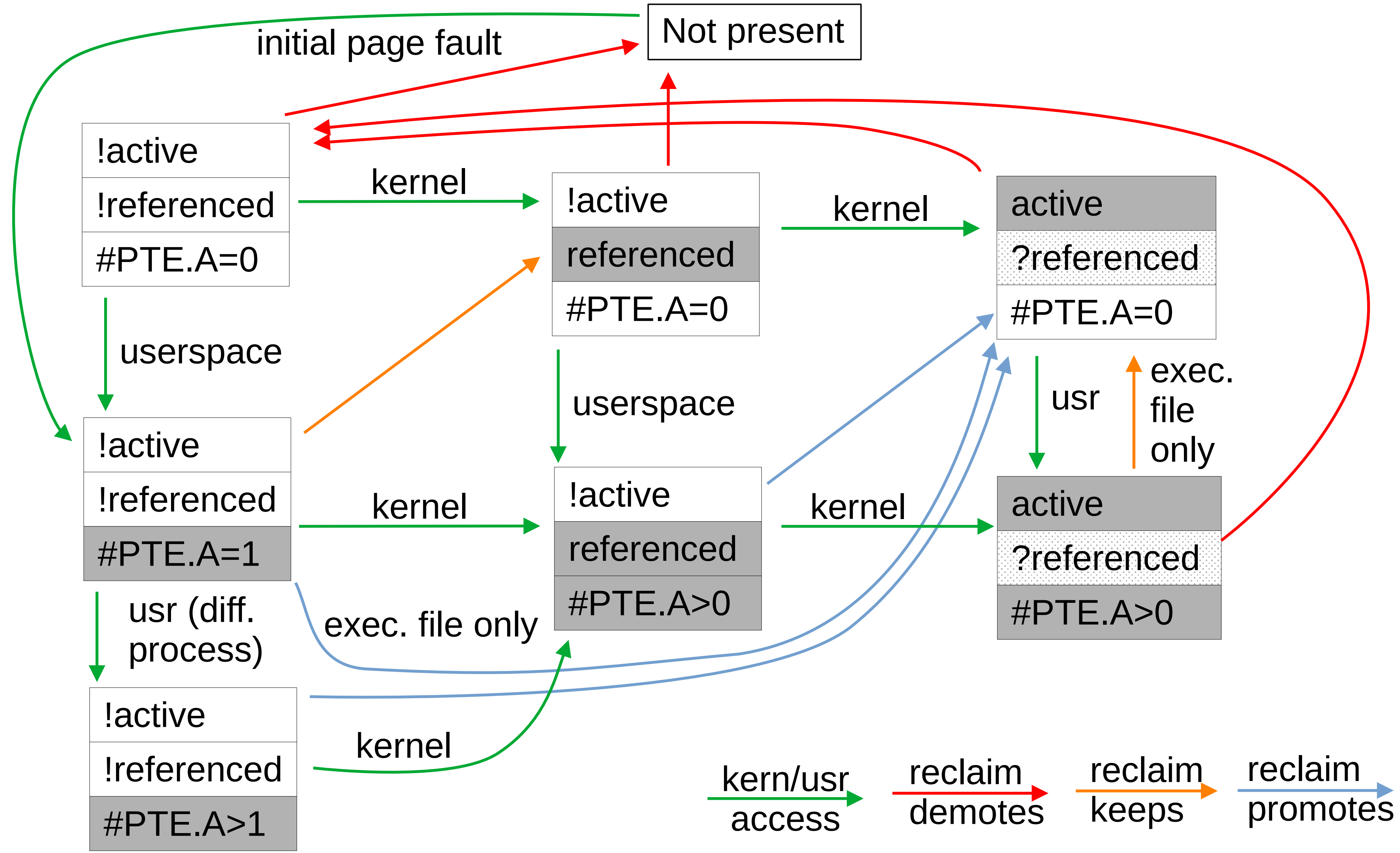

Not present
initial page fault
!active
!referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
usr (diff. process)
!active
!referenced
#PTE.A>1
!active
referenced
#PTE.A=0
userspace
!active
referenced
#PTE.A>0
exec. file only
active
?referenced
#PTE.A=0
usr
exec. file only
active
?referenced
Pages accessed from multiple processes are also immediately activated
kern/usr access
reclaim demotes
keeps
promotes


Not present
initial page fault
!active
!referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
usr (diff. process)
!active
!referenced
#PTE.A>1
!active
referenced
#PTE.A=0
userspace
!active
referenced
#PTE.A>0
exec. file only
active
?referenced
#PTE.A=0
usr
exec. file only
active
?referenced
#PTE.A>0
kern/usr access
reclaim demotes
reclaim keeps
reclaim promotes


Not present
initial page fault
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
Access by kernel such as by get_user_pages() is handled by mark_page_accessed()
userspace
usr
exec. file only
!active
referenced
#PTE.A>0
active
?referenced
#PTE.A>0
#PTE.A=1
usr (diff. process)
exec. file only
!active
!referenced
#PTE.A>1
kern/usr access
reclaim demotes
reclaim keeps
reclaim promotes


Not present
initial page fault
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
userspace
!active
!referenced
#PTE.A=1
userspace
!active
referenced
#PTE.A>0
usr
exec.
file
only
active
?referenced
#PTE.A>0
usr (diff.
process)
exec. file only
!active
!referenced
#PTE.A>1
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes


Not present
initial page fault
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
userspace
userspace
usr
exec.
file
only
!active
!referenced
#PTE.A=1
kernel
!active
referenced
#PTE.A>0
kernel
active
?referenced
#PTE.A>0
usr (diff.
process)
exec. file only
!active
!referenced
#PTE.A>1
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes



Not present
initial page fault
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
userspace
userspace
usr
exec.
file
only
!active
!referenced
#PTE.A=1
kernel
!active
referenced
#PTE.A>0
kernel
active
?referenced
#PTE.A>0
usr (diff.
process)
exec. file only
!active
!referenced
#PTE.A>1
kernel
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes

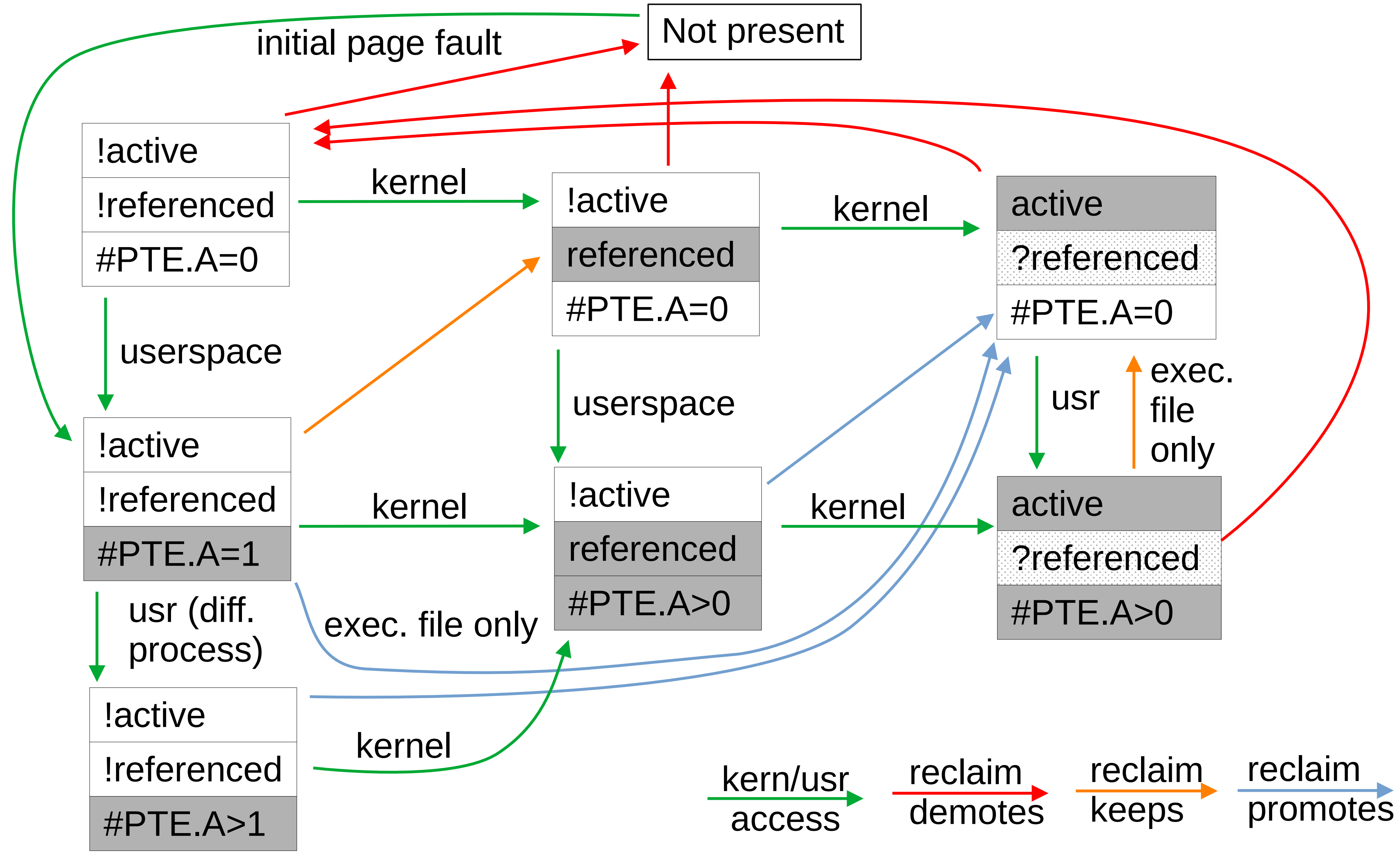
Workingset Detection

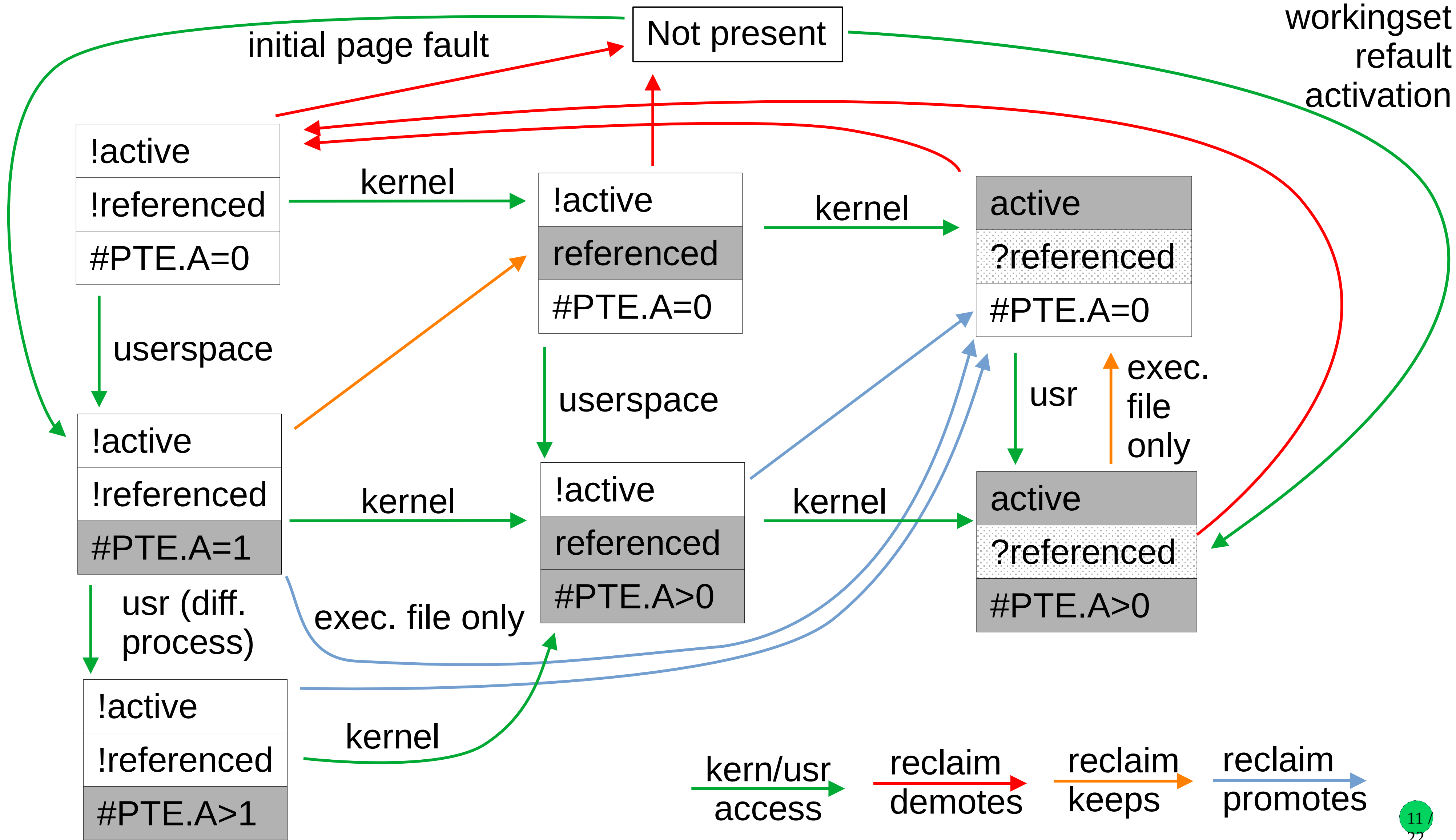
- Premise: transitioning workloads might be thrashing if pages are not accessed often enough while on inactive list to have chance to be promoted
  - Inactive list is intentionally small, the active working set might be just larger
  - If the reclaimed page is refaulted, we don't know if it's new or thrashing
  - Meanwhile the pages on active list might be idle, but we won't know
- Example: Workload accesses pages 7 8 9 10 11 7 8 9 10 11 ...
  - The access distance is 5 (4 different pages between two accesses to the same page)
  - Inactive list only has 4 pages, thus each access is a fault
  - Pages 1 – 6 might be actually idle
- Idea: determine this access distance, even for pages that have been evicted
  - Use *shadow entries* of radix tree/XArray for evicted pages
  - Precise tracking again impossible, need to approximate

active						inactive				evicted
1	2	3	4	5	6	11	10	9	8	7

Approximating Access Distance

- Observation: Access that causes fault places page to inactive list head, slides all towards tail, evicts tail page
- Observation: Access on inactive list results in activation, also slides all pages previously ahead of the page on the inactive list towards tail
- Thus: we can approximate inactive page accesses as sum of evictions and activations
- And: N of these accesses slide an inactive page N slots towards tail
- Eviction means **`NR_inactive`** pages were accessed while page was in memory
- If we note sum of evictions + activations at the moment of eviction (**`E`**), and at the moment of refault (**`R`**), the difference (**`R-E`**) approximates number of accesses while the page was evicted – called *refault distance*
- Complete access distance: **`NR_inactive + (R–E)`**
- Page would not be evicted if: **`NR_inactive + (R–E) <= NR_active + NR_inactive`**
- Simplified: **`(R–E) <= NR_active`**
  - When this inequality holds on refault, activate page immediately


Not present
initial page fault
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
userspace
userspace
usr
exec.
file
only
!active
!referenced
#PTE.A=1
kernel
!active
referenced
#PTE.A>0
kernel
active
?referenced
#PTE.A>0
usr (diff.
process)
exec. file only
!active
!referenced
#PTE.A>1
kernel
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes


Not present
initial page fault
workingset
refault
activation
!active
!referenced
#PTE.A=0
kernel
!active
referenced
#PTE.A=0
kernel
active
?referenced
#PTE.A=0
userspace
userspace
usr
exec.
file
only
!active
!referenced
#PTE.A=1
kernel
!active
referenced
#PTE.A>0
kernel
active
?referenced
#PTE.A>0
usr (diff.
process)
exec. file only
!active
!referenced
#PTE.A>1
kernel
kern/usr
access
reclaim
demotes
reclaim
keeps
reclaim
promotes

Workingset Detection Implementation

- Initially implemented for file pages only, recently also for anonymous pages
- Counter of evictions plus activations in `lruvec->nonresident_age`
- Refault distance is compared to workingset size
  - Sum of all LRU sizes except inactive list of page’s type
  - File page refault distance compared to `NR_active_file + NR_active_anon + NR_inactive_anon`
  - Anon page refault distance compared to `NR_active_anon + NR_active_file + NR_inactive_file`
  - But if swap is not available, anon list sizes are not included in the sums
- When page is deactivated, its `Workingset` flag is set
  - The flag is recorded in shadow entry, and set again upon refault, never cleared (i.e. only when stale shadow entries are reclaimed)
  - Refaults with `Workingset` flag restored play role in reclaim cost model
  - But frequent refaults with workingset flag mean the active list itself is thrashing; workload is not changing, but does not fit and we could OOM (with PSI)

Global Reclaim Algorithm

- Per-node kswapd or direct reclaim when a node is below watermarks – both eventually call `shrink_node()`
- Decide if anon and/or file pages should be deactivated – active/inactive balancing
  - Goal: large active list with low amount of reclaim work, small inactive list as a busy "proving ground", except when the workload is transitioning
  - Formula in `inactive_is_low()`, based on `sqrt` of the active+inactive list sizes
    - 1:1 up to 100MB worth of memory on the LRU lists
    - 3:1 (active:inactive) at 1GB memory – 25% pages should be on inactive list
    - 320:1 at 10TB memory
    - Consequence: memcg reclaim changes the ratio towards smaller active lists
  - Deactivation allowed when inactive list size is below the target ratio
  - Or when workingset refaults are happening, based on a rather coarse check (the counter of file workingset refaults changed since last reclaim)

Global Reclaim Algorithm #2

Anon/file balancing – decide how much to shrink from each type’s LRU

- Some corner case decisions first
  - “Many” (based on reclaim priority) inactive file pages and we do not deactivate file pages, prioritize file reclaim – “cache trim mode”
  - Too few file pages (active+inactive) with “many” inactive anon pages and we do not deactivate anon pages, prioritize anon reclaim – “file is tiny”
    - Tries to prevent runaway feedback loop where small file LRU means no chance to get pages promoted
- Iterate over all memcgs, calling `shrink_lruvec()`
- Determine how much to scan in each LRU list by `get_scan_count()`
  - Consider only file LRUs – swapping not possible or cache trim mode enabled
  - Consider only anon LRUs – “file is tiny”
  - Scan both equally – close to OOM (but `swappiness` is not 0) - no time for fine balancing
  - Balance anon and file LRUs according to Fractional Cost Model

Global Reclaim Algorithm #3

Anon/file fractional cost model

- Idea: if reclaim causes more IO for file pages than anon pages, put more pressure on anon pages, and vice versa – pressure is inversely proportional to to cost
- We count workingset refaults that restore `Workingset` flag (which means a formerly active page was reclaimed), and dirty page write-outs, as the reclaim cost
  - To soften corner cases, soften the resulting pressure between 0 and 1 to between 1/3 and 2/3
- This is also weighted by `vm.swappiness` sysctl, with range from 0 to 200 (default 60)
  - `vm.swappiness=0` – anon reclaim has infinite cost, reclaim only file pages
  - `vm.swappiness=100` – anon and file pages have same IO cost
  - `vm.swappiness=200` – file reclaim has infinite cost, reclaim only anon pages
- The result is fraction between 0 and 1 for anon, and for file, both add up to 1
- Calculate how many pages to scan from each LRU list - *target*
  - `NR_pages >> reclaim_prio` (prio starts at 12 – 1/4096 of the list, prio decreased each round)
  - Apply calculated fraction, or set to 0 if we are not reclaiming the particular type

Global Reclaim Algorithm #4

- The LRU list shrinking itself
  - Call `shrink_list()` in a loop, scan up to 32 pages (`SWAP_CLUSTER_MAX`) in iteration
    - Skip active list if deactivation is not allowed
  - Isolate pages from tail of list, then deactivate, keep or reclaim according to their flags and page table entries with active bit set
  - Terminate when budget (initialized by `get_scan_count()` targets) is exhausted for all lists
  - After having reclaimed the target number of pages (`SWAP_CLUSTER_MAX` or high watermark), keep scanning to deplete the rest of the budget, but:
    - Stop scanning the file/anon type with lower remaining budget
    - For the other type, adjust the budget to keep the original anon/file ratio
    - Example: target was 64 file, 32 anon pages, after scanning and reclaiming 16 from each, scan additional 16 file pages (so the result is 32 file, 16 anon)
  - Finally, scan 32 pages from active anon list
    - If swap is available and inactive anon is low
    - Ignores prior decision whether to deactivate anon

`madvise(2)` - reclaim related flags

- `MADV_DONTNEED` – throw away private anonymous pages, unmap file pages
  - might be reclaimed later due to memory pressure, no explicit reclaim action
- `MADV_FREE` – private anon only – clear page dirty, referenced flags, move it to inactive *file* list
  - pages will be discarded (destructive, no swap-out) soon in case of memory pressure
- `MADV_COLD` – deactivate pages (move to inactive list, clear referenced flags)
  - swap-out or dirty page writeback will happen during reclaim (non-destructive)
  - only pages not mapped by multiple processes
- `MADV_RECLAIM` – immediately reclaim pages
  - including swap-out or dirty page writeback
  - only pages not mapped by multiple processes

Conclusion

- This was an overview, implementation has even more details and special cases
- Some topics omitted completely
  - Writeback, swapping, dirty throttling, memcg reclaim, slab reclaim (shrinkers), watermarks handling, kswapd vs direct reclaim, reclaim/compaction, OOM, PSI…
- Complex system, results of years of evolution, including big recent changes
  - No overall documentation (perhaps getting there? :)
- Many moving parts, hard to predict behavior, hard to evaluate patches!
  - Elaborate cost models applied only to 1/3 of decision space
  - OTOH, major decisions made by looking if a number has changed since last time
  - Explicit corner case heuristics against undesired feedback loops
  - Lots of suspicious details to look at in my TODO
  - We’ve seen issues (in older kernel) e.g. with file pages thrashing and anon not reclaimed
- How to get better insight? A simulation model?

Recent patch series related to reclaim

- Migrating pages to slower memory instead of reclaim – merged for 5.15
  - By Dave Hansen and Huang Ying (Intel)
  - Such as persistent memory, when used as a NUMA node
  - Has to be enabled by /sys/kernel/mm/numa/demotion_enabled
  - For now, does not promote pages back to faster DRAM/closer node based on usage
  - Another patchset by Huang towards “memory tiering system” does that based on NUMA balancing code
  - Another patchset by Tim Chen (Intel) improves admin control of DRAM usage based on memcg and soft limits

Multigenerational LRU Framework

- Patchset from Yu Zhao (Google), v1 in March, v4 in August 2021
- Multiple generations (at least 3) instead of active/inactive lists – separate lists (per file/anon and zone), generation number in page flags word
  - Faults go to youngest generation, buffered file accessed to oldest
  - Accessed bit (found during scan) moves page to youngest generation
- Generations also divided to tiers for more fine-grained mark_page_accessed() counting, tier also part of page flags, but not separate lists
  - Balancing tiers using workingset refault info, PID controller-like feedback loop
- Scanning for accessed bits through page table walks, not lru lists (as was in past)
  - Attempts to exploit spatial locality, avoid expensive rmap walks, fallback on sparse maps
  - Lists of mm structs per memcgs, skipping of sleeping processes, inactive PMDs, no page level zigzag between vma's
- Eviction processes oldest generation, balances between file and anon by refaults

Multigenerational LRU Framework

- Optional, run-time enable, aging, protection, monitoring sysfs knobs
- Pros:
  - Kswapd reduced rmap walk CPU usage, reduced direct reclaim latency
  - Tools for workload scheduling decisions, proactive reclaim
  - Some success stories – reduced swap storms, improved throughputs…
- Cons:
  - Changes many things at once, kernel development prefers incremental improvements
  - Additional to existing mechanism, not replacement → maintenance burden
  - Adds user space knobs (but not mandatory to use)

Thank you.