Linux memory management
(with focus on page allocations)

Vlastimil Babka
Linux Kernel Developer, SUSE Labs
vbabka@suse.cz
Advanced Operating Systems 2022/2023
Documentation and other sources

• Documentation/mm
  – Still ad-hoc, ongoing effort to systematize and fill the gaps

• Books (of the past and the future)
  – Understanding The Linux Virtual Manager (Mel Gorman)
    • Very good and systematic coverage but too old – from 2.4 era (with What’s new in 2.6 sections)
    • Still very useful to understand core design principles
    • https://www.kernel.org/doc/gorman/
  – The Linux Memory Manager (Lorenzo Stoakes)
    • “The target release date is mid-late 2024” and “the book will target Linux 6.0”
    • “I have written 656 pages of a target of roughly 1,500 pages.”
    • https://linuxmemory.org/

• LWN - https://lwn.net/
  – Many very good articles (not limited to kernel), LSF/MM conference coverage…

• Various company-branded or personal blog posts
Linux MM – APIs for kernel

- bootmem/memblock allocator – early initialization
- page allocator – page order ($2^N$ physically contiguous pages)
- slab allocator – sub page granularity, internal fragmentation management
  - SLAB – based on Solaris design – optimized for CPU cache and memory efficiency
  - SLUB – new design – aimed for better scalability at the expense of more memory, also much better debugging capabilities; default for many years now
- vmalloc – virtually contiguous memory allocator – via page tables
- mempool allocator – a layer on top of page or slab allocator
  - guarantee for a forward progress – mostly for IO paths
- Page cache management for filesystems
- Memory tracking for userspace – process management
- Page table management
  - get_user_pages() – virtual→struct page translation
- On-demand memory paging
MM – APIs for userspace

• Syscalls to manage memory
  – mmap, munmap, mprotect, brk, mlock – POSIX
  – madvise – hints from userspace e.g. MADV_DONTNEED, MADV_FREE etc...
  – userfaultfd – page fault handling from userspace
  – SystemV shared memory – IPC, shmget, shmat, shmdt
  – memfd_create – anonymous memory referenced by a file descriptor – for IPC

• Memory backed filesystems
  – ramdisk – fixed sized memory backed block device
  – ramfs – simple memory backed filesystem
  – tmpfs – more advanced memory backed filesystem, support for swapout, ACL, extended attributes

• Memory cgroups controller – more fine grained partitioning of the system memory
  – Mostly for user space consumption limiting, kernel allocations are opt-in
  – Support for hard limit, soft/low limit, swap configuration, userspace OOM killer

• Access to huge pages (2MB, 1GB)
  – hugetlbfs – filesystem backed by preallocated huge pages
  – THP – transparent huge pages for anonymous private or tmpfs memory

• NUMA allocation policies
  – mbind, set_mempolicy, get_mempolicy
Physical memory representation

• Managed in page size granularity – arch specific, mostly 4kB
• Each order-0 page is represented by struct page
  – Higher-order pages typically “compound pages”, first struct page “head”, the rest “tail” with a link to head
• Heavily packed – 64B on 64bit systems (~1.5% with 4kB pages)
  – Lots of unions to distinguish different usage, or distinct types reinterpreting whole struct page (e.g. struct slab)
  – Special tricks to save space – set bottom bits in pointers etc…
  – Page flags for page state, including page lock (bit lock)
• Statically allocated during boot/memory hotplug – memmap
  – Typically “sparsemem vmemmap” – virtually contiguous, 0xffffffff on x86_64 (modulo KASLR)
  – Pages belong to different NUMA nodes and zones within the nodes, node/zone ids are part of page flags word
• Reference counted to control lifetime and allow sharing and ad-hoc access
  – get_page(), put_page(), get_page_unless_zero()
  – memory is returned to the page allocator when refcount drops to 0
• pfn_valid(), pfn_to_page(), page_to_pfn() – physical page frame number to struct page translation
• struct folio – a new type to better abstract both order-0 and compound head page (cannot be a tail page), layout matches struct page, gradually introduced throughout mm
Page allocator

- `alloc_pages(gfp_t gfp_mask, unsigned int order)` to get a struct page (and the associated physical memory)
  - `alloc_pages_node(int nid, ...)` to indicate the preferred numa node
- `order` – size of the allocation will be $2^\text{order}$ contiguous naturally aligned pages
- `gfp_mask` – bitmask for the allocation mode
  - Restrict to/allow specific zones – `__GFP_DMA`, `__GFP_DMA32`, `__GFP_HIGHMEM`, `__GFP_MOVABLE`
  - Define allocation context wrt possibility of doing memory reclaim if free memory not available anymore
    - `__GFP_KSWAPD_RECLAIM`, `__GFP_DIRECT_RECLAIM`, `__GFP_IO`, `__GFP_FS`
  - Define allocation context wrt how hard to try succeed vs availability to fallback
    - Reserves access: `__GFP_HIGH`, `__GFP_MEMALLOC`, `__GFP_NOMEMALLOC`
    - Urgency: `__GFP_NORETRY`, `__GFP_RETRY_MAYFAIL`, `__GFP_NOFAIL`
  - Page mobility hints to help anti-fragmentation mechanisms
    - `__GFP_MOVABLE`, `__GFP_RECLAIMABLE`
  - Standard combinations defined for most typical contexts:
    - `GFP_KERNEL: __GFP_RECLAIM | __GFP_IO | __GFP_FS` – unmovable allocation, can reclaim both by kswapd and directly
    - `GFP_HIGHUSER_MOVABLE: GFP_KERNEL | __GFP_HIGHMEM | __GFP_MOVABLE` – can reclaim, can use highmem and movable zones, pages are going to be movable
    - `GFP_NOWAIT: __GFP_KSWAPD_RECLAIM` – unmovable kernel allocation, cannot direct reclaim
    - `GFP_ATOMIC: __GFP_KSWAPD_RECLAIM | __GFP_HIGH` – like GFP_NOWAIT but higher priority, can dip into reserves
Page allocator – memory reclaim

• Eventually memory will become (nearly) all used due to caching file contents (page cache) as well as kernel objects, for faster access
• Each zone has watermarks (scaled to its size) min < low < high, free pages checked during page allocation
  – Below low watermark: wake up kswapd kthread to reclaim up to high watermark
  – Below min watermark: the allocation itself has to reclaim up to min watermark
• Reclaim will try to evict a mix of userspace pages and kernel objects
  – Anonymous pages (from mmap(MAP_PRIVATE)) must be swapped out first
  – Page cache must be written back when dirty, or simply discarded when clean
  – Kernel objects: each type of reclaimable objects registers shrinker callbacks with specific implementation of both tracking of hot/coldness, and actual freeing
• To minimize disk I/O and latency, we want to reclaim cold pages
  – Struct pages are linked on a LRU list sorted from most recent (head) to least recent (tail)
LRU list – ideal model

<table>
<thead>
<tr>
<th>recent</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

stale
# LRU list – ideal model

Page 5 accessed

<table>
<thead>
<tr>
<th>recent</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>stale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
LRU list – ideal model

<table>
<thead>
<tr>
<th>recent</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>stale</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 5 accessed

<table>
<thead>
<tr>
<th>recent</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>stale</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LRU list – ideal model

Page 5 accessed

Page 11 accessed
LRU list – ideal model

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

Page 5 accessed

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

Page 11 accessed

recent | stale
-------|-------
11     | 5     | 1     | 2     | 3     | 4     | 6     | 7     | 8     | 9
LRU list – ideal model

Recent
Page 5 accessed
Recent
Page 11 accessed
Recent
Page 10 evicted
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
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

<table>
<thead>
<tr>
<th>recent</th>
<th>stale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

anon LRU

<table>
<thead>
<tr>
<th>file LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

1 2 3 4 5 6 7 8
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 hotter and colder pages separately
• Hence separate active and inactive LRU lists for each type
  – Also virtual fifth list for unevictable pages – not relevant to reclaim, not linking any pages today
  – 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 hotter and colder pages separately

• Hence separate active and inactive LRU lists for each type
  – Also virtual fifth list for unevictable pages – not relevant to reclaim, not linking any pages today
  – 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 hotter and colder pages separately

• Hence separate active and inactive LRU lists for each type
  – Also virtual fifth list for unevictable pages – not relevant to reclaim, not linking any pages today
  – 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 hotter and colder pages separately

• Hence separate active and inactive LRU lists for each type
  – Also virtual fifth list for unevictable pages – not relevant to reclaim, not linking any pages today
  – All together that’s called lruvec

<table>
<thead>
<tr>
<th>anon LRU</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>file LRU</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

| anon active | 1 | 4 | 5 |
| anon inactive | 6 | 8 |
| file active | 2 | 3 | 7 |
| file inactive | 9 | 10 |
| unevictable LIST_POISON1 | LIST_POISON2 | lruvec |
LRU – node/memcg lruvecs

- Four reclaimable LRU lists per lruvec
  - Large part of reclaim heuristics 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 heuristics 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 heuristics 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 Iruvecs

• Four reclaimable LRU lists per Iruvec
  – Large part of reclaim heuristics 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 Iruvecs
  – Different memory cgroups have distinct Iruvecs, for memcg reclaim
    • Global memory reclaim has to iterate over all memcgs
  – Different NUMA nodes have distinct Iruvecs, 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 Iruvecs:

<table>
<thead>
<tr>
<th>Node</th>
<th>Root memcg</th>
<th>Memcg1</th>
<th>Memcg2</th>
<th>Memcg3</th>
<th>Memcg4</th>
<th>Memcg5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 0</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
</tr>
<tr>
<td>Node 1</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
<td>Iruvec</td>
</tr>
</tbody>
</table>
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
  – CPU sets them, folio_referenced() counts and resets (via a rmap walk) them
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
  – CPU sets them, folio_referenced() counts and resets (via a rmap walk) them

<table>
<thead>
<tr>
<th>Page flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
</tr>
</tbody>
</table>
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
  - CPU sets them, \texttt{folio\_referenced()} counts and resets (via a \texttt{rmap walk}) them

\begin{itemize}
  \item \texttt{struct page}
  \begin{tabular}{|c|c|c|c|}
    \hline
    Page flags & \hline
    LRU & Active & Referenced & Workingset \\
    \hline
  \end{tabular}

  \item \texttt{Page table entry}
  \begin{tabular}{|c|c|c|c|c|c|c|}
    \hline
    & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
    \hline
    Dirty & Accessed & ... & ... & U/S & R/W & P \\
    \hline
  \end{tabular}
\end{itemize}
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
  - CPU sets them, folio_referenced() counts and resets (via a rmap walk) them
### 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
  - CPU sets them, `folio_referenced()` counts and resets (via a `rmap walk`) them

---

**struct page**

<table>
<thead>
<tr>
<th>Page flags</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>Active</td>
</tr>
</tbody>
</table>

**Page table entry**

```
<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty</td>
<td>Accessed</td>
<td>...</td>
<td>...</td>
<td>U/S</td>
<td>R/W</td>
<td>P</td>
</tr>
</tbody>
</table>
```

---

**Page table entry**

```
<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty</td>
<td>Accessed</td>
<td>...</td>
<td>...</td>
<td>U/S</td>
<td>R/W</td>
<td>P</td>
</tr>
</tbody>
</table>
```
After fault is handled, the userspace access is restarted and sets PTE Accessed bit immediately.
initial page fault

!active
!referenced
#PTE.A=1
Reclaim filters out the initial access by only setting the referenced Page flag, but keeping Page on inactive list.
initial page fault

Not present

!active

referenced

#PTE.A=0

!active

!referenced

#PTE.A=1

kern/usr

access

reclaim

keeps
Another access sets PTE active bit

draws

initial page fault

Not present

!active

referenced

#PTE.A=0

userspace

!active

referenced

#PTE.A=1

Not present

!active

referenced

#PTE.A=0

Another access sets PTE active bit

kern/usr

access

reclaim

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

Not present

kern/usr
access

reclaim
keeps

reclaim
promotes

initial page fault
initial page fault

Not present

Reclaim sees no active bit PTEs, page was not accessed, evict it

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
<th>#PTE.A=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!active</td>
<td>!referenced</td>
<td></td>
</tr>
</tbody>
</table>

userspace

!active
!referenced
#PTE.A=1

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
<th>#PTE.A&gt;0</th>
</tr>
</thead>
<tbody>
<tr>
<td>!active</td>
<td>!referenced</td>
<td></td>
</tr>
</tbody>
</table>

kern/usr
access

reclaim
demotes

reclaim
keeps

reclaim
promotes
initial page fault

Active list reclaim (deactivation) referenced flag doesn’t matter

![chart](chart.png)
initial page fault

!active
!referenced
#PTE.A=0

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

active
?referenced
#PTE.A=0

userspace

kern/usr
access

reclaim
demotes

reclaim
keeps

reclaim
promotes

Not present

13 / 26
initial page fault

Not present

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

userspace

!active
referenced
#PTE.A=0

active
?referenced
#PTE.A=0

!active
referenced
#PTE.A=0

userspace

initial page fault

kern/usr access
reclaim demotes
reclaim keeps
reclaim promotes

Not present
initial page fault

Not present

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

!active
referenced
#PTE.A=0

active
?referenced
#PTE.A=0

kern/usr
don't access

reclaim
demotes

reclaim
keeps

reclaim
promotes
initial page fault

!active
!referenced
#PTE.A=0

userspace

!active
!referenced
#PTE.A=1

!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

13
Executable file pages are kept on active list as long as they are accessed.
Executable file pages are also immediately activated.

- **Not present**
  - active
  - referenced
  - #PTE.A=0

- **userspace**
  - active
  - referenced
  - #PTE.A=0

- **exec. file only**
  - active
  - referenced
  - #PTE.A>0

- **initial page fault**
  - active
  - referenced
  - #PTE.A=1

- kern/usr access
  - reclaim demotes
  - reclaim keeps
  - reclaim promotes
Pages accessed from multiple processes are also immediately activated.
initial page fault

Not present

!active
!referenced
#PTE.A=0

userspace

!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

userspace

!active
!referenced
#PTE.A=1

usr (diff. process)

!active
referenced
#PTE.A>0

exec. file only

!active
referenced
#PTE.A>0

usr (diff. process)

exec. file only

exec. file only

kern/usr access

reclaim demotes

reclaim keeps

reclaim promotes
Access by kernel such as by `get_user_pages()` is handled by `mark_page_accessed()`.

Not present

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A=0</td>
<td></td>
</tr>
</tbody>
</table>

Kernel

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A=0</td>
<td></td>
</tr>
</tbody>
</table>

Userspace

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A=0</td>
<td></td>
</tr>
</tbody>
</table>

usr (diff. process)

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A&gt;0</td>
<td></td>
</tr>
</tbody>
</table>

usr

exec. file only

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A&gt;0</td>
<td></td>
</tr>
</tbody>
</table>

exec. file only

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A=0</td>
<td></td>
</tr>
</tbody>
</table>

Kernel

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A=1</td>
<td></td>
</tr>
</tbody>
</table>

usr (diff. process)

<table>
<thead>
<tr>
<th>active</th>
<th>referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>#PTE.A&gt;1</td>
<td></td>
</tr>
</tbody>
</table>

Kern/usr access

Reclaim demotes

Reclaim keeps

Reclaim promotes

Initial page fault

Not present
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 workload’s working set might be just larger
  - If a recently reclaimed page is faulted in again, 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 (NR_inactive = 4), thus each access is a fault
  - Pages 1 – 6 were active before but now may 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 to store information
  - Precise tracking again impossible, need to approximate

<table>
<thead>
<tr>
<th>active</th>
<th>inactive</th>
<th>evicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>11 10 9 8 7</td>
<td></td>
</tr>
</tbody>
</table>
Approximating Access Distance

• Observation: Access that causes page fault places the page to inactive list head, pushes all other pages towards tail, evicts tail page
• Observation: Access on inactive list results in activation, also pushes all pages previously ahead of the page on the inactive list towards tail
• Thus: sum of evictions and activations over some time period means at least N inactive page accesses happened during that period
• And: pushing an inactive page N slots towards tail needs at least N inactive page accesses
• Eviction of a page means at least $NR_{inactive}$ pages were accessed while it 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 minimum access distance: $NR_{inactive} + (R - E)$
• Page would not be evicted if: $NR_{inactive} + (R - E) \leq NR_{active} + NR_{inactive}$
• Simplified: $(R - E) \leq NR_{active}$
  – When this inequality holds on refault, activate page immediately
• Full writeup: see mm/workingset.c
Workingset Detection Implementation

• Initially implemented for file pages only, later also for anonymous pages
• Counter of evictions plus activations in lruvec->nonresident_age
  – Counters of refaults in lruvec->refaults[ANON_AND_FILE]
• Refault distance (R-E) is compared to workingset size
  – Sum of all LRU sizes except the inactive list of the 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
Global Reclaim Algorithm

• Per-node kswapd or direct reclaim when a node is below watermarks – both eventually call \texttt{shrink}\_\texttt{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’s working set is transitioning
  – Formula in \texttt{inactive\_is\_low()}, based on $\sqrt{\text{active+inactive list sizes}}$
    • 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
  – 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 of anon/file 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 – in `get_scan_count()`

• 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 IO 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 from interval [0, 1] to [1/3, 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** (since 4.5) – 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
  - a write to the page before the discard will cancel the discard
  - cheaper than MADV_DONTNEED – no immediate page table zapping

Since 5.4, also two new always non-destructive modes:

- **MADV_COLD** – deactivate pages (move to inactive list, clear referenced flags)
  - swap-out or dirty page writeback will happen during reclaim
  - only pages not mapped by multiple processes
- **MADV_PAGEOUT** – immediately reclaim pages
  - including swap-out or dirty page writeback
  - only pages not mapped by multiple processes
Page reclaim - 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
• 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
  – We’ve seen issues (in older kernel) e.g. with file pages thrashing and anon not reclaimed
Multigenerational LRU Framework

• Patchset from Yu Zhao (Google), v1 in March 2021, merged in v6.1 (Dec 2022)
• 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
  – Attempts to exploit spatial locality, avoid expensive rmap walks (fallback to Lru on sparse mappings); was actually done in old Linux versions
  – Maintains lists of mm structs per memcogs, skipping of sleeping processes and inactive PMDs, no page level zigzag between vma's
• Eviction processes oldest generation, balances between file and anon by refaults
Multigenerational LRU Framework

• Optional. Has sysfs knobs for run-time enable, protection, aging monitoring
• 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
    • Feedback not fully successful, “Linus likes this” helped merging anyway
  – Largely orthogonal to existing mechanism, not its replacement → maintenance burden
  – Adds user space knobs (at least not mandatory to use)
Thank you.