Introduction and Agenda

- Basics of Virtual Memory
- Huge Pages in the Virtual Memory Model
- Huge Pages in Linux
- Hugetlbfs Specifics

Scope and Expectations
- High level description of Virtual Memory
- Intel Architecture in examples
- Expertise only in hugetlbfs
Virtual Memory Basics

Process

System Memory
Memory Map

struct page for each page in system
Memory Map

Struct Page Key Fields

- Page Flags
  - PG_locked, PG_dirty, PG_active, PG_uptodate, PG_head, PG_hwpoison ...
- Reference Count
- Map Count

Commonly 64 Bytes in Size
Process

Virtual Addresses

Page Tables

System Memory

Physical Addresses
Page Table Diagram

- **pgd_t**: Pgd Page
- **pud_t**: Pud Page
- **pmd_t**: Pmd Page
- **pte_t**: Pte Page

`mm_struct.mm->pgd`
Page Table Specifics

- Each PGD, PUD, PMD or PTE table is one Page in size
- Table is an array of entries, each a word in size
- X86_64 example
  - Page - 4K
  - Word - 8bytes
  - 512 entries per page
    - PTRS_PER_PGD, PTRS_PER_PUD, PTRS_PER_PMD, PTRS_PER_PTE
Page Table Entries

- Typed for containing Table Page
  - pgd_t, pud_t, pmd_t, pte_t
- Provide a pointer to next Table Page (or User Data)
  - Page Frame Number (PFN) or Physical Address
  - Always points to PAGE, so PAGE_SHIFT bits available
- Entries contain flags such as:
  - _PAGE_PRESENT
  - _PAGE_RW
  - _PAGE_DIRTY
  - _PAGE_PSE
Process Memory

Virtual Addresses

Physical Addresses

Page Tables

System Memory
Virtual Address
bits per unsigned long

<table>
<thead>
<tr>
<th>PGD</th>
<th>PUD</th>
<th>PMD</th>
<th>PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
</table>

- **PGDIR_SHIFT** - 39
- **PUD_SHIFT** - 30
- **PMD_SHIFT** - 21
- **PAGE_SHIFT** - 12

X86_64 4 Level Page Tables
Virtual/Linear Address

Offset within PGD | Offset within PUD | Offset within PMD | Offset within PTE | Offset within Page

| pgd_offset | pgd_t | ... | | |
| pud_offset | pud_t | ... | | |
| mm_struct   | mm->pgd | | | |

pto_offset: pte_t
pte page

pmd_offset: pmd_t
pmd page

pud_offset: pud_t
pud page

pgd_offset: pgd_t
pgd page

User Data Page
### Virtual/Linear Address

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
</table>

- **pgd_offset**
- **pgd_t**
- **...**

**pgd page**

- **mm_struct**
- **mm->pgd**

**pud page**

- **pgd_offset** = `virtual_address` >> `PGDIR_SHIFT`
  - *mask off upper bits*

- **pgd_offset** - Index into PGD page
  - 0 - 511
Virtual/Linear Address

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
</table>

```
pud_offset = virtual_address >> PUD_SHIFT
mask off upper bits
```

```
pud_offset - Index into PUD page
```
Virtual/Linear Address

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
</table>

\[
pmd\_offset = \text{virtual\_address} \gg \text{PMD\_SHIFT}
\]

mask off upper bits

\[
pmd\_offset - \text{Index into PMD page}
\]
pte_offset = virtual_address >> PTE_SHIFT

mask off upper bits

pte_offset - Index into PTE page
Virtual/Linear Address

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
</table>

Offset within Page = virtual_address \gg PAGE_SHIFT

mask off upper bits

Finally have our data. Yeah!
Virtual Addresses → Page Tables → Physical Addresses
<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgd_offset</td>
<td>pud_offset</td>
<td>pmd_offset</td>
<td>pte_offset</td>
<td>User Data Page</td>
</tr>
<tr>
<td>pgd_t</td>
<td>pud_t</td>
<td>pmd_t</td>
<td>pte_t</td>
<td></td>
</tr>
<tr>
<td>pgd page</td>
<td>pud page</td>
<td>pmd page</td>
<td>pte page</td>
<td></td>
</tr>
<tr>
<td>mm_struct</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Translation Lookaside Buffer TLB

- Normally part of a CPU’s Memory Management Unit (MMU)
- A TLB is a cache of virtual-to-physical translations
  - Information stored in Page Tables
- Typically a scarce resource
## TLB Sizes

<table>
<thead>
<tr>
<th>Processor</th>
<th>ITLB 4K</th>
<th>ITLB 2M</th>
<th>DTLB 4K</th>
<th>DTLB 2M</th>
<th>DTLB 1G</th>
<th>STL B 4K</th>
<th>STL B 2M</th>
<th>STL B 1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>128</td>
<td>7</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sandy Br.</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Haswell</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>1024</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sky Lake</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>1536</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Ice Lake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2048</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>
Virtual Memory in the Kernel

• Kernel mostly uses virtual memory addresses
• Kernel also has a set of page tables
  – Translate from kernel virtual addresses to kernel data
Huge Pages … Finally!

- Huge Page sizes are typically associated with Page Table Level (PMD, PUD)
- Sizes Architecture dependent
- MMU/TLB support
- Huge Pages are contiguous areas of physical memory
  - Aligned to huge page size
  - Buddy Allocator contiguous areas less than \texttt{MAX\_ORDER} in size
  - \texttt{alloc\_contig\_pages()} for areas greater than \texttt{MAX\_ORDER}
  - CMA Allocator, MemBlock Allocator, Firmware
<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within PTE</th>
<th>Offset within Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgd_offset</td>
<td>pud_offset</td>
<td>pmd_offset</td>
<td>pte_offset</td>
<td>User Data Page</td>
</tr>
<tr>
<td>pgd_t</td>
<td>⋮</td>
<td>pmd_t</td>
<td>⋮</td>
<td>User Data Page</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
<td>User Data Page</td>
</tr>
<tr>
<td>mm_struct</td>
<td>mm-&gt;pgd</td>
<td>pud_t</td>
<td>⋮</td>
<td>User Data Page</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⋮</td>
<td>pte_t</td>
<td>User Data Page</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⋮</td>
<td>⋮</td>
<td>User Data Page</td>
</tr>
</tbody>
</table>
Huge Page at PMD Level

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within PMD</th>
<th>Offset within Huge Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgd_offset</td>
<td>pud_offset</td>
<td>pmd_offset</td>
<td></td>
</tr>
<tr>
<td>pgd_t</td>
<td>pud_t</td>
<td>pmd_t</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

`pgd_offset` is in the mm_struct `mm->pgd`, `pud_offset` is within `pud_t`, and `pmd_offset` is within `pmd_t`. The `_PAGE_PSE` flag is set in `pmd_t` (X86).
Huge Page at PUD Level

<table>
<thead>
<tr>
<th>Offset within PGD</th>
<th>Offset within PUD</th>
<th>Offset within Huge Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgd_offset</td>
<td>pud_offset</td>
<td>User Data</td>
</tr>
<tr>
<td>pgd_page</td>
<td>pud_page</td>
<td>Huge Page</td>
</tr>
<tr>
<td>mm_struct</td>
<td></td>
<td>User Data</td>
</tr>
<tr>
<td>mm-&gt;pgd</td>
<td></td>
<td>Huge Page</td>
</tr>
</tbody>
</table>

_PAGE_PSE_ flag set in pud_t (X86)
Huge Pages MAY Increase Performance

Pros
● Fewer Translation Entries
● Less time Servicing TLB misses
● Access Pattern Dependent

Cons
● Less Granular Page Size
● Fewer TLB entries
● Access Pattern Dependent

Benchmark  Benchmark  Benchmark
## Intel Processor TLB Sizes

<table>
<thead>
<tr>
<th>Processor</th>
<th>ITLB 4K</th>
<th>ITLB 2M</th>
<th>DTLB 4K</th>
<th>DTLB 2M</th>
<th>DTLB 1G</th>
<th>STLB 4K</th>
<th>STLB 2M</th>
<th>STLB 1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>128</td>
<td>7</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sandy Br.</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haswell</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sky Lake</td>
<td>128</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>4</td>
<td>1536</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Ice Lake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2048</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>
Huge Page APIs in Linux

- Transparent Huge Pages - THP
  - *Transparent* to the application
  - Automatic with some control

- Hugetlbfs
  - Requires application modification
  - Sysadmin intervention/setup
Transparent Huge Pages - THP

- Primarily used for anonymous memory
  - Can be used in tmpfs
  - Limited support for file mappings (XFS, experimental)
- Currently PMD_SIZE support only
- System control via `/sys/kernel/mm/transparent_hugepage/enabled` always [madvise] never
THP application use via madvise

madvise(void *addr, size_t length, int advice)

**MADV_HUGEPAGE**

Enable Transparent Huge Pages (THP) for pages in the range specified by addr and length. Currently, Transparent Huge Pages work only with private anonymous pages (see mmap(2)). The kernel will regularly scan the areas marked as huge page candidates to replace them with huge pages. The kernel will also allocate huge pages directly when the region is naturally aligned to the huge page size (see posix_memalign(2)).

**MADV_NOHUGEPAGE**

Ensures that memory in the address range specified by addr and
Hugetlbfs

- Requires Application modification
- Huge Pages are generally preallocated via sysadmin control
- Has been used by database for many years
- More recent use as backing for Virtual Machines
  - THP commonly used to back VMs
- Multiple Huge Page Sizes as supported by Architecture

Pools of hugetlb pages are created/preallocated
Application use the pages in these pools
Hugetlbfs Multiple Page Size Pools (arm64 4K Base Page Size)

/sys/kernel/mm/hugepages/
  ├── hugepages-1048576kB
  │   ├── free_hugepages
  │   │   └── surplus_hugepages
  │   ├── nr_hugepages
  │   │   └── surplus_hugepages
  │   └── nr_hugepages_mempolicy
  │       └── surplus_hugepages
  │       └── nr_overcommit_hugepages
  │           └── surplus_hugepages
  │           └── resv_hugepages
  │               └── surplus_hugepages
  └── hugepages-2048kB
      ├── free_hugepages
      │   └── surplus_hugepages
      ├── nr_hugepages
      │   └── surplus_hugepages
      └── nr_hugepages_mempolicy
          └── surplus_hugepages
          └── nr_overcommit_hugepages
              └── surplus_hugepages
              └── resv_hugepages
                  └── surplus_hugepages
                  └── surplus_hugepages

...
Default Hugetlb Page Size

# grep Huge /proc/meminfo
AnonHugePages:  71680 kB
ShmemHugePages:  0 kB
FileHugePages:  0 kB
HugePages_Total:  0
HugePages_Free:  0
HugePages_Rsvd:  0
HugePages_Surp:  0
Hugepagesize:  2048 kB
Hugetlb:  0 kB
Populating Hugetlb Page Pools

Boot Time

- `hugepagesz=X hugepages=Y`
  - `hugepages=N1:Y1, N2:Y2, N3:Y3, N4:Y4`
- `hugepage_cma=Y`
  - `hugepage_cma=N1:Y1, N2:Y2, N3:Y3, N4:Y4`
- `default_hugepagesz=X`

Run Time

- `echo N > /sys/kernel/mm/hugepages/hugepages-<size>/nr_hugepages`
- `echo N > /proc/sys/vm/nr_hugepages`
Mounting Hugetlbfs Filesystems

```
mount -t hugetlbfs -o uid=<value>,gid=<value>,mode=<value>,pagesize=<value>,size=<value>,
min_size=<value>,nr_inodes=<value> none /mnt/huge
```

- All files in filesystem are backed by Huge Pages
- pagesize if the Huge Page size
- Most file/filesystem, operations supported
  - write system call NOT supported
Application use via System V Shared memory

shmget(key, size, SHM_HUGETLB)

- Creates a shared memory segment backed by Huge Pages
- Additional flags  SHM_HUGE_2MB,  SHM_HUGE_1GB ...  
  - Specify the Huge Page Size
Application use via mmap

```c
void *mmap(addr, length, prot, flags, fd, offset)
```

- **fd** File in a mounted hugetlbfs filesystem
  - Backed by huge pages of the filesystem size
- **flags** `MAP_ANONYMOUS | MAP_HUGETLB`
  - Anonymous mapping backed by huge pages
  - **flags** `MAP_HUGE_2MB, MAP_HUGE_1GB` ...
    - Size of pages to back anonymous mapping
- **addr** and **offset** must be aligned to underlying Huge Page size
Running out of Huge Pages

- Huge Pages are generally limited to those in the Pool
- Huge Pages are not swappable or reclaimable
- When we are out, we are out
- Page fault with no Huge Pages available == SIGBUS
- Situation mitigated with Huge Page Reservations
Huge Page Reservations

- Global counter per-pool (resv_hugepages/HugePages_Rsvd)
- Incremented at `mmap/shmget` time
  - `ENOMEM` if not enough huge pages for reservation
- Decremented at page fault/allocation time
- `resv_map` internal data struct tracks reservations at a page level for all mappings
- `HugePages Available = HugePages_Free - HugePages_Rsvd`
PMD Sharing

- Processes can share PMD pages in their Page Tables
  - Only on x86 and arm64
- SHARED Hugetlb mappings (file or anonymous)
- Shared Virtual Range must be at least 1GB ($PUD\_SIZE$) and aligned

Example: 1TB Shared Mapping, 10,000 Processes sharing the mapping
- 4K PMD Page per 1G of shared mapping
- 1TB = 1024G * 9,999 shared mappings
- 39GB memory savings
Hugetlb vmemmap Freeing

- Recently added in v5.14
- vmemmap - Virtually mapped memmap so entries are virtually contiguous
- Memory saving feature
Memory Map

struct page for each page in system
sparsemem Memory Model

System Memory

Section 0
mem_map section 0

Section 1
mem_map section 1

Section 2
mem_map section 2

Section 3
mem_map section 3
| 2M huge page  
512 base pages |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>head page</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 1</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 2</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 3</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 4</td>
</tr>
<tr>
<td>strut page</td>
</tr>
<tr>
<td>tail page 510</td>
</tr>
<tr>
<td>strut page</td>
</tr>
<tr>
<td>tail page 511</td>
</tr>
</tbody>
</table>

| 1G huge page  
262144 base pages |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>head page</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 1</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 2</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 3</td>
</tr>
<tr>
<td>struct page</td>
</tr>
<tr>
<td>tail page 4</td>
</tr>
<tr>
<td>strut page</td>
</tr>
<tr>
<td>tail page 262142</td>
</tr>
<tr>
<td>strut page</td>
</tr>
<tr>
<td>tail page 262143</td>
</tr>
</tbody>
</table>
Remap and free pages of tail pages
Hugetlb vmemmap Freeing

- vmemmap pages freed at huge page allocation time
  - When added to hugetlb pools
- Only 2 vmemmap pages needed for each huge page
  - (v5.17 modification to reduce to 1 page needed)
- 2MB huge pages, 8 vmemmap pages total, free 6 pages
  - free 7 pages in v5.17
- 1GB huge pages, 4096 vmemmap pages total, free 4094 pages
  - free 4095 pages in v5.17
Hugetlb vmemmap Freeing

• Downside
  – Freed vmemmap pages must be allocated (and remapped) BEFORE huge pages can be removed from pool
  – Freeing a huge page can now fail if vmemmap can not be allocated

• Enabling
  – CONFIG_HUGETLB_PAGE_FREE_VMEMMAP
  – hugetlb_free_vmemmap=on  (default off)
  – CONFIG_HUGETLB_PAGE_FREE_VMEMMAP_DEFAULT_ON
Summary
Huge Pages MAY Increase Performance

Pros
● Fewer Translation Entries
● Less time Servicing TLB misses
● Access Pattern Dependent

Cons
● Less Granular Page Size
● Fewer TLB entries
● Access Pattern Dependent

Benchmark Benchmark Benchmark Benchmark
Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The **LF Mentoring Program** is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- **Outreachy remote internships program** supports diversity in open source and free software.
- **Linux Foundation Training** offers a wide range of **free courses**, webinars, tutorials and publications to help you explore the open source technology landscape.
- **Linux Foundation Events** also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at [events.linuxfoundation.org](http://events.linuxfoundation.org).