

# cgroup と sysfs ファイル

## トラブルシューティング事例から cgroup を深追いする

GMOペパボ 伊藤洋也

2020/10/17 第12回 コンテナ技術の情報交換会@オンライン

## 名前: 伊藤洋也 いとう ひろや (@hiboma )

所属: GMOペパボ セキュリティ対策室 (在職14年目)

職位: プリンシパルエンジニア

得意: Linux 低レイヤーのトラブルシューティング

ブログ: <https://hiboma.hatenadiary.jp/>

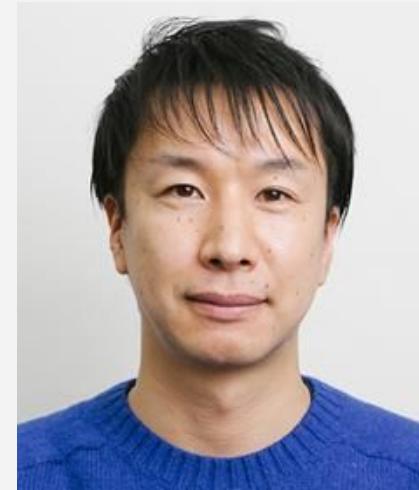
アウトプット事例:

『ペパボ トラブルシート伝 - TCP: out of memory -- consider tuning tcp\_mem の dmesg から辿る 詳解 Linux net.ipv4.tcp\_mem』

[https://tech.pepabo.com/2020/06/26/kernel-dive-tcp\\_mem/](https://tech.pepabo.com/2020/06/26/kernel-dive-tcp_mem/)

『ペパボ トラブルシート伝 - node プロセスの general protection fault を追う - abort(3) の意外な実装』

<https://tech.pepabo.com/2020/06/11/trouble-shooting-node-abort/>



# GMOペパボはテレワークを基本とする体制へ

GMOペパボ株式会社 は 2020年6月1日(月)より全パートナー(従業員)を  
対象にテレワークを基本とした勤務体制へと移行しました

私は、9月後半に東京都練馬区から栃木県那須塩原市に移住しました  
本日の発表も那須塩原の自宅から参加しています!

コンテナ勉強会オンライン開催に参加、ばんざい！

GMOペパボ、全社でテレワークを基本とする勤務体制へ移行 ~withコロナ時代における経営スタイル「新しいビジネス様式 byGMO」に基づき多様な働き方に対応

<https://pepabo.com/news/press/202006011200>



## 過去のコンテナ勉強会と私

第2回コンテナ型仮想化の情報交換会@東京に参加。2013/10/05（7年前）でした



- Docker の OSS が 2013年3月に出ていて、そのようなコンテナ時代での発表です。
- LXC で Ruby, PHP のWeb コンテナ環境を提供する PaaS Sqale (スケール) の設計・実装の話をしました

@ten\_foward さん、今回もよろしくお願いします!

カーネル／VM探検隊 第2回 コンテナ型仮想化の情報交換会@東京

<https://sites.google.com/site/kernelvm/kanren-ibento/di2hui-kontena-xing-jia-xiang-huan-qing-bao-jiao-huan-hui-dong-jing>

⚠ Sqale は既にサービスを閉じましたが、ロリポップ！マネージドクラウドに遺伝子が継がれています

## アジェンダ

- ロリポップ! マネージドクラウドの紹介
- cgroup と sysfs ファイル
- sysfs ファイルと uevent
- ロリポップ! マネージドクラウドでのトラブルシューティング
- まとめ



## Section 1

# ロリポップ！マネージドクラウド

先立ってロリポップマネージドクラウドの紹介をします  
本スライドのメインは、当サービスで実際に行ったトラブルシューティングを  
元に調査・検証した内容をまとめています

使うほどお得 > GMOあおぞらネット銀行 FX取引高国内1位 > GMOクリック証券 法的に有効 > GMO電子印鑑Agree

GMOペパボ株式会社 ロリポップ！レンタルサーバー ヘテムルレンタルサーバー ムームードメイン マネージドクラウド

2メートル距離を保けよう G M O  
GMO  
INTERNET SERVICES

LOLIPPOP! マネージドクラウド by GMOペパボ

ログイン 新規登録

落ちないから、  
大規模サイトも継々と移行中

マネージドクラウドがインフラ管理・コスト管理をし、制作会社や  
エンジニアの負担を減らします。環境構築もワンボタンで完了！

まずは無料でお試し10日間

2020/10/07 Python 3.9 提供開始のお知らせ

様々な用途でマネージドクラウドが採用されています。

人気ビジネス番組への出演時も機会損失が無くて  
良かったです

七洋製作所 様

「テレビ出演というビジネスチャンスであり、絶対にサーバ  
ーダウンは避けたい状況でしたが、マネクラはしっかり踏み

<https://mc.lollipop.jp/>

## サービスの簡単な紹介

- 各種言語・CMS のコンテナ環境を整えて  
Web アプリケーション実行環境を提供しております
- コンテナランタイムは haconiwa ( id: udzura, mruby 製 )



The screenshot shows the main interface of the 'lolipop! managed cloud' service. At the top, there's a navigation bar with links like 'GMOペパボ', 'ロリポップ!', 'マネージドクラウド', and 'ログイン'. Below the navigation, there's a large central area featuring a video player and several small preview windows showing different application environments. To the right of this central area, there's descriptive text about the service's simplicity and its support for various languages and frameworks. At the bottom, there's a yellow banner with the text 'マネージドクラウドは、Webアプリケーションや Webサイトを「楽」に運営するためのサービスです。' (The managed cloud is a service for easily operating Web applications and websites.) and a blue button labeled 'まずは無料でお試しいただけ'.



ロリポップ！マネージドクラウド by GMOペパボ  
<https://mc.lolipop.jp/>  
haconiwa  
<https://github.com/haconiwa/haconiwa>



The screenshot shows a Twitter profile for a user named 'Uchio Kondo'. The profile picture is a photo of a brown dog sleeping. The bio on the profile includes the text '9件のツイート' (9 tweets) and '@udzura'. There's also a 'フォロー' (Follow) button at the bottom right of the profile card.

## FastContainerアーキテクチャの採用

マネージドクラウドのコンテナは以下の“リクエスト契機”で動作します

- ・HTTP リクエストを契機にしてコンテナが起動する
- ・コンテナは一定のライフタイムの間（20分）、起動する
- ・ライフタイムが過ぎたらコンテナは停止する

アーキテクチャを論ずる大事なポイントが他にもたくさんありますが、このスライドでは上記の3点をおさえておくことが理解のキーポイントになります。

FastContainerアーキテクチャ概論 / 松本亮介

[https://speakerdeck.com/matsumoto\\_r/reactive-stateless-and-mortal-architecture-for-web-applications](https://speakerdeck.com/matsumoto_r/reactive-stateless-and-mortal-architecture-for-web-applications)

FastContainer: 実行環境の変化に素早く適応できる。恒常性を持つシステムアーキテクチャ。松本亮介 1.a)、近藤宇智朗 2.、三宅悠介 1.、力武健次 1,3.、栗林健太郎 1.

<https://rand.pepabo.com/papers/iotics2017-matsumotory.pdf>

How Ruby Survives in the Cloud Native World / Uchio Kondo

<https://speakerdeck.com/udzura/how-ruby-survives-in-the-cloud-native-world?slide=26>

GMOペパボ

FastContainerアーキテクチャ概論

Reactive, Stateless and Mortal Architecture for Web Applications

松本亮介 / ペパボ研究所  
GMOペパボ ホスティング技術カンファレンス  
～破壊的イノベーションをおこす革新的技術～

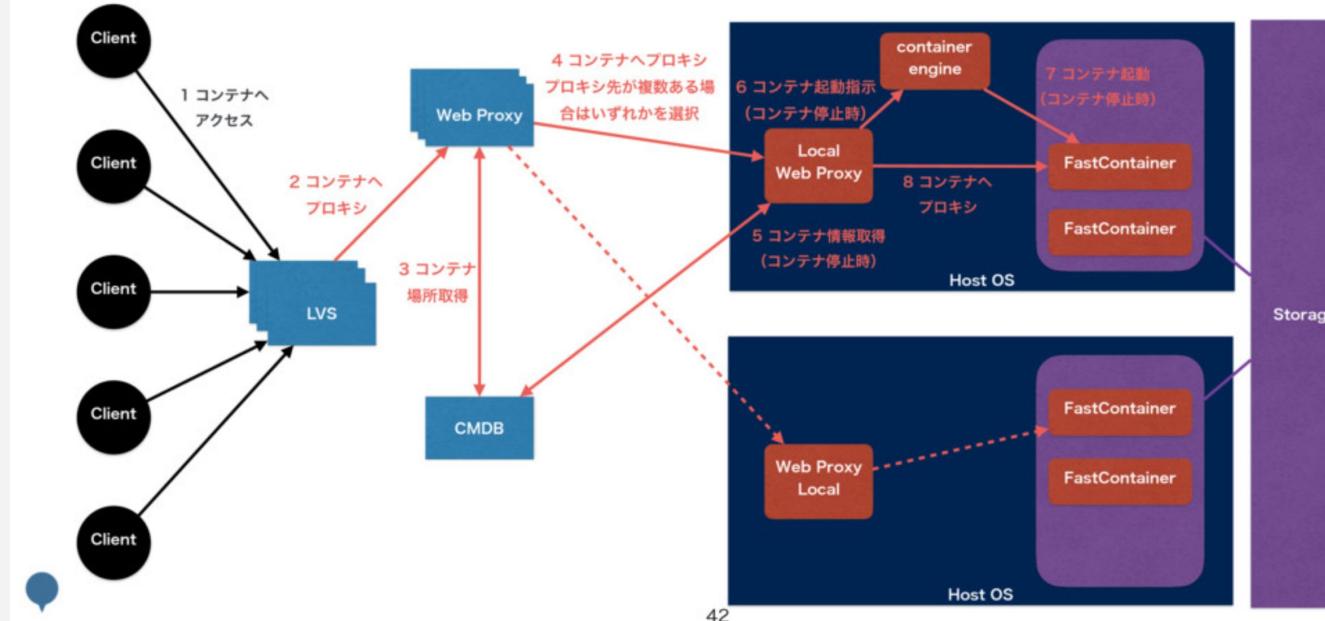
How Ruby Survives

...in the World of the Cloud-Native

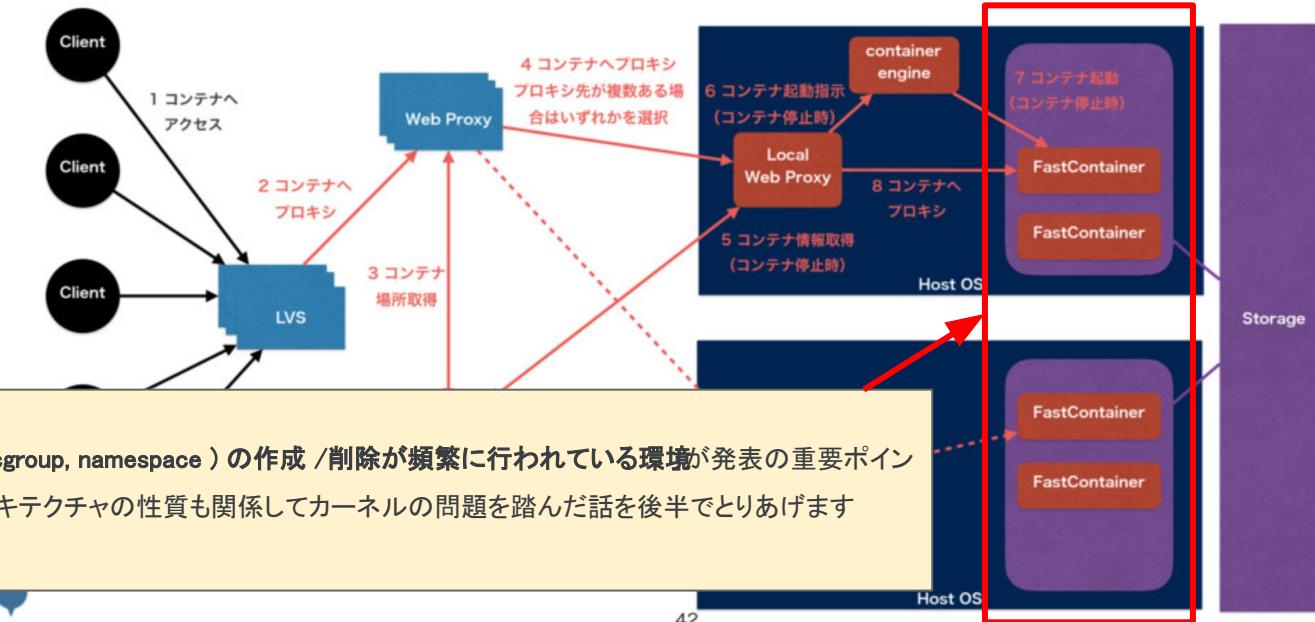
Uchio Kondo / GMO Pepabo, Inc.  
RubyKaigi 2018

GMOペパボ研究所

# FastContainerフロー概要



# FastContainerフロー概要



## Section 2

# cgroup と sysfs ファイル

ロリポップ！マネージドクラウドの話は一旦終わります。  
次に cgroup と sysfs ファイルの話に入ります

## 本スライドでの検証環境

macOS + Vagrant + Ubuntu Bionic に v5.3.9 mainline build のカーネルをインストールしています

```
vagrant@ubuntu-bionic:~$ uname -a
Linux ubuntu-bionic 5.3.9-050309-generic #201911061337 SMP Wed Nov 6 13:39:34 UTC 2019 x86_64 x86_64
x86_64 GNU/Linux
```

後半でカーネルの問題に遭遇した過去事例を紹介します。2019年11月時点で、このバージョンを用いた環境で問題に遭いました。

## /sys/fs/cgroup

cgroup やコンテナの内部に興味ある方、おそらく /sys/fs/cgroup はみたことがあるかな？（無くても大丈夫）

```
vagrant@ubuntu-bionic:~$ ls /sys/fs/cgroup/
blkio  cpu  cpu,cpuacct  cpusacct  cpuset  devices  freezer  hugetlb  memory  net_cls
net_cls,net_prio  net_prio  perf_event  pids  rdma  systemd  unified
```

⚠️ /sys/fs/cgroup に cgroupfs がマウントされている前提で話をしています！

しかし、今日は /sys/fs/cgroup ではなく /sys/kernel/slab の下をみていきます！

## /sys/kernel/slab

たくさんのディレクトリが並びます。ディレクトリ名は **スラブキャッシュ** の名前です

```
vagrant@ubuntu-bionic:~$ ls /sys/kernel/slab
:0000016 :A-0001344      biovec-64
:0000024 :A-0004928      biovec-max
:0000032 :a-0000016      blkdev_ioc
:0000040 :a-0000032      btree_node
:0000048 :a-0000040      btrfs_delayed_data_ref
:0000056 :a-0000048      btrfs_delayed_extent_op
:0000064 :a-0000056      btrfs_delayed_node
:0000072 :a-0000064      btrfs_delayed_ref_head
:0000080 :a-0000104      btrfs_delayed_tree_ref
:0000088 :a-0000108      btrfs_end_io_wq
:0000104 :a-0000144      btrfs_extent_buffer
:0000112 :a-0000256      btrfs_extent_map
:0000128 :d-0000008      btrfs_extent_state
:0000144 :d-0000016      btrfs_free_space
:0000184 :d-0000032      btrfs_free_space_bitmap
:0000192 :d-0000064      btrfs_inode
                           encryptfs_inode_cache    isofs_inode_cache    pde_opener
                           encryptfs_key_record_cache jbd2_inode        pid
                           encryptfs_key_sig_cache   jbd2_journal_handle pid_namespace
                           encryptfs_key_tfm_cache   jbd2_journal_head   pool_workqueue
                           encryptfs_sb_cache       jbd2_revoke_record_s posix_timers_cache
                           encryptfs_xattr_cache    jbd2_revoke_table_s  proc_dir_entry
                           eventpoll_ep1            jbd2_transaction_s  proc_inode_cache
                           eventpoll_pwq            kcopyd_job         radix_tree_node
                           ext4_allocation_context kernfs_iattrs_cache request_queue
                           ext4_bio_post_read_ctx  kernfs_node_cache  request_sock_TCP
                           ext4_extent_status       key_jar           request_sock_TCPv6
                           ext4_free_data           khugepaged_mm_slot scsi_data_buffer
                           ext4_groupinfo_4k         kioctx            scsi_sense_cache
                           ext4_inode_cache          kmalloc-128       sd_ext_cdb
                           ext4_io_end               kmalloc-16        seq_file
                           ext4_pending_reservation kmalloc-192       sgpool-128

...
... snip
```

⚠ SLUB アロケータについては下記のブログエントリ が非常に参考になります。感謝

φ(・・\*)ゞ ウーン カーネルとか弄ったりのメモ Linuxカーネル4.1のSLUBアロケータ(ドラフト) <https://kernhack.hatenablog.com/entry/2019/05/10/001425>

## /sys/kernel/slab/\*/cgroup

さらにサブディレクトリをみてみましょう。cgroup のディレクトリが見つかります。

```
vagrant@ubuntu-bionic:~$ ls -d /sys/kernel/slab/*/cgroup
/sys/kernel/slab/:0000016/cgroup
/sys/kernel/slab/:0000024/cgroup
/sys/kernel/slab/:0000032/cgroup
/sys/kernel/slab/:0000040/cgroup
/sys/kernel/slab/:0000048/cgroup
/sys/kernel/slab/:0000056/cgroup
/sys/kernel/slab/:0000064/cgroup
/sys/kernel/slab/:0000072/cgroup
/sys/kernel/slab/:0000080/cgroup
/sys/kernel/slab/:0000088/cgroup
/sys/kernel/slab/:0000104/cgroup
/sys/kernel/slab/:0000112/cgroup
/sys/kernel/slab/:0000128/cgroup
/sys/kernel/slab/:0000144/cgroup
/sys/kernel/slab/:0000184/cgroup
/sys/kernel/slab/:0000192/cgroup
/sys/kernel/slab/:0000200/cgroup
/sys/kernel/slab/:0000208/cgroup
/sys/kernel/slab/:0000216/cgroup
/sys/kernel/slab/:0000256/cgroup
/sys/kernel/slab/:0000264/cgroup
... snip
          /sys/kernel/slab/biovec-16/cgroup
          /sys/kernel/slab/biovec-64/cgroup
          /sys/kernel/slab/biovec-max/cgroup
          /sys/kernel/slab/blkdev_ioc/cgroup
          /sys/kernel/slab/btree_node/cgroup
          /sys/kernel/slab/btrfs_delayed_data_ref/cgroup
          /sys/kernel/slab/btrfs_delayed_extent_op/cgroup
          /sys/kernel/slab/btrfs_delayed_node/cgroup
          /sys/kernel/slab/btrfs_delayed_ref_head/cgroup
          /sys/kernel/slab/btrfs_delayed_tree_ref/cgroup
          /sys/kernel/slab/btrfs_end_io_wq/cgroup
          /sys/kernel/slab/btrfs_extent_buffer/cgroup
          /sys/kernel/slab/btrfs_extent_map/cgroup
          /sys/kernel/slab/btrfs_extent_state/cgroup
          /sys/kernel/slab/btrfs_free_space/cgroup
          /sys/kernel/slab/btrfs_free_space_bitmap/cgroup
          /sys/kernel/slab/btrfs_inode/cgroup
          /sys/kernel/slab/btrfs_inode_defrag/cgroup
          /sys/kernel/slab/btrfs_ordered_extent/cgroup
          /sys/kernel/slab/btrfs_path/cgroup
          /sys/kernel/slab/btrfs_prelim_ref/cgroup
          /sys/kernel/slab/ip_mrt_cache/cgroup
          /sys/kernel/slab/iser_descriptors/cgroup
          /sys/kernel/slab/isofs_inode_cache/cgroup
          /sys/kernel/slab/jbd2_inode/cgroup
          /sys/kernel/slab/jbd2_journal_handle/cgroup
          /sys/kernel/slab/jbd2_journal_head/cgroup
          /sys/kernel/slab/jbd2_revoke_record_s/cgroup
          /sys/kernel/slab/jbd2_revoke_table_s/cgroup
          /sys/kernel/slab/jbd2_transaction_s/cgroup
          /sys/kernel/slab/kcopyd_job/cgroup
          /sys/kernel/slab/kernfe_iattr_cache/cgroup
          /sys/kernel/slab/kernfs_node_cache/cgroup
          /sys/kernel/slab/key_jar/cgroup
          /sys/kernel/slab/khugepaged_mm_slot/cgroup
          /sys/kernel/slab/kioctx/cgroup
          /sys/kernel/slab/kmalloc-128/cgroup
          /sys/kernel/slab/kmalloc-16/cgroup
          /sys/kernel/slab/kmalloc-192/cgroup
          /sys/kernel/slab/kmalloc-1k/cgroup
          /sys/kernel/slab/kmalloc-256/cgroup
          /sys/kernel/slab/kmalloc-2k/cgroup
```

## /sys/kernel/slab/\$スラブキャッシュ名/cgroup の中身

cgroup 以下をみてみましょう。試しに \$スラブキャッシュ名 = dentry のディレクトリをみてみましょう

```
vagrant@ubuntu-bionic:~$ ls -hal /sys/kernel/slab/dentry/cgroup/
total 0
drwxr-xr-x 49 root root 0 Oct 14 13:19 .
drwxr-xr-x  3 root root 0 Oct 14 13:19 ..
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(102:sys-kernel-debug.mount)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(128:keyboard-setup.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(141:kmod-static-nodes.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(154:dev-hugepages.mount)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(180:systemd-journald.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(193:lvm2-monitor.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(206:cloud-init-local.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(219:systemd-random-seed.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(232:systemd-tmpfiles-setup-dev.service)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(245:sys-fs-fuse-connections.mount)'
drwxr-xr-x  2 root root 0 Oct 14 13:26 'dentry(24:user.slice)'

... snip
```

systemd が扱う cgroup (スライス) の名前が入ったサブディレクトリが並んでいます。

# ディレクトリパスのフォーマット

```
/sys/kernel/slab/$スラブキャッシュの名前/cgroup/$スラブキャッシュの名前($シリアル番号:$cgroupの名前)
```

\$シリアル番号 は、`struct cgroup_subsys_state` の `serial_nr` を反映していて 一意に採番されるようです

`mm/slab_common.c` の `memcg_create_kmem_cache()` に実装が書いてあります

```
void memcg_create_kmem_cache(struct mem_cgroup *memcg,
                           struct kmem_cache *root_cache)
{
    ...
    cgroup_name(css->cgroup, memcg_name_buf, sizeof(memcg_name_buf));
    cache_name = kasprintf(GFP_KERNEL, "%s(%llu:%s)", root_cache->name,
                          css->serial_nr, memcg_name_buf); ↗
```

/sys/kernel/slab/\$スラブキャッシュ名/cgroup に新しいディレクトリを生やす

1. 適当な memory cgroup を作成して bash プロセスを追加してみましょう

```
root@ubuntu-bionic:~# mkdir      /sys/fs/cgroup/memory/test000
root@ubuntu-bionic:~# echo $$ > /sys/fs/cgroup/memory/test000/tasks
```

2. /sys/kernel/slab/dentry/cgroup/\* に test000 の名前を含むサブディレクトリが追加されました

```
root@ubuntu-bionic:~# ls -halld /sys/kernel/slab/dentry/cgroup/* | grep test000
drwxr-xr-x 2 root root 0 Oct 13 07:43 /sys/kernel/slab/dentry/cgroup/dentry(887:test000)
```

## その他の /sys/kernel/slab/\*/cgroup/\*

例に挙げた deny 以外にも /sys/kernel/slab/\*/cgroup/\* 以下に test000 の名前を含むサブディレクトリが追加されています。

```
root@ubuntu-bionic:~# ls /sys/kernel/slab/*/cgroup/ | grep test000 | sort -u
anon_vma(3680:test000)
anon_vma_chain(3680:test000)
cred_jar(3680:test000)
dentry(3680:test000)
ext4_inode_cache(3680:test000)
files_cache(3680:test000)
filp(3680:test000)
inode_cache(3680:test000)
kmalloc-192(3680:test000)
kmalloc-1k(3680:test000)
kmalloc-2k(3680:test000)
kmalloc-32(3680:test000)
kmalloc-4k(3680:test000)
kmalloc-64(3680:test000)
kmalloc-96(3680:test000)
mm_struct(3680:test000)
pde_opener(3680:test000)
pid(3680:test000)

... snip
```

/sys/kernel/slab/\$キッシュ名/cgroup/\$キャッシュ名(\$シリアル番号:\$cgroupの名前) の中身

ディレクトリ以下に下記のようなファイルが並びます

```
root@ubuntu-bionic:~# ls -hal /sys/kernel/slab/dentry/cgroup/dentry\((887\:test000\)/
total 0
drwxr-xr-x  2 root root    0 Oct 12 07:42 .
drwxr-xr-x 47 root root    0 Oct 12 07:42 ..
-r-----  1 root root 4.0K Oct 12 07:43 aliases
-r-----  1 root root 4.0K Oct 12 07:43 align
-r-----  1 root root 4.0K Oct 12 07:43 alloc_calls
-r-----  1 root root 4.0K Oct 12 07:43 cache_dma
-rw-----  1 root root 4.0K Oct 12 07:43 cpu_partial
-r-----  1 root root 4.0K Oct 12 07:43 cpu_slabs
-r-----  1 root root 4.0K Oct 12 07:43 ctor
-r-----  1 root root 4.0K Oct 12 07:43 destroy_by_rcu
-r-----  1 root root 4.0K Oct 12 07:43 free_calls
-r-----  1 root root 4.0K Oct 12 07:43 hwcache_align
-rw-----  1 root root 4.0K Oct 12 07:43 min_partial
-r-----  1 root root 4.0K Oct 12 07:43 object_size
-r-----  1 root root 4.0K Oct 12 07:43 objects
-r-----  1 root root 4.0K Oct 12 07:43 objects_partial

... snip
```

## /sys/kernel/slab 以下のファイルの用途は?

カーネルのドキュメント Documentation/ABI/testing/sysfs-kernel-slab に説明が載っています

```
What:      /sys/kernel/slab
Date:      May 2007
KernelVersion: 2.6.22
Contact:   Pekka Enberg <penberg@cs.helsinki.fi>,
           Christoph Lameter <c1@linux-foundation.org>
Description:
The /sys/kernel/slab directory contains a snapshot of the
internal state of the SLUB allocator for each cache. Certain
files may be modified to change the behavior of the cache (and
any cache it aliases, if any).
Users:      kernel memory tuning tools
```

“/sys/kernel/slab ディレクトリは各スラブキャッシュのSLUB アロケータの内部状態のスナップショットを含んでいます。特定のファイルを変更することで、キャッシュの動作を変更することができます。”

/sys/kernel/slab/\$キャッシュ名/cgroup/\$キャッシュ名(\$シリアル番号:\$cgroup名)以下も同様です

/sys/kernel/slab/\$キャッシュ名/cgroup/\$キャッシュ名(\$シリアル番号:\$cgroup名) とカーネルコンフィグ

これらの sysfs ファイルは SLUB\_MEMCG\_SYSFS\_ON が有効な時に作成されるファイルです

```
config SLUB_MEMCG_SYSFS_ON
    default n
    bool "Enable memcg SLUB sysfs support by default" if EXPERT
    depends on SLUB && SYSFS && MEMCG
    help
        SLUB creates a directory under /sys/kernel/slab for each
        allocation cache to host info and debug files. If memory
        cgroup is enabled, each cache can have per memory cgroup
        caches. SLUB can create the same sysfs directories for these
        caches under /sys/kernel/slab/CACHE/cgroup but it can lead
        to a very high number of debug files being created. This is
        controlled by slub_memcg_sysfs boot parameter and this
        config option determines the parameter's default value.
```

ブートパラメータの slub\_memcg\_sysfs を指定することで、sysfs ファイルを作成しないように挙動を変えられます  
このブートパラメータは後半で再び取り上げます

スラブキャッシュが /sys/kernel/slab/\$キャッシュ名/cgroup/\$キャッシュ名 (\$シリアル番号:\$cgroup名) を作る

```
config SLUB_MEMCG_SYSFS_ON
    default n
    bool "Enable memcg SLUB sysfs support by default" if EXPERT
    depends on SLUB && SYSFS && MEMCG
    help
        SLUB creates a directory under /sys/kernel/slab for each
        allocation cache to host info and debug files. If memory
        cgroup is enabled, each cache can have per memory cgroup
        caches. SLUB can create the same sysfs directories for these
        caches under /sys/kernel/slab/CACHE/cgroup but it can lead
        to a very high number of debug files being created. This is
        controlled by slub_memcg_sysfs boot parameter and this
        config option determines the parameter's default value.
```

これまで記述した「memory cgroupを作ると /sys/kernel/slab/\*/\* のファイルができる」は正確な記述ではありません  
「memory cgroup に所属するプロセスが動作しカーネル内部でスラブキャッシュが作成されるタイミングで /sys/kernel/slab/\*/\* を作る」と  
いうのが正確な説明となるでしょう。そのためのコンフィグです。

しかしながら、ユーザ空間からスラブキャッシュの動作は観測が難しく、説明を簡素にするために便宜的に前者の表現を用いています。

## Section 3

# sysfs ファイル と uevent



memory cgroup と sysfs ファイル の話をしました。  
次に、sysfs ファイルと uevent の話をします。

sysfs ファイルが作成されると uevent が送出される

- ・/sys/kernel/slab/\$キャッシュ名/cgroup/\$キャッシュ名(\$シリアル番号:\$cgroup) が作成される際、カーネルは uevent を送出します。(\*)
- ・memory cgroup を削除し、上記の sysfs ファイルも削除される際に uevent を送出します

⚠️ 正確にはカーネル内部で、cgroup 内のスラブキャッシュが生成/削除されるタイミングで sysfs ファイルが作成/削除され uevent が飛ぶ。ユーザ空間からみた視点（≒ 観測しやすい）で記述しています。

## memory cgroup を作って uevent を見てみる

1. 適当な memory cgroup を作成し、bash プロセスで touch コマンドを起動します

```
root@ubuntu-bionic:~# mkdir      /sys/fs/cgroup/memory/test000
root@ubuntu-bionic:~# echo $$ > /sys/fs/cgroup/memory/test000/tasks

# ここで uevent が発生します 🔥
root@ubuntu-bionic:~# touch /
```

2. 同時に、別のターミナルで udevadm monitor --kernel 実行してイベント ⚡ を捉えます

```
vagrant@ubuntu-bionic:~$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[20158.423918] add      /kernel/slab/:A-0000192/cgroup/cred_jar(3960:test000) (cgroup)
KERNEL[20158.424065] add      /kernel/slab/kmalloc-64/cgroup/kmalloc-64(3960:test000) (cgroup)
KERNEL[20158.424087] add      /kernel/slab/inode_cache/cgroup/inode_cache(3960:test000) (cgroup)
KERNEL[20158.424104] add      /kernel/slab/kmalloc-192/cgroup/kmalloc-192(3960:test000) (cgroup)
KERNEL[20158.424125] add      /kernel/slab/kmalloc-1k/cgroup/kmalloc-1k(3960:test000) (cgroup)
KERNEL[20158.424208] add      /kernel/slab/:A-0000256/cgroup/filp(3960:test000) (cgroup)
KERNEL[20158.426278] add      /kernel/slab/ext4_inode_cache/cgroup/ext4_inode_cache(3960:test000) (cgroup)

... snip
```

## memory cgroup 削除時の uevent を見てみる

1. memory cgroup の削除時にも uevent が飛びます

```
root@ubuntu-bionic:~# rmdir /sys/fs/cgroup/memory/test000
```

2. 先と同様に、別のターミナルで udevadm monitor --kernel を実行してイベント ⚡ を捉えられます

```
vagrant@ubuntu-bionic:~$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[20456.994677] remove   /kernel/slab/ext4_inode_cache/cgroup/ext4_inode_cache(3960:test000) (cgroup)
KERNEL[20456.994753] remove   /kernel/slab/:A-0000080/cgroup/task_delay_info(3960:test000) (cgroup)
KERNEL[20456.994774] remove   /kernel/slab/:A-0000256/cgroup/filp(3960:test000) (cgroup)
KERNEL[20456.994791] remove   /kernel/slab/inode_cache/cgroup/inode_cache(3960:test000) (cgroup)
KERNEL[20456.994807] remove   /kernel/slab/:A-0000208/cgroup/vm_area_struct(3960:test000) (cgroup)
KERNEL[20456.994828] remove   /kernel/slab/mm_struct/cgroup/mm_struct(3960:test000) (cgroup)
KERNEL[20456.994844] remove   /kernel/slab/:A-0000704/cgroup/files_cache(3960:test000) (cgroup)
KERNEL[20456.994859] remove   /kernel/slab/:A-0001088/cgroup/signal_cache(3960:test000) (cgroup)
KERNEL[20456.995052] remove   /kernel/slab/anon_vma/cgroup/anon_vma(3960:test000) (cgroup)
KERNEL[20456.995080] remove   /kernel/slab/:A-0000128/cgroup/pid(3960:test000) (cgroup)

... snip
```

そもそも uevent とは何ですか？

デバイスをホットプラグした際に、カーネルからユーザ空間に通知を出しデバイスファイル /dev をコントロールするための仕組み *udev = userspace device management* に使う「イベント」と理解しています。

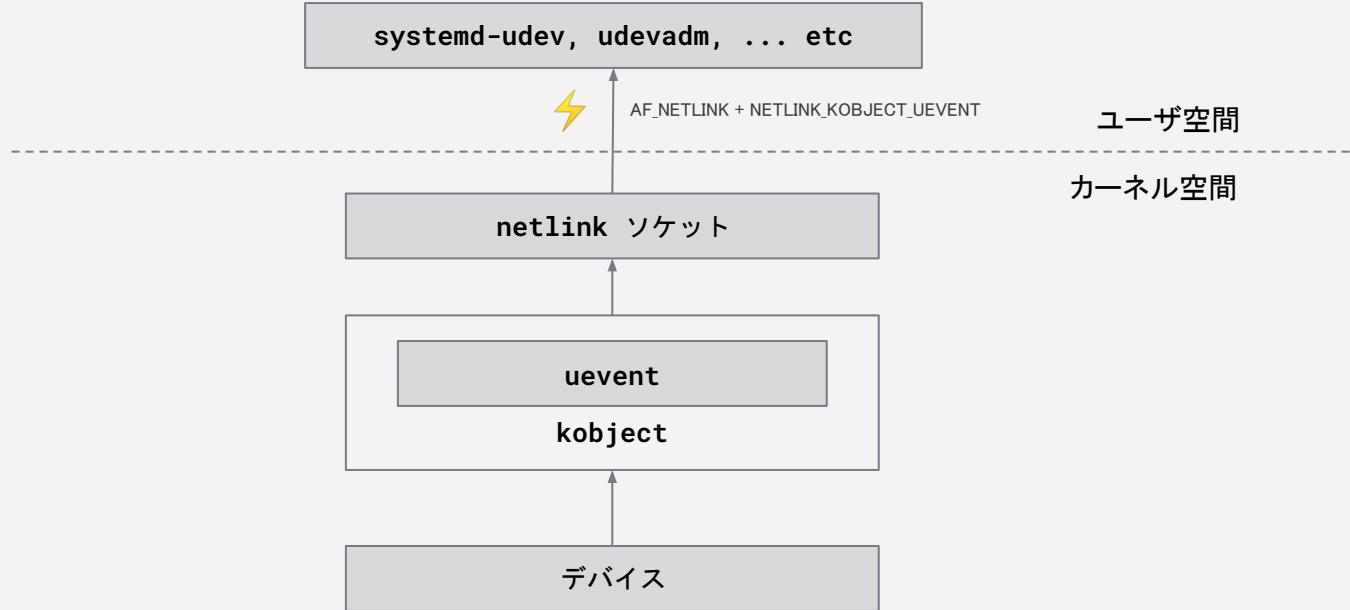
“USB や SDIO のように稼動中の抜き差し(ホットプラグ)に対応したバスの場合、新しいデバイスの挿入検出ごとに UEVENT と呼ばれるカーネルイベントが発生します”

Wiress・おと Linuxドライバのはなし <http://www.silex.jp/blog/wireless/2015/08/linux.htm>より引用

“その比較的新しめの 2.6 カーネルでは、デバイスの存在を認識したり、デバイスがなくなったりしたときに、netlink を介して、ユーザにその旨を通知してくれるようになりました。”

いますぐ実践 Linux システム管理/ Vol.114 <http://www.usupi.org/sysad/114.htm>より引用

## uevent のモデル図



cgroup でプロセスを動かした時の uevent はどこで出している?

スラブキャッシュ作成 → sysfs ファイルを作成する処理内で kobject\_uevent() で出します

```
static int sysfs_slab_add(struct kmem_cache *s)
{
...
#endif CONFIG_MEMCG
    if (is_root_cache(s) && memcg_sysfs_enabled) {
        s->memcg_kset = kset_create_and_add("cgroup", NULL, &s->kobj);
        if (!s->memcg_kset) {
            err = -ENOMEM;
            goto out_del_kobj;
        }
    }
#endif

    kobject_uevent(&s->kobj, KOBJ_ADD); ➔
```

#### 呼び出しの順番

```
kmem_cache_alloc
-> slab_alloc
-> slab_pre_alloc_hook
-> memcg_kmem_get_cache
-> memcg_schedule_kmem_cache_create
```

(work\_queue で非同期処理)

```
memcg_kmem_cache_create_func
-> memcg_create_kmem_cache
-> create_cache
-> __kmem_cache_create
-> sysfs_slab_add ➔
```

⚠ cgroup のスラブキャッシュ作成は kmem\_cache\_create() ではなく kmem\_cache\_alloc() の延長で行われます

⚠ uevent を説明するにあたって sysfs と kobject, kset の説明もあるとよいのですが ボリュームが大きくなりすぎるので割愛します

cgroup 削除時の uevent はどこで出している?

cgroup に属するスラブキャッシュを削除 -> sysfs ファイルを削除 -> 途中で kobject\_uevent() を呼ぶ

```
static void sysfs_slab_remove_workfn(struct work_struct *work)
{
    struct kmem_cache *s =
        container_of(work, struct kmem_cache, kobj_remove_work);
    ...
#define KOBJ_REMOVE 0x00000001

#endif /* CONFIG_MM_SLAB */
#ifdef CONFIG_MM_SLAB
    kmem_cache_destroy(s);
#endif
    kobject_uevent(&s->kobj, KOBJ_REMOVE); ➔
out:
    kobject_put(&s->kobj);
}
```

## 呼び出しの順番

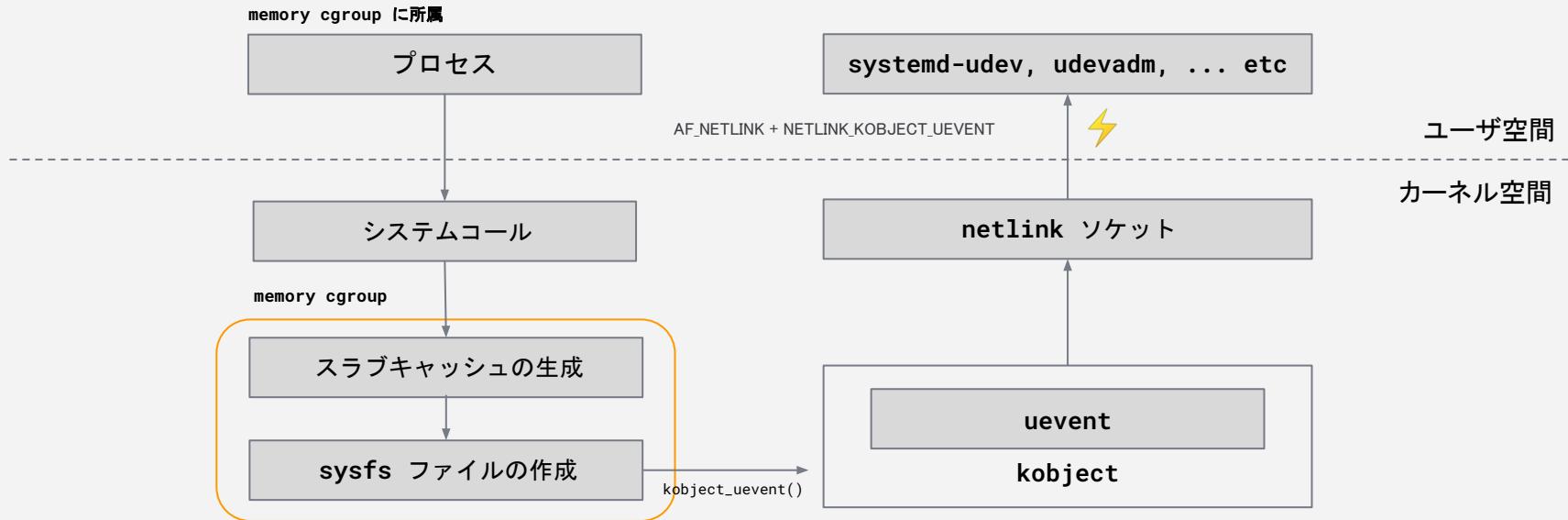
```
kmem_cache_destroy
-> shutdown_memcg_caches
-> shutdown_cache
-> __kmem_cache_shutdown
-> sysfs_slab_remove

(workqueue で非同期処理)

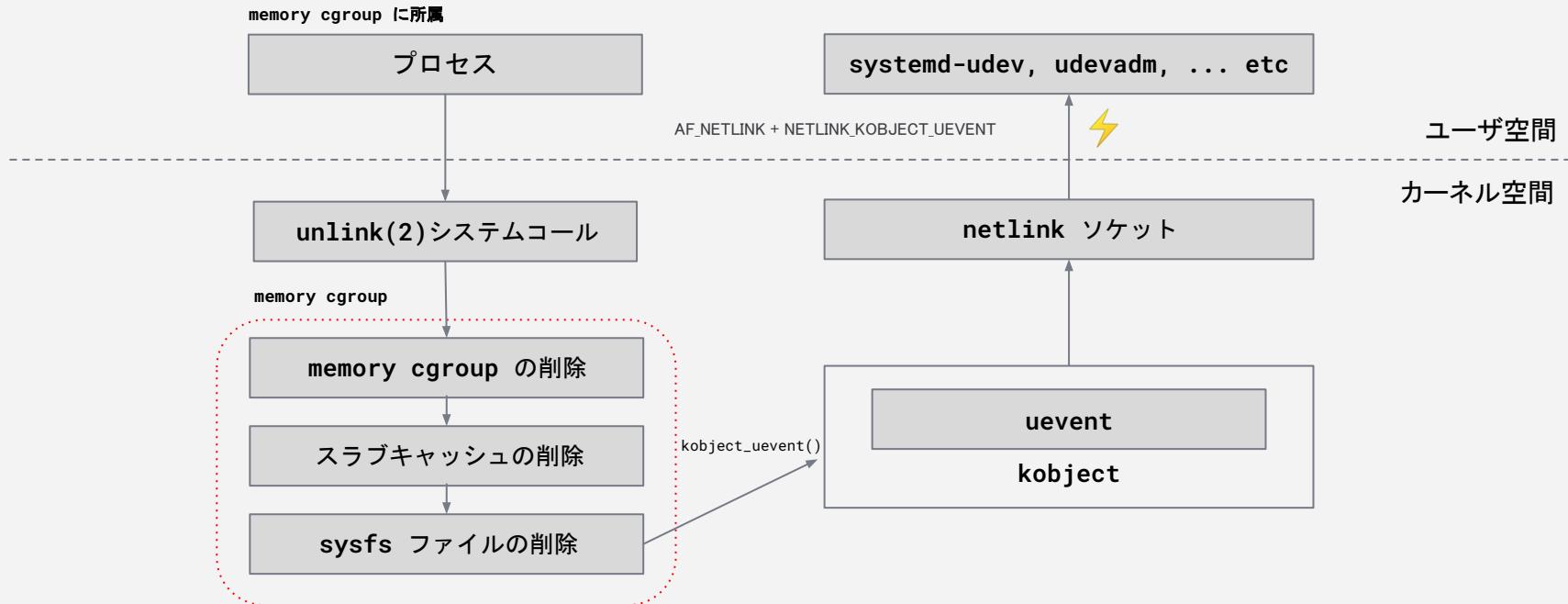
-> sysfs_slab_remove_workfn ➔
```

⚠ uevent を正確に理解するにあたって sysfs と kobject, kset の説明もあるとよいのですが ポリュームが大きくなりすぎるので割愛します

memory cgroup と uevent のフロー モデル ( cgroup 作成 -> プロセス動作時 )



## memory cgroup と uevent のフローモデル（ cgroup 削除時 ）



uevent はどんな情報を送ってくるのか?

uevent の詳細は **--property** を付けて確認できます

```
vagrant@ubuntu-bionic:~$ udevadm monitor --property
...
KERNEL[1973.328329] add      /kernel/slab/dentry/cgroup/dentry(884:test000) (cgroup)
ACTION=add
DEVPATH=/kernel/slab/dentry/cgroup/dentry(884:test000)
SEQNUM=4583
SUBSYSTEM=cgroup
```

新しく作られたスラブキャッシュの名前と sysfs のパスが分かるんだな。ふーん??? 😐

uevent のユースケースは?

「cgroup のスラブキャッシュが作成/削除されるタイミングで uevent が飛ぶの、何に使うんだろう?」

... と思った方はいませんか?

- ・1年前、この問題を調べていた際に、私はそう思いました!
- ・cgroup 内のスラブキャッシュの作成と削除をトリガーとして、ユーザ空間で何かフックする処理はあるかな?
- ・デバイスのホットプラグとは関係ないし
- ・カーネルのデバッグ用途には使えそうだ?

真相はこの後で説明いたします!

このスライドを書くのには「役にたっている」🤔 いや、そういう話ではない。

## Section 4

## トラブルシューティング事例

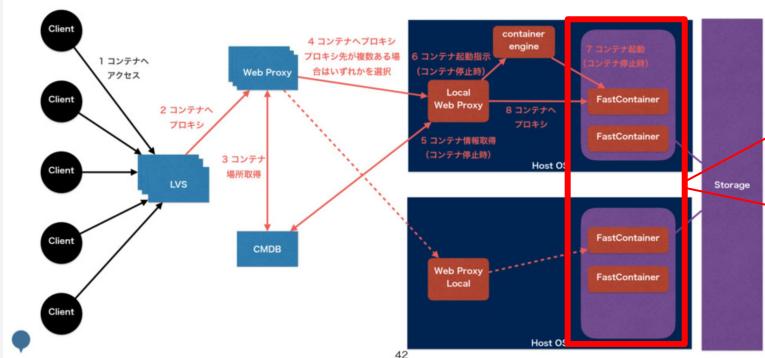


次に、ロリポップ！マネージドクラウドで実際に遭遇した問題を紹介します  
ここまでに話をした sysfs ファイルや uevent の話が出てきます。

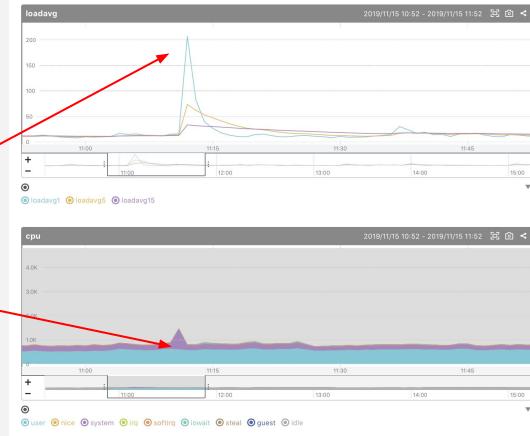
謎のロードアベレージ上昇アラートを追った（2019年10月～11月ごろの話）

コンテナを動作させているホストで、時折 ロードアベレージと %system cpu 上昇しアラートが飛んでいた  
偶発的にアラートが出るもの、サービスの可用性には影響しておらず、根本原因の調査が進んでいなかった

## FastContainerフロー概要



42



⚠️ 当時の時系列順ではなく、説明を容易にするためにブレイクダウンして再構成した順序でスライドを構成しています

# トラブルシューティング

GMOペパボ

メトリクス (mackerel) で %system と ロードアベレージが上昇した様子

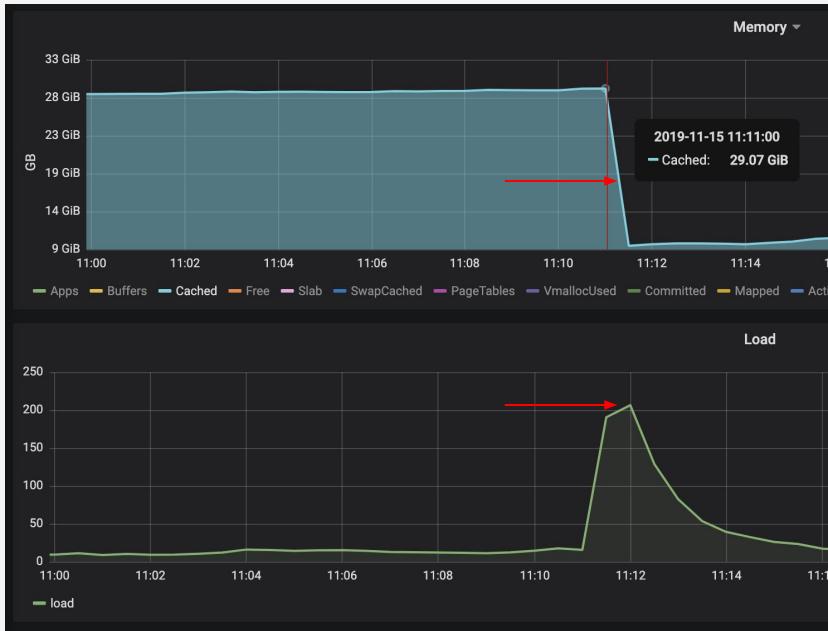


mackerel は 株式会社はてなの監視サービスです <https://mackerel.io/ja/>

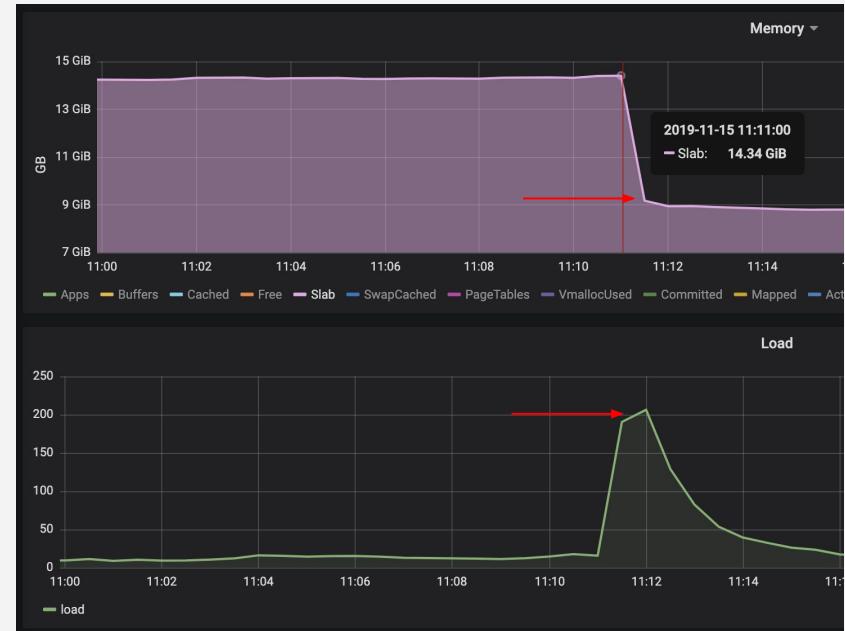
# トラブルシューティング

GMOペパボ

アラート発生時の Memory (Cached) と LoadAverage



アラート発生時の Memory (Slab) と LoadAverage



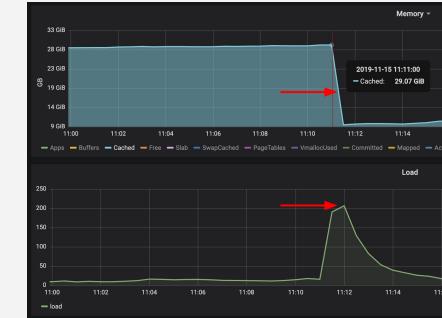
ん、LA 上昇と同時に cached と slab がガクンと減ってるな? 😞

なんと `sysctl vm.drop_caches` が動いていた！

- ・一定のメモリが使用されると `sysctl vm.drop_caches` する cron で実行 ⏳
- ・過去に踏んだ問題のワークアラウンド(\*)として入れていた
- ・チームの同僚も `sysctl vm.drop_caches` が怪しそうと指摘していた

トリガーとなる事象はチームでも既知。

- ・では、なぜこれほどのロードアベレージの上昇を招くのか？
- ・毎回必ずアラートを出している訳でもなかった



⚠️ 理由は後述します

⚠️ production 環境で `vm.drop_caches` はなあ… はその通りでしょう。正直に失敗例を公開する代わりに、優しくしてください

## ⚠ sysctl vm.drop\_caches のワークアラウンドを必要とした背景 → unshare(2) のボトルネック回避

- ・コンテナ作成時に namespace を作るために unshare(2) を呼び出す際に copy\_net\_ns() の呼び出しで deactivate\_slab() が走るとコンテナ起動時のボトルネックとなり障害レベルの遅延を招いていた。リクエスト契機でコンテナ作成( cgroup / namespace を作る) FastContainer アーキテクチャと相性が悪い問題が存在していた。サービスの安定稼働を優先にして、sysctl vm.drop\_caches でメモリの Free を確保しておくことで deactivate\_slab() を回避していました。
- ・その後、改修が入り namespace を事前作成する方式に移行し unshare(2) のボトルネックを回避しました。、ワークアラウンドの sysctl vm.drop\_caches が残つたままになってしまったという顛末です。
- ・詳しくは “高集積コンテナホスティングにおけるボトルネックとその解法 - @pyama86”

<https://speakerdeck.com/pyama86/gao-ji-ji-kontenahosuteinguniokerubotorunetukutosofalsejie-fa> をご参照ください

GMOペパボ

### 高集積コンテナホスティングにおけるボトルネックとその解法

～持て余してるフルストレーション、You've got an easy day～

@pyama86 / GMO Pepabo, Inc.  
2018.09.08 builderscon tokyo 2018

えいっ！

```
$ sync; echo 3 > /proc/sys/vm/drop_caches
```

slabtopで確認したところ、dentryキャッシュが多かったため、キャッシングのクリアが有効と考えた

### Slabは肥大化するとボトルネックとなる

一般的にはSlabの肥大化に起因してSwapメモリが利用され、システム全体が爆速になることが多いが、単純に使用量が多いと初期化処理時のロックでボトルネックとなることがある。

またこれらのOSレイヤの仕組みの調査においてはstraceやperfを利用することでたりをつけ、ソースを読み込むことで原因に対する仮説を立てることができる。

アラートが発生した際に採取した ps の結果（一部）

STATE D で wchan が kobject\_uevent\_env でブロックしている kworker が大量に観測された

```
$ ps ax -L -opid,state,wchan:40,cmd:100
...
41290 D kmemcg_workfn [kworker/2:31+me]
41413 D kobject_uevent_env [kworker/19:60+e]
41466 D kobject_uevent_env [kworker/19:73+e]
41470 D kobject_uevent_env [kworker/19:74+e]
41682 D kobject_uevent_env [kworker/46:47+e]
41752 D kmemcg_workfn [kworker/19:126+]
41799 D kobject_uevent_env [kworker/19:130+]
41835 D kmemcg_workfn [kworker/14:65+m]
42080 D kobject_uevent_env [kworker/14:94+e]

...
... snip
```

調査にはCPU使用率やロードアベレージ等をトリガーとして、任意のコマンドを指定回数トリガー実行できるツールを用い@k1low <https://github.com/k1LoW/sheer-heart-attack>

アラートが発生した際に採取できた ps の結果 -> 集計すると kobject\_uevent\_env だけ

STATE D で wchan が kobject\_uevent\_env でブロックする kworker が大量に観測された

```
$ cat ps.txt | grep ' D ' | awk '{ print $3 }' | sort | uniq -c | sort -rn
135 kobject_uevent_env ↗
 10 kmemcg_workfn
   6 rpc_wait_bit_killable
   1 lock_page_killable
```

アラートが発生している際に取った echo 1 > /proc/sysrq-trigger のログ  
実行中のすべてのタスクのバックトレースがdmesg に出来ます

```
Nov 25 21:51:13 hostname kernel: NMI backtrace for cpu 35
Nov 25 21:51:13 hostname kernel: CPU: 35 PID: 30519 Comm: kworker/35:28 Kdump: loaded Not tainted 5.3.9-050309-generic #201911061337
Nov 25 21:51:13 hostname kernel: Hardware name: Dell Inc. PowerEdge ***/***, BIOS ***
Nov 25 21:51:13 hostname kernel: Workqueue: events sysfs_slab_remove_workfn ↗
Nov 25 21:51:13 hostname kernel: RIP: 0010:netlink_has_listeners+0xc/0x60 ↗
Nov 25 21:51:13 hostname kernel: Code: 41 bc ea ff ff e9 b8 fe ff 31 f6 eb 9f 66 66 2e 0f 1f 84 00 00 00 00 00 00 0f 1f 40 00 0f 1f 44 00 00 55 8b
87 04 03 00 00 <f7> d0 48 89 e5 83 e0 01 75 3d 0f b6 97 11 02 00 00 48 8d 0c 52 48
Nov 25 21:51:13 hostname kernel: RSP: 0018:fffffad5562017d78 EFLAGS: 00000287
Nov 25 21:51:13 hostname kernel: RAX: 0000000000000001 RBX: ffff96899f2f8320 RCX: 0000000000000000
Nov 25 21:51:13 hostname kernel: RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff96899a1af800
Nov 25 21:51:13 hostname kernel: RBP: fffffad5562017df0 R08: 0000000000000000 R09: ffff96899a1af800
Nov 25 21:51:13 hostname kernel: R10: 0000000000000000 R11: fffff96aa3f8aa870 R12: fffff96a2d2cbc000
Nov 25 21:51:13 hostname kernel: R13: fffff96a3c90d3800 R14: fffff967fd77618f0 R15: 0000000000000000
Nov 25 21:51:13 hostname kernel: FS: 0000000000000000(0000) GS:ffff96aa3fa40000(0000) knlGS:0000000000000000
Nov 25 21:51:13 hostname kernel: CS: 0010 DS: 0000 ES: 0000 CR0: 0000000000050033
Nov 25 21:51:13 hostname kernel: CR2: 000056173c9b7000 CR3: 0000006b6af6006 CR4: 0000000003606e0
Nov 25 21:51:13 hostname kernel: DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
Nov 25 21:51:13 hostname kernel: DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
Nov 25 21:51:13 hostname kernel: Call Trace:
Nov 25 21:51:13 hostname kernel: ? kobject_uevent_env+0x442/0x7c0 ↗
Nov 25 21:51:13 hostname kernel: kobject_uevent+0xb/0x10 ↗
Nov 25 21:51:13 hostname kernel: kobject_release+0xf2/0x180
Nov 25 21:51:13 hostname kernel: kobject_put+0x2b/0x50
Nov 25 21:51:13 hostname kernel: sysfs_slab_remove_workfn+0x36/0x40
Nov 25 21:51:13 hostname kernel: process_one_work+0x1db/0x380
Nov 25 21:51:13 hostname kernel: worker_thread+0x4d/0x400
Nov 25 21:51:13 hostname kernel: kthread+0x104/0x140
Nov 25 21:51:13 hostname kernel: ? process_one_work+0x380/0x380
Nov 25 21:51:13 hostname kernel: ? kthread_park+0x80/0x80
Nov 25 21:51:13 hostname kernel: ret_from_fork+0x35/0x40
```

## kobject\_uevent\_env のソース

kobject\_uevent\_env() 内の mutex\_lock が競合していそうと推論した

```
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,
                      char *envp_ext[])
{
    ...
    ... snip

    mutex_lock(&uevent_sock_mutex); 🔒 🔥
    /* we will send an event, so request a new sequence number */
    retval = add_uevent_var(env, "SEQNUM=%llu", ++uevent_seqnum);
    if (retval) {
        mutex_unlock(&uevent_sock_mutex);
        goto exit;
    }
    retval = kobject_uevent_net_broadcast(kobj, env, action_string, ⚡
                                         devpath);
    mutex_unlock(&uevent_sock_mutex); 🔒
}
```

⚠ uevent\_sock\_mutex は Global Mutex ( static DEFINE\_MUTEX(uevent\_sock\_mutex) )。1タスクのみ実行可能なクリティカルセクションになる

シンボルを手がかりに uevent, cgroup と sysfs ファイルを調べ始める

ps の wchan や バックトレースに現れたシンボルを調査し、スライド中盤に説明した内容にたどり着きます

- `kobject_uevent_env()`, `kobject_uevent()` から ...
  - uevent の手がかりをえる
  - udevadm monitor で観測できることを知る
- `sysfs_slab_remove_workfn()` から ...
  - sysfs, slab 周りの手がかりをえる
  - コンテナ名を含む sysfs ファイルを見つける
  - スラブキャッシュの削除処理を疑う

sysfs ファイルと uevent

GMOペパボ

memory cgroup 削除時の uevent を見てみる  
memory cgroup の削除時にも uevent が飛びます

```
root@ubuntu-bionic:~# rmdir /sys/fs/cgroup/memory/test000
```

先と同様に、別のターミナルで `udevadm monitor --kernel` を実行してイベントを捉えられます

```
vagrant@ubuntu-bionic:~$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[28456.994677] remove   /kernel/lib/ext4 inode cache/cgroup/ext4_inode_cache(3960:test000) (cgroup)
KERNEL[28456.994680] remove   /kernel/lib/ext4 inode cache/cgroup/ext4_inode_cache(3960:test000) (cgroup)
KERNEL[28456.994741] remove   /kernel/lib/ufs400255/cgroup/f1p3960:test000 (cgroup)
KERNEL[28456.994791] remove   /kernel/lib/inode cache/cgroup/inode_cache(3960:test000) (cgroup)
KERNEL[28456.994807] remove   /kernel/lib/ufs400268/cgroup/vm_area_struct(3960:test000) (cgroup)
KERNEL[28456.994811] remove   /kernel/lib/ufs400268/cgroup/vm_area_struct(3960:test000) (cgroup)
KERNEL[28456.994844] remove   /kernel/lib/ufs400764/cgroup/files_cache(3960:test000) (cgroup)
KERNEL[28456.994859] remove   /kernel/lib/ufs400768/cgroup/signals_cache(3960:test000) (cgroup)
KERNEL[28456.995052] remove   /kernel/lib/anon_vma/cgroup/anon_vma(3960:test000) (cgroup)
KERNEL[28456.995080] remove   /kernel/lib/ufs400128/cgroup/pid(3960:test000) (cgroup)
```

... snip

28

不可解な事象：削除済みのコンテナの名前を含む sysfs ファイルが残っている

さらに調査を続けると、削除済み(\*) のコンテナ名を含む sysfs ファイルが残っているのを見つけます！

```
$ ls -hal /sys/kernel/slab/dentry/cgroup/ | grep php-??? | head
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(1047519:php-???.lolipop.io-b66d784562)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(1050287:php-???.lolipop.io-d521a277ab)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(1081752:php-???.lolipop.io-b062b877fb)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(2963369:php-???.lolipop.io-bb9c93473e)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(2963651:php-???.lolipop.io-105469307c)
drwxr-xr-x    2 root root 0 Dec  2 14:25 dentry(2964831:php-???.lolipop.io-837af8bd9f)
drwxr-xr-x    2 root root 0 Dec  2 14:26 dentry(2965257:php-???.lolipop.io-c25d1b19b4)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(2967029:php-???.lolipop.io-c6eede56fc)
drwxr-xr-x    2 root root 0 Dec  3 21:26 dentry(2967613:php-???.lolipop.io-98a285af86)
drwxr-xr-x    2 root root 0 Dec  2 14:40 dentry(2967930:php-???.lolipop.io-1aa6f1f40d)

... snip
```

⚠️ 削除 = 関連する cgroup, namespace, プロセスが全て削除されている。コンテナのファイルが全て削除されてるものもあった

cgroup を削除しても sysfs ファイルが残り続ける問題を見つける -> VM で再現を試みる

1. 適当な memory cgroup を作り、bash プロセスを動かしファイルの作成と削除を行います

```
root@ubuntu-bionic:~# mkdir /sys/fs/cgroup/memory/test000
# bash を起動し 適当なファイルを作成・削除する。プロセスはすぐ exit する
root@ubuntu-bionic:~# bash -c 'echo $$ > /sys/fs/cgroup/memory/test000/tasks; touch /tmp/hoge; unlink /tmp/hoge'; exit
```

2. 次に memory cgroup を削除します

```
root@ubuntu-bionic:~# rmdir /sys/fs/cgroup/memory/test000
```

3. memory cgroup と プロセスが無いのに /sys/kernel/slab/\*/\*cgroup 以下にファイルが残ったままになります

```
root@ubuntu-bionic:~# find /sys/kernel/slab/ -type d | grep test000
/sys/kernel/slab/dentry/cgroup/dentry(912:test000) ↗
/sys/kernel/slab/ext4_inode_cache/cgroup/ext4_inode_cache(912:test000) ↗
```

`memory cgroup` を削除しても `sysfs` ファイルが残り続ける問題を見つける

`dentry` や `inode` に関するスラブキャッシュ (reclaimable) のファイルが残ります (\*1)

```
root@ubuntu-bionic:~# find /sys/kernel/slab/ -type d | grep test000
/sys/kernel/slab/dentry/cgroup/dentry(912:test000) ↗
/sys/kernel/slab/ext4_inode_cache/cgroup/ext4_inode_cache(912:test000) ↗
```

これらのファイルは `sysctl vm.drop_caches=2` (\*2) で スラブオブジェクトをクリアすると消えます

```
root@ubuntu-bionic:~$ sysctl vm.drop_caches=2
vm.drop_caches = 2
root@ubuntu-bionic:~$ find /sys/kernel/slab/ -type d | grep test000 # 消えた!
```

⚠️ サービスの環境では、NFS を使っており `nfs_inode_cache` などが残るのも確認した。`buffer_head`, `radix_tree_node` が残るケースもあった。

⚠️ “To free reclaimable slab objects (includes dentries and inodes): echo 2 > /proc/sys/vm/drop\_caches” <https://www.kernel.org/doc/Documentation/sysctl/vm.txt>

memory cgroup を削除しても sysfs ファイルが残り続ける -> uevent

sysctl vm.drop\_caches=2 のタイミングで cgroup 内のスラブキャッシュが削除され uevent が飛びます

```
root@ubuntu-bionic:~$ sysctl vm.drop_caches=2
vm.drop_caches = 2
```

```
vagrant@ubuntu-bionic:~$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[32208.595653] remove    /dentry(3680:test000) (cgroup) ⚡
KERNEL[32208.595717] remove    /dentry(3960:test000) (cgroup) ⚡
...
... snip ( その他のイベントは除外 )
```

sysctl vm.drop\_caches でクリアできることから スラブアロケータのメモリリークではなさそうと推測した(何度も再現できます)

memory cgroup 削除後も、何らかの理由でスラブキャッシュ関連のデータが残ってしまう状態と類推できます

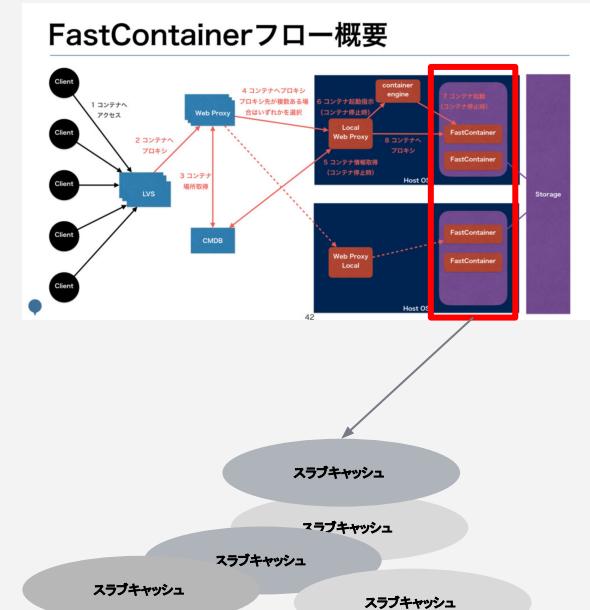
## FastContainer アーキテクチャと合わせて sysfs ファイル残る問題を見直す

- ・HTTP リクエストを契機にしてコンテナが起動する
- ・コンテナは一定のライフタイムの間、起動する
- ・ライフタイムが過ぎたらコンテナは停止する

これを cgroup と sysfs ファイル、スラブキャッシュの観点でみると

- ・HTTP リクエストを契機に memory cgroup が作られる
- ・memory cgroup 内のプロセスによりスラブキャッシュ、sysfs ファイルが作成される
- ・ライフタイムが過ぎてプロセスが停止、memory cgroup が削除される。
- ・しかし、一部のスラブキャッシュ、sysfs ファイルは残る

というサイクルが 繰り返し起きていると推論をたてました



推論の検証： 多数のコンテナ（cgroup）を削除しても sysfs ファイルが残り続ける

名前を変えつつ大量の memory cgroup を作る → bash でファイル作成/削除 → memory cgroup を削除

```
#!/bin/bash

# 100個の cgroup を作る
for i in {000..100}; do
    # memory cgroup を作成
    mkdir /sys/fs/cgroup/memory/test${i}

    # cgroup 内で bash を起動、適当なディレクトリを作成/削除する
    bash -c "echo \$\$ > /sys/fs/cgroup/memory/test${i}/tasks; mkdir /tmp/$i; rmdir /tmp/$i; exit";

    # memory cgroup を削除
    rmdir /sys/fs/cgroup/memory/test${i}
done
```

推論の検証：コンテナ（cgroup）を削除しても sysfs ファイルが残り続ける

やっぱり memory cgroup を作った分だけ sysfs ファイルが残ります！

```
root@ubuntu-bionic:~# find /sys/kernel/slab/ -type d | grep test | sort
/sys/kernel/slab/:a-0000104/cgroup/buffer_head(2564:test000)
/sys/kernel/slab/dentry/cgroup/dentry(1958:test000)
/sys/kernel/slab/dentry/cgroup/dentry(2160:test000)
/sys/kernel/slab/dentry/cgroup/dentry(2564:test000)
/sys/kernel/slab/dentry/cgroup/dentry(2568:test002)
/sys/kernel/slab/dentry/cgroup/dentry(2572:test004)
/sys/kernel/slab/dentry/cgroup/dentry(2574:test005)
/sys/kernel/slab/dentry/cgroup/dentry(2576:test006)
/sys/kernel/slab/dentry/cgroup/dentry(2578:test007)
/sys/kernel/slab/dentry/cgroup/dentry(2582:test009)
/sys/kernel/slab/dentry/cgroup/dentry(2584:test010)
/sys/kernel/slab/dentry/cgroup/dentry(2586:test011)
/sys/kernel/slab/dentry/cgroup/dentry(2588:test012)
/sys/kernel/slab/dentry/cgroup/dentry(2590:test013)

... snip
```

推論の検証：コンテナ（cgroup）を削除しても sysfs ファイルが残り続ける -> uevent の検証

sysctl vm.drop\_caches=2 で消すと、sysfs ファイル数の分だけ uevent が送出されます

```
vagrant@ubuntu-bionic:~$ udevadm monitor --kernel
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[12337.611322] remove /dentry(1155:test100) (cgroup) ⚡
KERNEL[12337.612239] remove /dentry(1153:test099) (cgroup) ⚡
KERNEL[12337.612260] remove /dentry(1151:test098) (cgroup) ⚡
KERNEL[12337.612267] remove /dentry(1149:test097) (cgroup) ⚡
KERNEL[12337.612274] remove /dentry(1147:test096) (cgroup) ⚡
KERNEL[12337.612285] remove /dentry(1145:test095) (cgroup) ⚡
KERNEL[12337.612291] remove /dentry(1143:test094) (cgroup) ⚡
KERNEL[12337.612297] remove /dentry(1141:test093) (cgroup) ⚡
KERNEL[12337.612303] remove /dentry(1139:test092) (cgroup) ⚡
KERNEL[12337.612315] remove /dentry(1137:test091) (cgroup) ⚡
KERNEL[12337.612325] remove /dentry(1135:test090) (cgroup) ⚡

... snip
```

調査：コンテナ（cgroup）を削除しても sysfs ファイルが残り続ける → LKML を調べた

2019年11月の時点では <https://lkml.org/lkml/2019/10/10/1233> がよく似たケースだと推測していました

```
From      Roman Gushchin <>
Subject   [PATCH v2] cgroup, blkcg: prevent dirty inodes to pin dying memory cgroups
Date      Thu, 10 Oct 2019 16:40:36 -0700
```

「`struct bdi_writeback` が `cgroup` への参照をもっているため、dirty な `inode` が 削除される `memory cgroup` へ `pin down` したままになる」と説明されています。また、`cgroup` を定期的に作る環境で遭遇しているとコメントもあり、環境が類似していると予想しました。

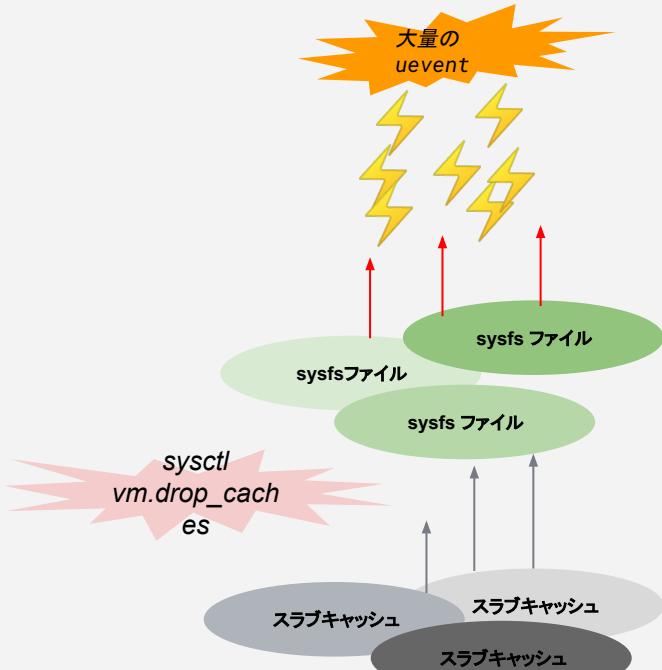
パッチが投稿されていますが、マージはされておらず 実際にこのパッチで問題が解決したかの検証はしていません。  
正直、ちょっと理解が及んでない箇所もあり … たぶん … 類似

## ロードアベレージが上がるシナリオを結論づける

ここまで調査と検証から ...

- ・コンテナの作成/削除が繰り返され 徐々にスラブキャッシュ, sysfs ファイルが溜まってゆく
- ・`sysctl drop_caches` で スラブキャッシュ, sysfs ファイルが削除される
- ・uevent が大量に送出されカーネルレスレッドがブロックし LA 上昇を招く

というシナリオと結論付けました。



## 問題をどうやって解決したか？

- ブートパラメータ `slob_memcg_sysfs=0` を設定した（序盤に説明済み）
  - `/sys/kernel/slab/*/cgroup/*` が作成されなくなる
  - コンテナ(memory cgroup) 削除後も sysfs ファイルが溜まらなくなる
  - uevent も飛ばなくなり、ロードアベレージ上昇は回避できた
- そもそも `sysctl vm.drop_caches` も見直し！

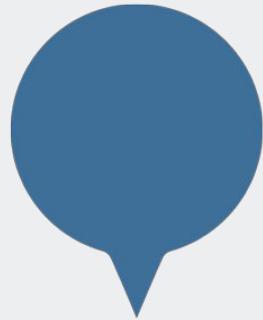
cgroup と sysfs ファイル  
GMOペパボ

```
./sys/kernel/slab/Sラップキヤッッシュ名/cgroup/Sキヤッッシュ名(シリアル番号:Sgroup#) とカーネルコンフィグ  
このリソースの sysfs フォルダは SLOB_MEMCG_SYSFS_ON が有効化に作成されるファイルです
```

**config SLOB\_MEMCG\_SYSFS\_ON**  
defbool 0  
depends on SLOB\_MEMCG  
SLOB creates a directory under /sys/kernel/slab for each  
cgroup in memory, with cache and slab pages being stored  
under it. This is useful for memory accounting, and for  
isolating memory usage between different parts of the system.  
This is also useful for memory debugging, as it allows  
the user to easily see which files have been created. This is  
a very high configuration parameter, so it is recommended  
to set it to a very low value. The default value is 0.  
config option SLOB\_MEMCG\_SYSFS\_ON\_DEFAULT 0  
config option SLOB\_MEMCG\_SYSFS\_ON\_DEFAULT\_VALUE 0

また、ブートパラメータの `slob_memcg_sysfs` を指定することで、sysfs ファイルを削除しないように参数を変更されます  
このノードパラメータは表示や操作上OKです

⚠️ memory cgroup 削除後もスラブキヤッッシュが残っているのは解決していない気がするが、問題として顕在化していない



## Section 5

# 2020年10月時点でのアップデート

ここまで話は 2019年11月時点での情報や記録を元に構成した内容です。  
2020年10月時点で Linux カーネル 5.9 (mainline) がリリースされており  
本スライドに関するアップデートが複数入っています

## 5.8-rc1 の変更

sysfs ファイル（スラブキャッシュ生成）の uevent 通知が削除されました

```
author  🌐 Christoph Lameter <cl@linux.com>      2020-06-01 21:45:50 -0700
committer 🖼 Linus Torvalds <torvalds@linux-foundation.org> 2020-06-02 10:59:06 -0700
commit   d7660ce5914d396242bfc56c8f45ef117101fb58 (patch)
tree     22ee5b95973815e7b8603c1d2356580fb7fceee69
parent   52f23478081ae0dcdb95d1650ea1e7d52d586829 (diff)
download linux-d7660ce.tar.gz
```

**slub: Remove userspace notifier for cache add/remove**

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d7660ce5914d396242bfc56c8f45ef117101fb58>

## 5.8-rc1 での変更: sysfs ファイル（スラブキャッシュ生成）の uevent 通知が削除されました

```
@@ -5795,7 +5782,6 @@ static void sysfs_slab_remove_workfn(struct work_struct *work)
```

```
#ifdef CONFIG_MEMCG
    kset_unregister(s->memcg_kset);
#endif
-    kobject_uevent(&s->kobj, KOBJ_REMOVE);
out:
    kobject_put(&s->kobj);
}
```

```
@@ -5853,7 +5839,6 @@ static int sysfs_slab_add(struct kmem_cache *s)
```

```
}
#endif

-    kobject_uevent(&s->kobj, KOBJ_ADD);
if (!unmergeable) {
    /* Setup first alias */
    sysfs_slab_alias(s, s->name);
```

5.8-rc1 での変更: sysfs ファイル（スラブキャッシュ生成）の uevent 通知が削除されました

以下のようなコミットログが付いています

*“These notifiers may just exist because in the initial slub release the sysfs code was copied from another subsystem.”*

“これらの通知は、最初のslubリリースでsysfsのコードが他のサブシステムからコピーされたために存在しているだけかも”

他のサブシステムからコピーされたコードだから ... これは笑っていいところかな 😊

## 2020年10月時点でのアップデート

5.8-rc1 での変更: sysfs ファイル（スラブキャッシュ生成）の uevent 通知が削除されました

コミットログは下記のように続きます

*"I came across some unnecessary uevents once again which reminded me this. ... (snip)"*

不要ない uevent をいくつか見つけたんだけど、再びこれ（スラブキャッシュの uevent）を思い出したよ

*"Kmem caches are internal kernel structures so it is strange that userspace notifiers would be needed. And I am not aware of any use of these notifiers."*

*kmem caches* (スラブキャッシュ) はカーネルの内部構造だから、ユーザ空間通知が必要ってのは不思議だよね。

そして、この通知を使ってるモノは知らないなあ

ユースケースはなんだろう？と疑問に思っていたら、不要ない機能で削除されたというオチがつきました

## 5.9-rc1 で解決された問題

memory cgroup 削除後も sysfs ファイル（スラブキャッシュ）が残ってしまう問題が解決されました

```
author      Roman Gushchin <guro@fb.com>          2020-08-06 23:21:27 -0700
committer   Linus Torvalds <torvalds@linux-foundation.org> 2020-08-07 11:33:25 -0700
commit      10befea91b61c4e2c2d1df06a2e978d182fcf792 (patch)
tree        cc5c9a0e4d351194ccde996f5455248f45b5d56c
parent      15999eef7f25e2ea6a1c33f026166f472c5714e9 (diff)
download    linux-10befea91b61c4e2c2d1df06a2e978d182fcf792.tar.gz
```

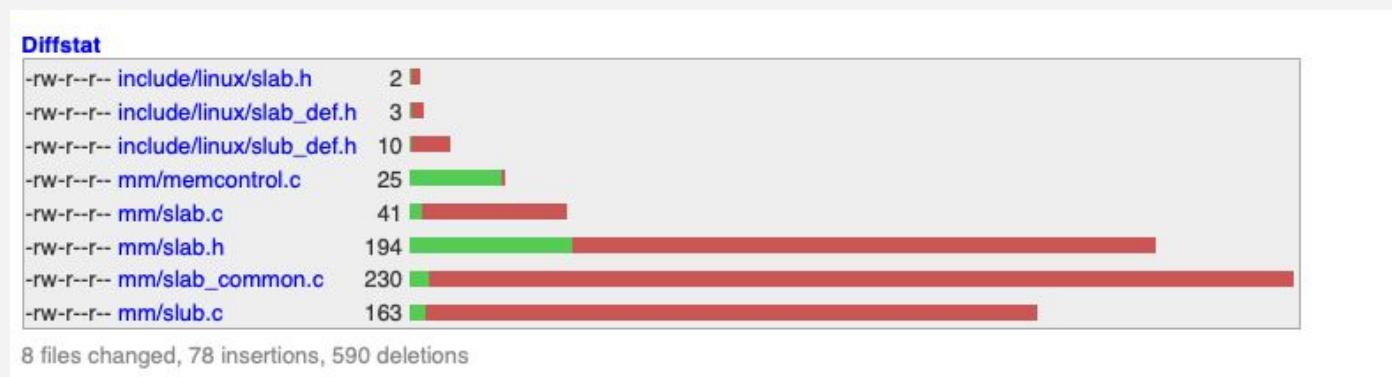
**mm: memcg/slab: use a single set of kmem\_caches for all allocations**

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=10befea91b61c4e2c2d1df06a2e978d182fcf792>

## 2020年10月時点でのアップデート

GMOペパボ

5.9-rc1 で解決された問題: `memory cgroup` 削除後も `sysfs` ファイル（スラブキャッシュ）が残ってしまう問題が解決されました。



<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=10befea91b61c4e2c2d1df06a2e978d182fcf792>

cgroup で kmem\_cache の扱い変える粒度の大きいコミットの中で、副次的に解決されたようです。

コードの量が多くて、正確に解説できる自信がないです! すいません

## 5.9-rc1: slub\_memcg\_sysfs に変更が入りました

/sys/kernel/slab/\*/cgroup/\* のファイルを作成しないようにするブートオプション `slub_memcg_sysfs` は、何も機能しないオプションになりました（一つ前のコミットに含まれている変更）

```
5763 - #ifdef CONFIG_MEMCG
5764 -     if (is_root_cache(s) && memcg_sysfs_enabled) { → on / off を保持する変数
5765 -         s->memcg_kset = kset_create_and_add("cgroup", NULL, &s->kobj);
5766 -         if (!s->memcg_kset) {
5767 -             err = -ENOMEM;
5768 -             goto out_del_kobj;
5769 -         }
5770 -     }
5771 - #endif
5772 -
```

- ・ただブートオプションとして存在するだけになった
- ・オプション自体消してもよさそうですが、後方互換のためにだけ残されているのか？

余談: Linux カーネル 5.9 は 2020/10/11 にリリースされました

The Linux Kernel Archives



About Contact us FAQ Releases Signatures Site news

Protocol Location  
HTTP <https://www.kernel.org/pub/>  
GIT <https://git.kernel.org/>  
RSYNC <rsync://rsync.kernel.org/pub/>

Latest Release  
**5.9** 

mainline: 5.9	2020-10-11	[tarball] [pgp] [patch]	[view diff] [browse]
stable: 5.8.15	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
longterm: 5.4.71	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
longterm: 4.19.151	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
longterm: 4.14.201	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
longterm: 4.9.239	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
longterm: 4.4.239	2020-10-14	[tarball] [pgp] [patch]	[inc. patch] [view diff] [browse] [changelog]
linux-next: next-20201015	2020-10-15		[browse]

二ヶ月ほど前に @ten\_forward さんのお誘いを受けてから 勉強会に向けて一年前のネタを調べ直していたら、なんと問題が解決された mainline がちょうどリリースされることになるとは ... !

発表内容も厚くなり、まとまりとオチもついて非常に安心(?)しました ☕

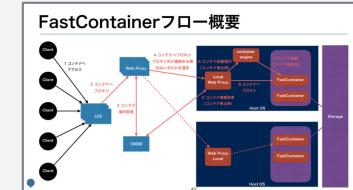


Section 6

# まとめ

## まとめ（1）以下のセクションのお話をしました

- リリップ！マネージドクラウドの紹介
  - FastContainer アーキテクチャの紹介
- cgroup と sysfs ファイル
- sysfs ファイルと uevent
- トラブルシューティング事例
  - sysctl vm.drop\_caches !
- 2020年10時点でのアップデート



トラブルシューティングを切り口にしたことで、少し角度で cgroup あるいはコンテナの勉強になったでしょうか？

## まとめ（2）感想

- サービスを提供していると、環境や条件が組み合わさって思わぬ「問題」として踏むことがあります
- ビジネスでは 技術的な問題は踏まないことが最もです ... 🔥 💰
- 詳細を探究するトリガーになることもありますね
- トラブルシューティングや失敗事例も公開できる、心理的安全性の高い業界にしていきたいですね

（繰り返しますが、トラブルや失敗はないのがベストですね）

## まとめ（3） ブログも書いています

今回の話は私のブログでもまとめております（トラブルシューティング事例はこのスライドだけに載せています）



2020-09-29

### 【続】Linux Kernel: cgroup 削除後も残り続ける slab キャッシュについての調べ物 - upstream は修正パッチが入って解決済み

以前に下記のエントリを書きました。勉強会に登壇するネタとして調べ直したところ新しいLinuxカーネルでは同問題が解決されていました。

hiboma@DESKTOP-1HJF0D8: ~ % cd /tmp  
hiboma@DESKTOP-1HJF0D8: /tmp % git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git  
hiboma@DESKTOP-1HJF0D8: /tmp % cd linux  
hiboma@DESKTOP-1HJF0D8: /tmp/linux % make -j4  
hiboma@DESKTOP-1HJF0D8: /tmp/linux % ./vmlinuz-5.1.0-500401-gene...  
hiboma@DESKTOP-1HJF0D8: /tmp/linux %

hiboma の日記 -【続】Linux Kernel: cgroup 削除後も残り続ける slab キャッシュについての調べ物 - upstream は修正パッチが入って解決済み

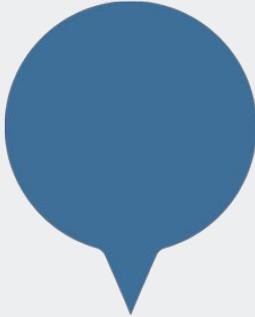
<https://hiboma.hatenadiary.jp/entry/2020/09/29/122059>

ご清聴ありがとうございました

最後に。引越し後、近所のホームセンターでみつけたコンテナです



社内 Slack に写真をあげたところ「CRI 互換じゃないね」と同僚に突っ込まれました。 ほんとですね!



## Section 6

# 落穂拾い

時間が余っていたらお話しします

## アラート発生までの全体像を改めて整頓

1. ホストのメモリが一定の使用量に達するとシステムのスラブキャッシュが `sysctl vm.drop_caches` でクリアされる
2. コンテナ削除後にも溜まっていたスラブキャッシュが一斉にクリアされる

`/sys/kernel/slab/cgroup/$キャッシュ名($シリアル番号:$コンテナ名)` の削除も起きる

3. sysfs ファイル削除の uevent が大量に送出されようとする
4. uevent を処理する `kobject_uevent_env()` の `mutex_lock()` で競合を引き起こす
5. 多数の kworker カーネルスレッドがロックでブロックし `TASK_UNINTERRUPTIBLE` になる (\*)
6. ロードアベレージが上昇し、監視の閾値に達してアラートを出す 

⚠ STATE D `TASK_UNINTERRUPTIBLE` なタスクはロードアベレージに計上される

落穂拾い:当時のトラブルシューティングでのリアルな様子 🔥

当時 遭遇した時はスムーズには調査は進まず、何がなんだか分からない状態で調べていたのがリアルなところ

「コンテナ提供しているホストで、時折 でる LA 上昇のアラートなんだろうね?」

「`sysctl vm.drop_caches` じゃない?」

「なんか LA 上がる際にたくさんのカーネルスレッドが `kobject_uevent` で詰まってるぽいな?」

「`uevent? cgroup` 関連の `uevent` 飛んでるぞ?」

「イベントの中身を見ると `/sys/kernel/slab/ ...`」

「すでに存在しないコンテナ (`cgroup`) の スラブが混じってるな?」

「`cgroup` 消しても残る スラブキャッシュがいるな ???」

トラブルシューティング事例は、後からまとめ直すとスッキリと書きがちですね

## スラブキャッシュが溜まるのは他の環境で問題にならないのだろうか？

memory cgroup 削除後も残るのは reclaimable なスラブキャッシュでカーネルが回収 (reclaim) 可能です。

ホストの Free なメモリが少なくなつて cache pressure がかかり カーネルのメモリ回収 (reclaim) が走った場合にも スラブオブジェクトが reclaim  
→ スラブキャッシュが削除され uevent は送出されます。(\*1)

しかしながら sysctl vm.drop\_caches と比較して、cache pressure で 回収 (reclaim) される場合

- reclaim されるスラブキャッシュの件数が少なくおさまる？
- uevent 件数が少なくなる？

と推測されます。結果、観測できる負荷にはならないか？このようにして、その他 cgroup を用いる多くの環境では特に「問題」として顕在化していない可能性があります。(\*2)

⚠️ 先の再現の手順を行った後に、ホストのメモリを使い切るようなプロセスを動かすと再現できます

⚠️ 繰り返しますが、推測のコメントです