

コンテナのセキュリティを 中身から理解しよう

皆さんは、unshare(2)しますか？

近藤宇智朗 / 森田浩平

GMO Pepabo, Inc.

Kyushu Security Conference 2018



シニア・プリンシパルエンジニア

近藤 宇智朗 / @udzura

Uchio Kondo

技術部 技術基盤チーム

<https://blog.udzura.jp/>

本日の手順、コードなど



<https://bit.ly/kyusec-pepabo>



アジェンダ

1. コンテナ仮想化って何だ？

難易度: 初級

2. コンテナの仕組みをのぞいてみよう

難易度: 初～中級

～小休憩～

3. コンテナに対する攻撃の考え方

難易度: 中級

～休憩～

4. 実践！コンテナ攻撃ワークショップ

難易度: 上級

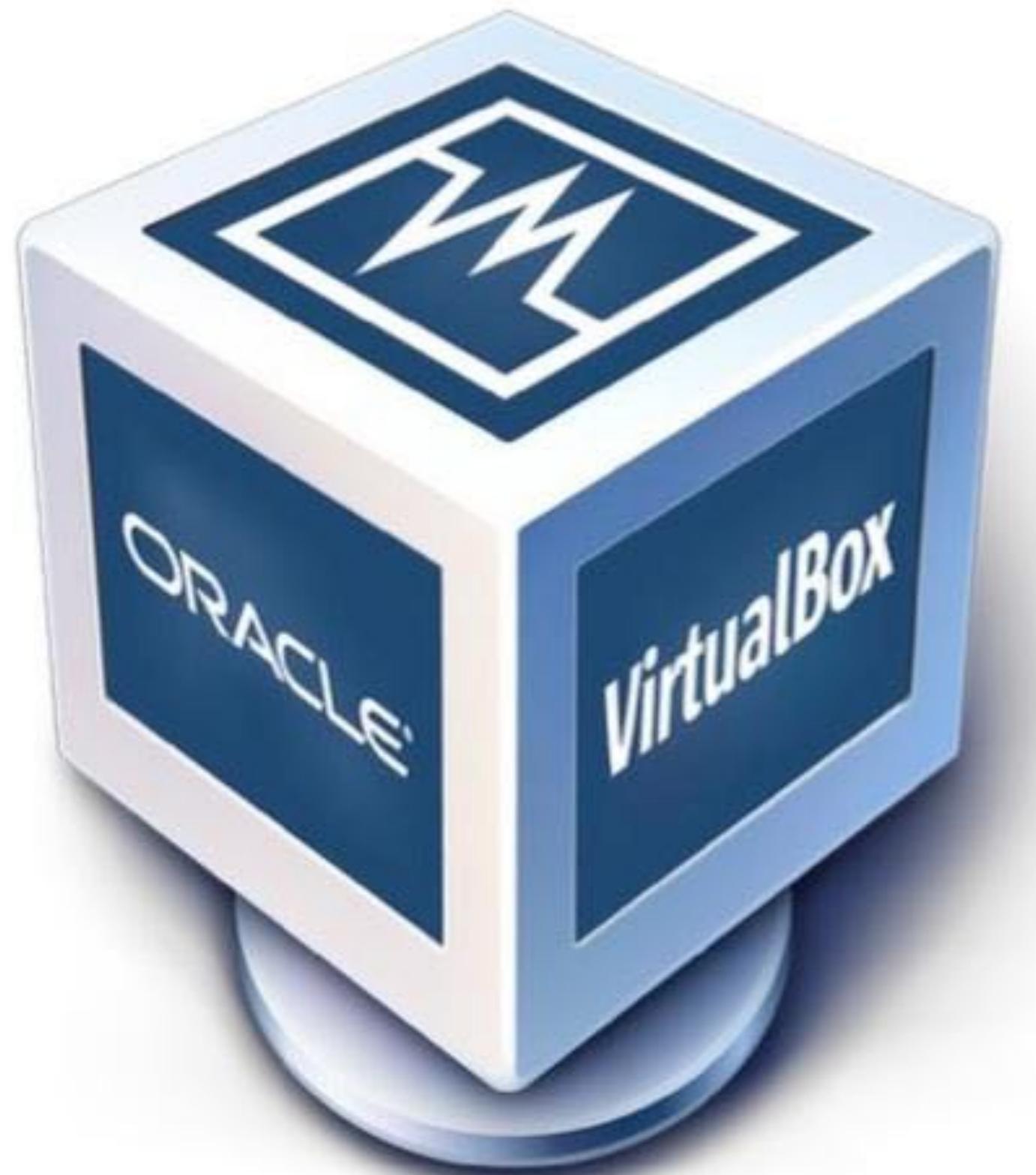
81

コンテナ仮想化って何だ？

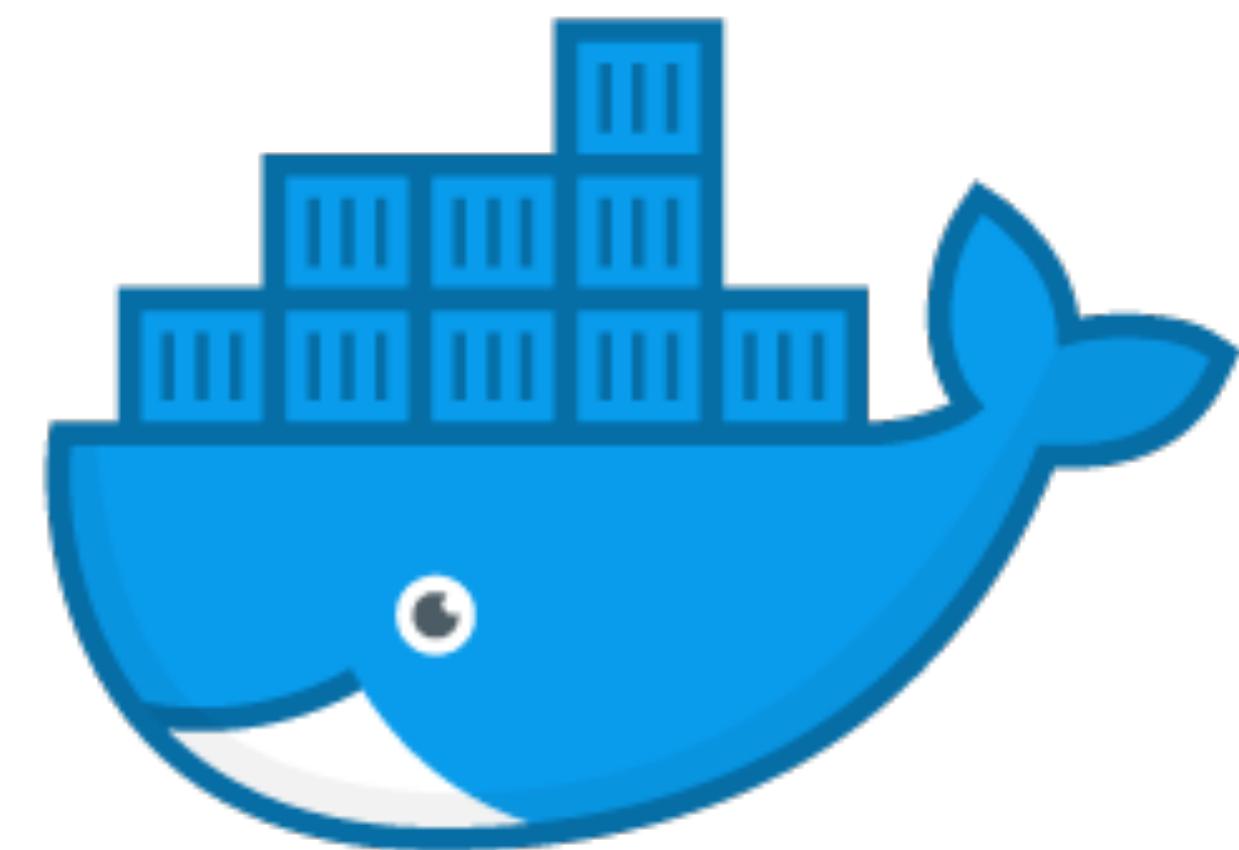
<https://flic.kr/p/gjDrZY>



次のソフトウェアを
知っている方
使ったことがある方？





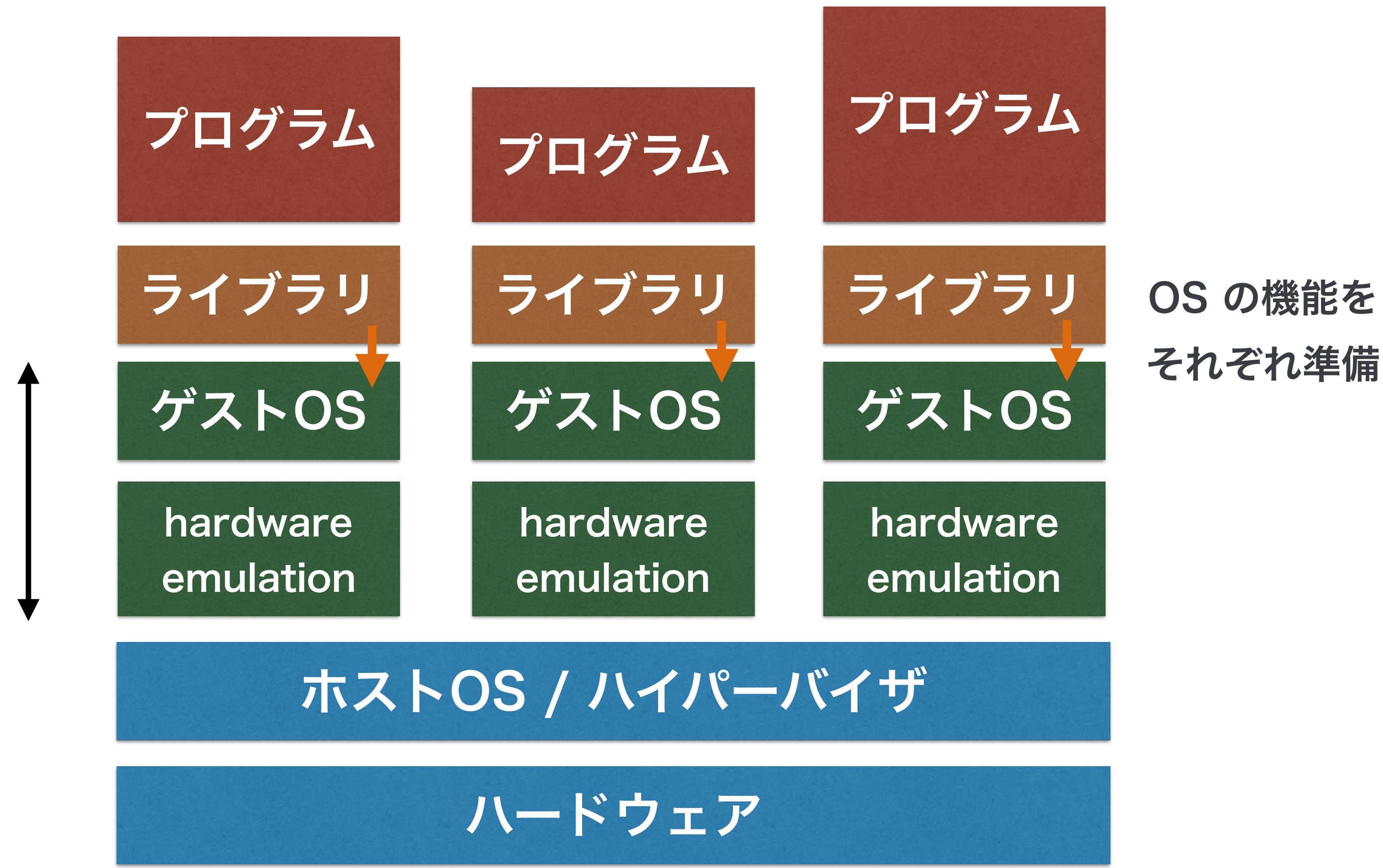


docker®

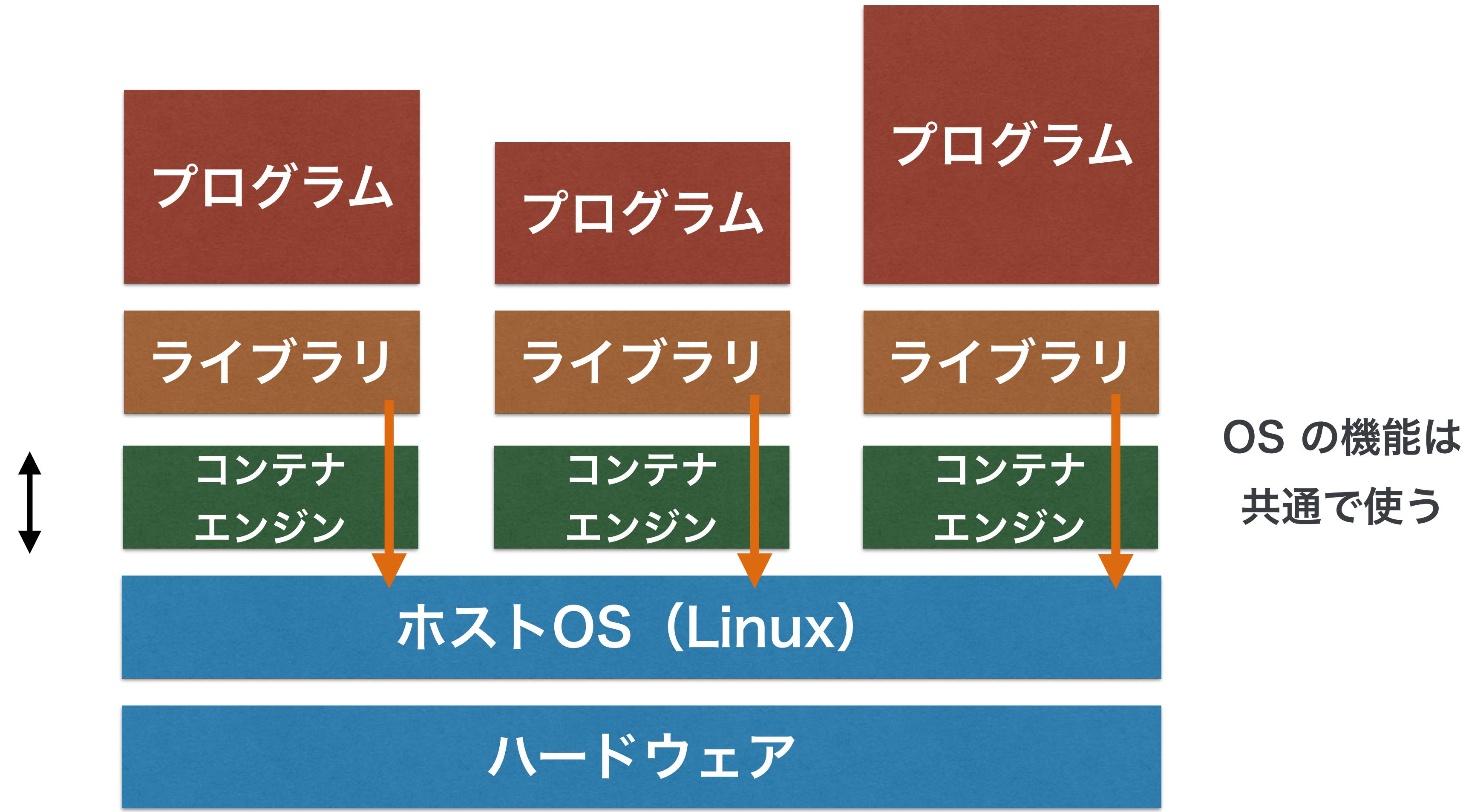


今日のテーマ コンテナ

いわゆる仮想化（例）

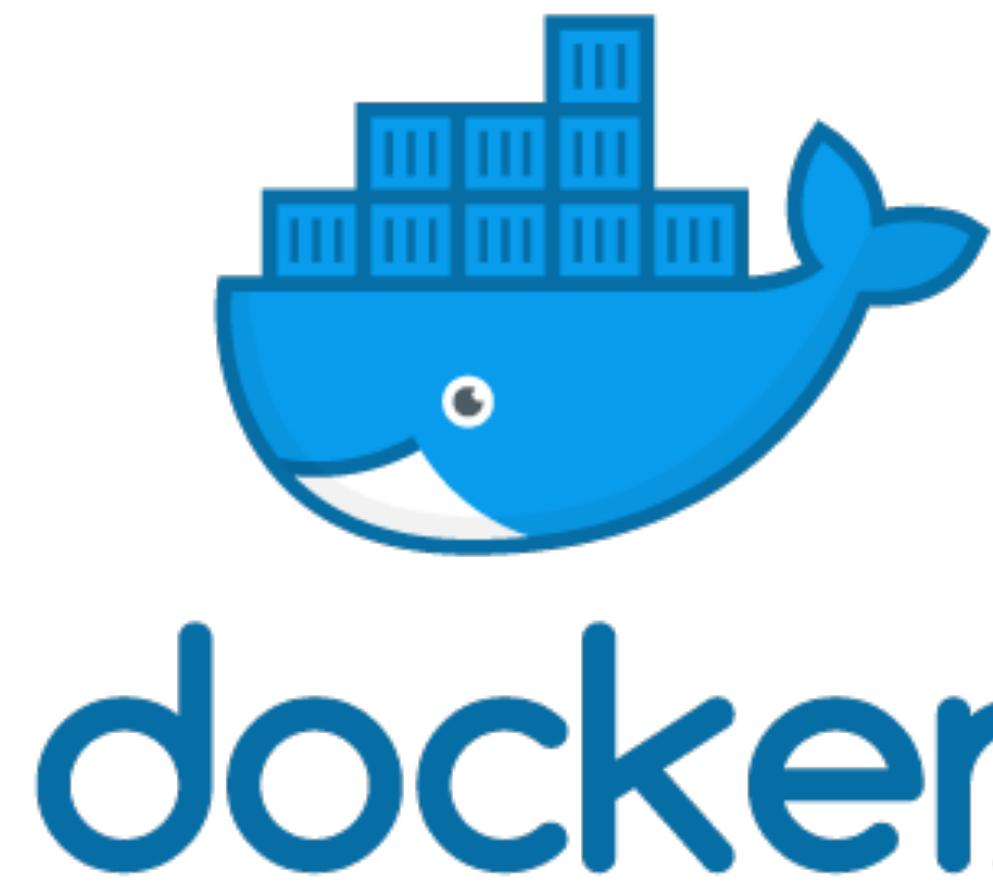


コンテナ型「仮想化」

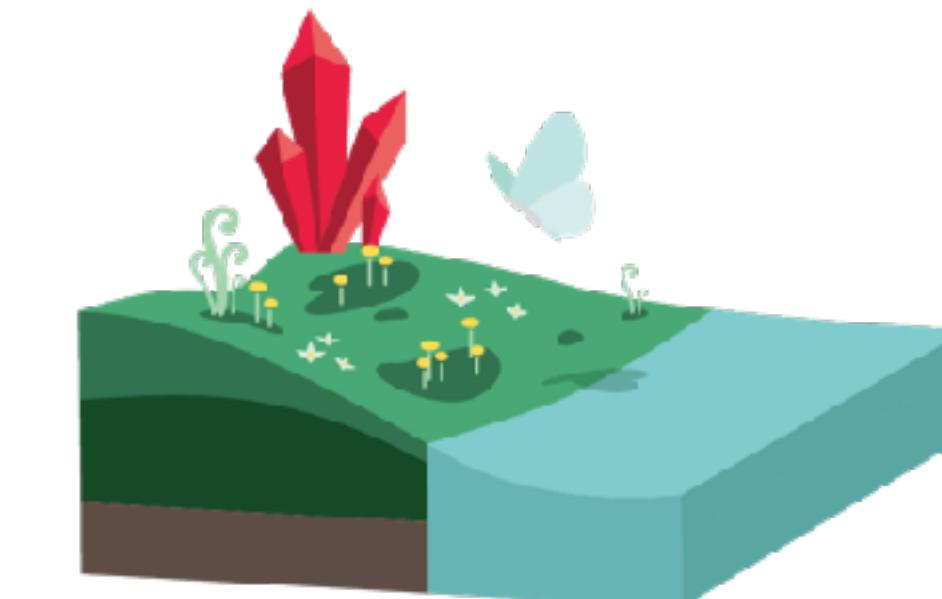


コンテナの実装

代表的実装



LXC



HA CON IWA



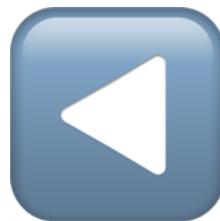
コンテナのメリット

- 1) 起動がより高速、軽量
- 2) リソースを細かく制御可能

→ クラウド、IoTに向いている

逆に、デメリットは？

分離がいわゆるVMと比べ「弱い」



強い

.....

権限分離

.....

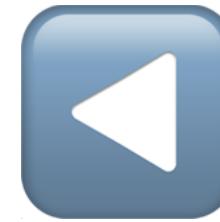
弱い



VM 作成する仮想化
ハイパーバイザ型/
ソフトウェア型...

コンテナ型仮想化

ミドルウェアの中で
権限分離するタイプ
VirtualHost等



悪い

.....

リソース効率

.....

良い



セキュリティ
重要

コンテナのセキュリティを押さえよう

- ・コンテナ自体、他の技術と比べごく最近出てきたものである
- ・情報も運用実績もまだまだ少ない
- ・でも、今後欠かせない技術になるので、セキュリティをしっかり押さえて他のエンジニアと差をつけましょう
- ・(つけているとは言っていない)



§2

コンテナの仕組みを
のぞいてみよう

<https://flic.kr/p/9R6ast>



ワークショップ(前半)のアジェンダ

- ・ 全体の解説

座学

1. Dockerのインストール
2. 「コンテナはプロセス」の確認
3. Linux Namespace の確認と実習
4. cgroup の確認と実習
5. 簡単なコンテナを自作してみましょう



一緒に手を動かしつつ
見てみましょう

プログラミング
してみましょう！

コンテナは
どうやって
コンテナになっているのか？

**次の画面を
見たことがある方？**

アクティビティモニタ (自分のプロセス)

CPU メモリ エネルギー ディスク ネットワーク Q. 検索

プロセス名	% CPU	CPU時間	スレッド	アイドル・ウェイクアップ	PID	ユーザ
アクティビティモニタ	38.7	9:34.78	7	11	1400	udzura
FirefoxCP Web Content	1.8	25:42.09	45	39	23087	udzura
VBoxHeadless	1.7	35:48.75	31	217	7790	udzura
FirefoxCP Web Content	1.5	36:31.20	44	140	22053	udzura
Firefox	1.4	1:13:04.46	77	6	22052	udzura
Slack	1.3	41:52.18	36	5	1828	udzura
VBoxHeadless	1.2	16:40.09	34	197	67243	udzura
com.docker.hyperkit	1.2	21:56.67	20	73	1754	udzura
Google Chrome Helper	1.0	23:02.74	19	5	1971	udzura
Slack Helper	1.0	29:43.06	22	4	8568	udzura
VBoxHeadless	0.8	17:18.01	28	195	65118	udzura
SKYSEA Client View	0.5	36:27.20	37	14	1490	udzura
CoreServicesUIAgent	0.5	0.11	3	3	9337	udzura
FirefoxCP Web Content	0.4	16:26.22	49	117	24620	udzura
distnoted	0.3	8:32.29	11	3	1252	udzura
BitBar	0.2	4:39.77	6	2	1507	udzura
FirefoxCP Web Content	0.2	33:49.73	42	7	35370	udzura
Google Chrome	0.2	15:24.39	33	14	1866	udzura
iconservicesagent	0.2	28.46	3	2	1438	udzura
VBoxSVC	0.2	3:59.12	20	7	7712	udzura
Slack Helper	0.2	4:42.51	20	3	8147	udzura
mysqld	0.2	3:09.01	37	75	1619	udzura
Keynote	0.2	28:39.66	6	4	34546	udzura
Slack Helper	0.1	29.22	20	4	91090	udzura
Slack Helper	0.1	3:53.13	19	1	1857	udzura

システム: 6.97% CPU負荷: スレッド: 2355
 ユーザ: 6.52% プロセス: 459
 アイドル状態: 86.50%

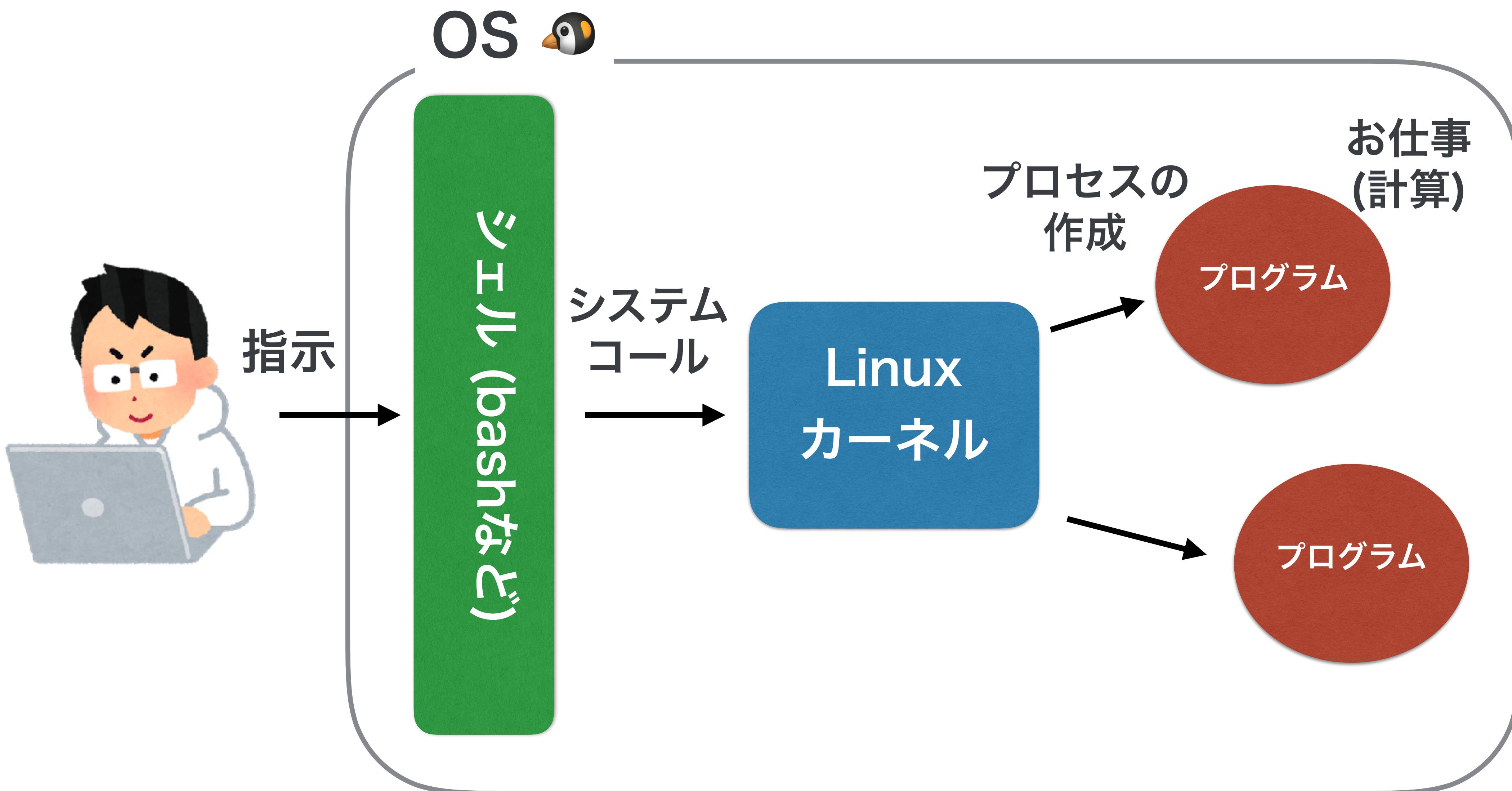


ターミナル — tmux — 145x40 [11/492689]

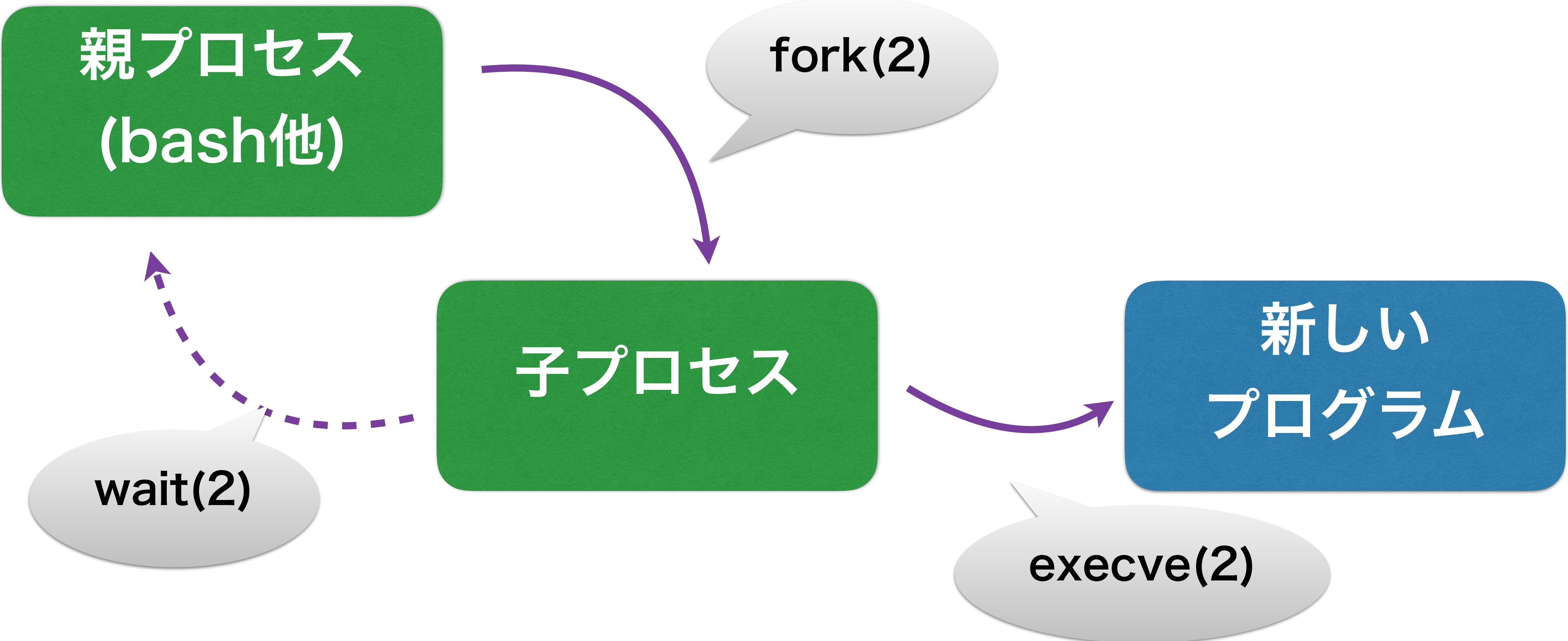
USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.5	37864	5856	?	Ss	06:31	0:01	/sbin/init
root	247	0.0	0.3	28356	3240	?	Ss	06:31	0:00	/lib/systemd/systemd-journald
root	290	0.0	0.1	102968	1556	?	Ss	06:31	0:00	/sbin/lvmetad -f
root	305	0.0	0.3	44236	3352	?	Ss	06:31	0:00	/lib/systemd/systemd-udevd
root	616	0.0	0.7	278204	7972	?	Ssl	06:31	0:00	/usr/lib/accountsservice/accounts-daemon
syslog	639	0.0	0.2	256396	3048	?	Ssl	06:31	0:00	/usr/sbin/rsyslogd -n
root	641	0.0	0.2	28632	2880	?	Ss	06:31	0:00	/lib/systemd/systemd-logind
message+	650	0.0	0.3	42904	3548	?	Ss	06:31	0:02	/usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation
root	723	0.0	0.2	29268	2596	?	Ss	06:31	0:00	/usr/sbin/cron -f
root	809	0.0	0.2	16128	2924	?	Ss	06:31	0:00	/sbin/dhclient -1 -v -pf /run/dhclient.enp0s3.pid -lf /var/lib/dhcp/dhclient.enp0s3.leases -I -df /var/lib/dhcp/dhclient6.enp0s3.leases enp0s3
root	1060	0.0	0.2	250624	2444	?	S1	06:31	0:08	/usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.pid
root	1106	0.0	0.5	65612	5964	?	Ss	06:31	0:00	/usr/sbin/sshd -D
vagrant	3777	0.0	0.4	95460	5060	?	S	08:40	0:38	_ sshd: vagrant@pts/0
vagrant	3778	0.0	0.4	25172	5048	pts/0	Ss	08:40	0:00	_ -bash
vagrant	15826	0.0	0.3	37764	3096	pts/0	R+	23:35	0:00	_ ps auxf
vagrant	6390	0.0	0.4	95460	4868	?	S	09:09	0:00	_ sshd: vagrant@pts/1
vagrant	6391	0.0	0.4	24996	4820	pts/1	Ss+	09:09	0:00	_ -bash
root	1107	0.0	5.4	558784	55044	?	Ssl	06:31	0:57	/usr/bin/dockerd -H fd://
root	1229	0.0	1.3	367208	13996	?	Ssl	06:31	0:32	_ containerd -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --metrics-interval=0 --start-timeout 2m --state-dir /var/run/docker/libcontainerd/containerd --shim containerd-shim --runtime runc
root	5314	0.0	0.3	207368	3664	?	S1	08:48	0:00	_ containerd-shim 88b7e296fe92836b46295978e9f284088593d8287201a04db521cdb129faa10e runc
root	5331	0.0	0.1	4504	1624	?	Ss	08:48	0:00	_ /bin/sh /usr/sbin/apache2ctl -D FOREGROUND
root	5357	0.0	0.4	71584	4436	?	S	08:48	0:01	_ /usr/sbin/apache2 -D FOREGROUND
www-data	5358	0.0	0.6	426340	6248	?	S1	08:48	0:03	_ /usr/sbin/apache2 -D FOREGROUND
www-data	5359	0.0	0.6	426340	6280	?	S1	08:48	0:03	_ /usr/sbin/apache2 -D FOREGROUND
root	6991	0.0	0.3	141832	3604	?	S1	10:29	0:00	_ containerd-shim ebb02470253b91584ce9f829754bc00bc959803ee5332051eaf068cfec5b9a73 runc
root	7005	0.0	0.1	4504	1548	?	Ss	10:29	0:00	_ /bin/sh /usr/sbin/apache2ctl -D FOREGROUND
root	7031	0.0	0.3	71584	3568	?	S	10:29	0:01	_ /usr/sbin/apache2 -D FOREGROUND
www-data	7032	0.0	0.3	426340	3104	?	S1	10:29	0:02	_ /usr/sbin/apache2 -D FOREGROUND
www-data	7033	0.0	0.3	426340	3108	?	S1	10:29	0:02	_ /usr/sbin/apache2 -D FOREGROUND
root	13654	0.0	0.5	141832	5648	?	S1	16:50	0:00	_ containerd-shim 92caf63068eecf285189635778c52e645df678efac92a324ea567199cb47ba05 runc
root	13668	0.0	0.1	4504	1604	?	Ss	16:50	0:00	_ /bin/sh /usr/sbin/apache2ctl -D FOREGROUND
root	13694	0.0	0.3	71584	3520	?	S	16:50	0:00	_ /usr/sbin/apache2 -D FOREGROUND

[0] 1:vagrant 2:zsh 3:zsh 4:zsh- 5:[tmux]* "root@ebb02470253b: /" 15:50 27- 9-18

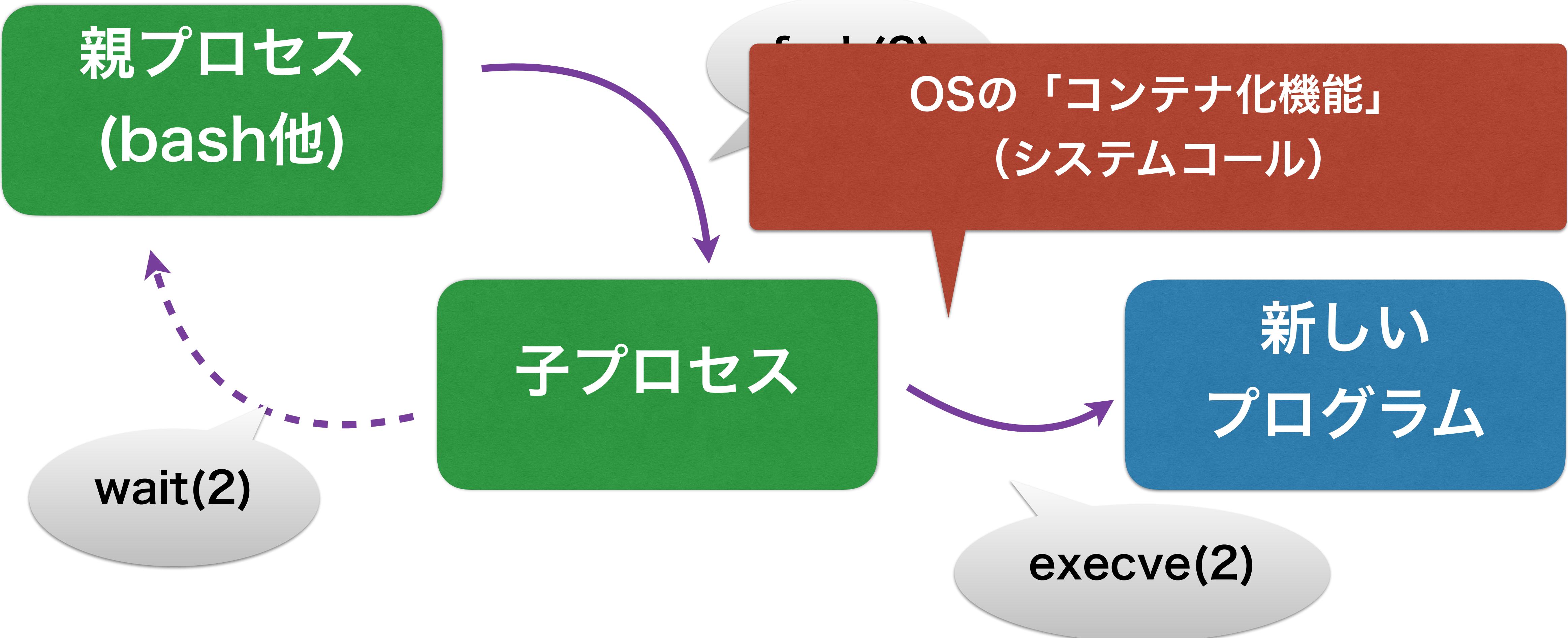
Linuxでプログラムを動かしたい！



Linuxの「プロセス」



コンテナは「特殊なプロセス」



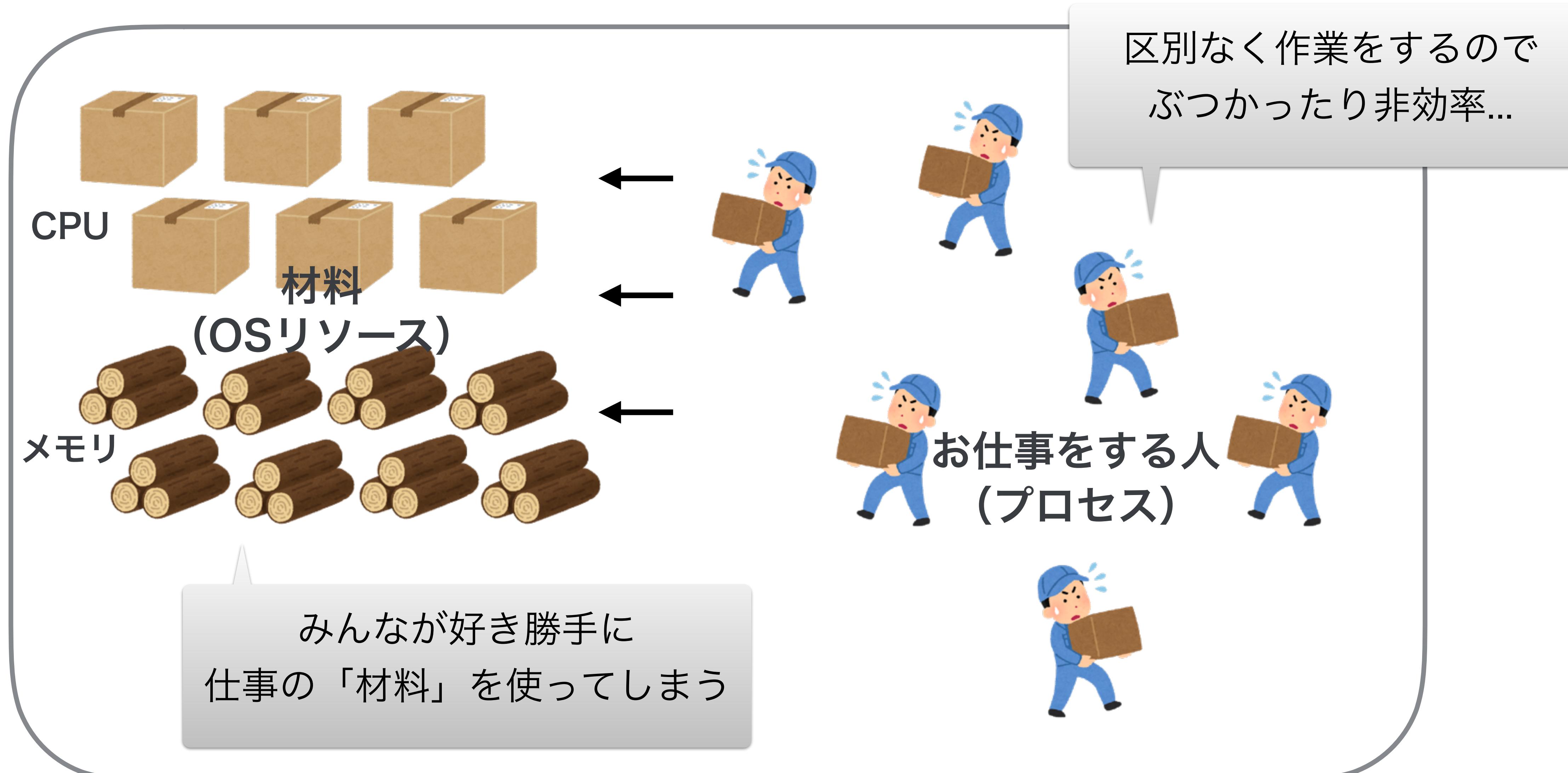
「特殊」とは？

普通のプロセスとコンテナの違い

- ・コンテナは、特殊なプロセスであると考えられる。
- ・具体的には、 1) ホストから独立したリソース空間を付与し、
2) ホストから利用できるハードウェアリソースなどに制限を与える
ことで、個別に独立した作業空間を確保しているイメージ
- ・1) のための機能の代表が **Linux Namespace**、
2) のための機能の代表が **cgroup** である。

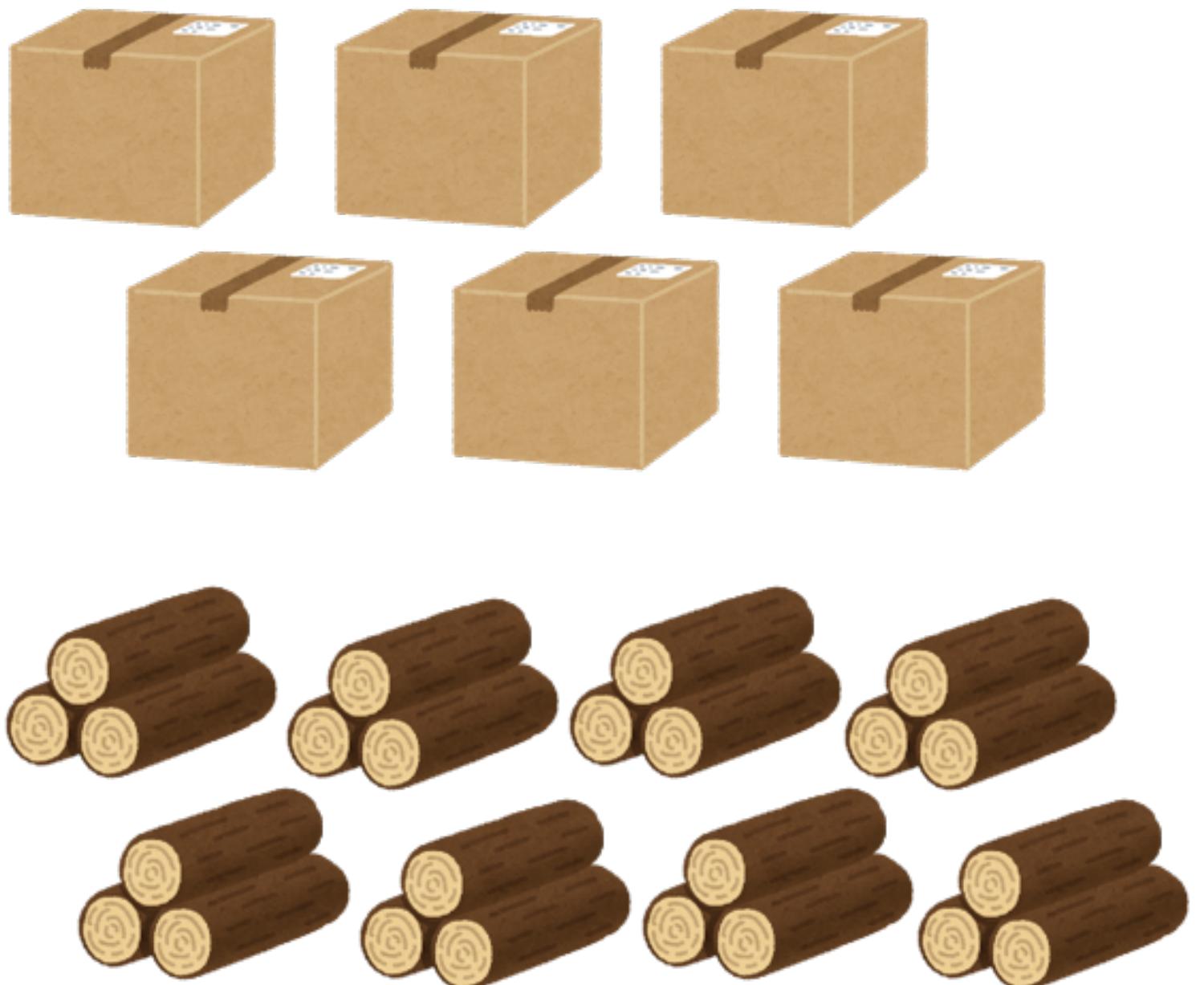


コンテナのない世界のOS



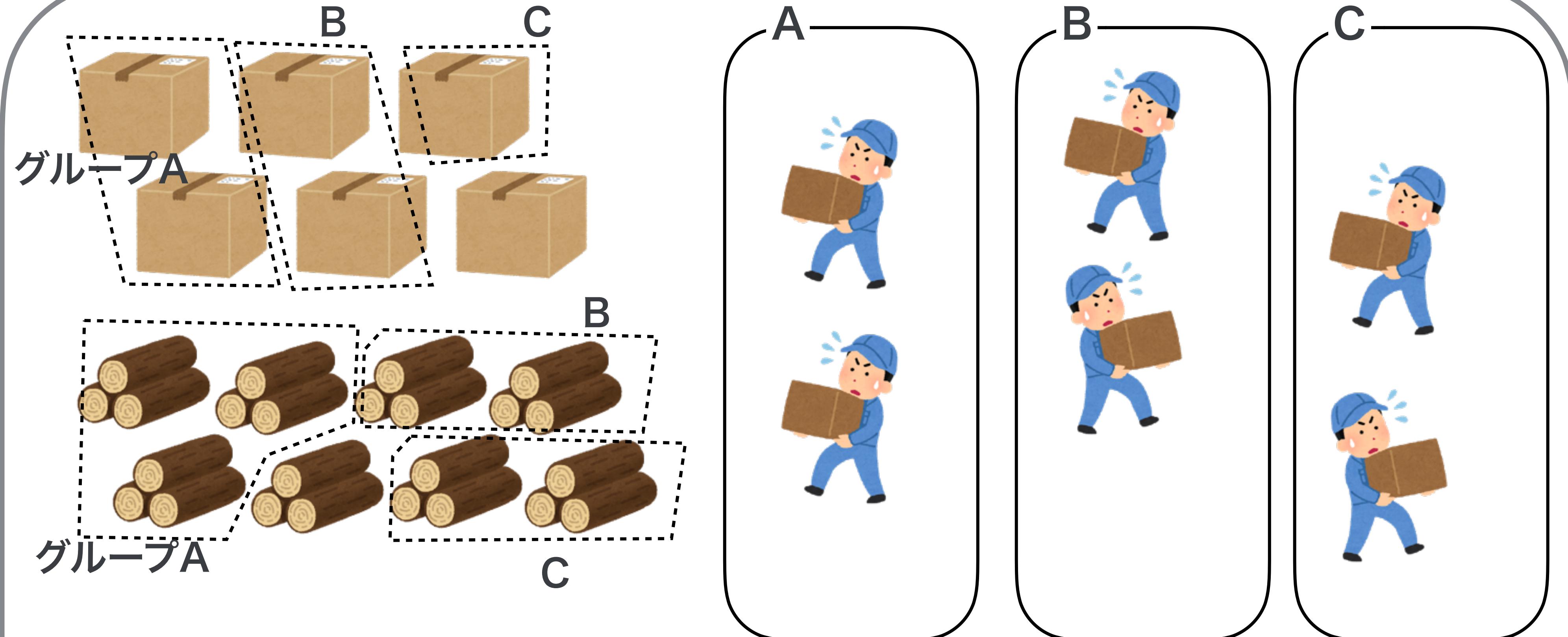
※ざっくりした例えです

作業を安全・効率化するため



- 「作業の部屋」を分ける必要がある

作業を安全・効率化するため



- ・「作業のための材料の割り振り」も明確にしたい

手を動かして
見てみましょう

班分け

- ・人数も多いので、（参加したい方は）ざっくり前後の人たちで班わけという形にします。
- ・詰めて座ってください！
- ・班の方々で、協力しあって進めてみてください。
- ・たまに 当てます。



1.

Docker のインストール

Dockerをインストール

- ・(事前インストールされた方は省略可能です...)
- ・ご準備していただいた Ubuntu Xenial に、公式の手順でdocker-ce パッケージを入れてください。
- ・→ <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- ・インストール後、以下のコマンドを打っておくとsudoが不要になって楽です

```
sudo gpasswd -a vagrant docker
```

確認

```
vagrant@localhost:~$ docker info
Containers: 1
Running: 0
Paused: 0
Stopped: 1
Images: 2
Server Version: 1.13.1
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
Dirs: 16
Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: N/A (expected: 9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
```

2.

「コンテナはプロセス」
の確認

Docker上でapacheを動かそう

- 以下のようにDockerfileを作成する (sample1/Dockerfile)

```
FROM ubuntu:xenial
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get -q -y update && apt-get -q -y install apache2
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

- 以下のようにimageをビルドする

```
$ docker build -t kyusec-1 udzura/section-2
```

コンテナを立ち上げる

- ・以下でapache2のコンテナが立ち上がり、アクセスできる。

```
$ docker run -p 8080:80 -d kyusec-1
$ curl localhost:8080 -s | grep '<title>'<br/>
<title>Apache2 Ubuntu Default Page: It works</title>
```

一緒にやってみよう

- ・a) 次のコマンドでdockerが作るプロセスツリーを確認しよう

```
$ ps auxf
```

- ・b) ホストで直接apache2を立ち上げてみる。

プロセスツリーを比較しよう

```
$ sudo apt-get install apache2 && systemctl start apache2
```

3.

Linux Namespace

「部屋」の存在を
確認しよう

Linux Namespaceを「観察」する

- ・先ほど立ち上げたコンテナのIDを確認し、「アタッチ」する

```
$ docker ps  
CONTAINER ID IMAGE      COMMAND          ... NAMES  
88b7e296fe92 kyusec-1 "/usr/sbin/apache2..." ... angry_rosalind  
$ docker exec -ti 88b7e296fe92 bash
```

- ・その状態でコンテナ内部のネットワークを確認する

```
# ip a
```

- ・そのネットワークと、ホストのネットワークを比較する。

```
# exit  
$ ip a
```

コンテナの中、独立したネットワークがある

- ・先ほど確認したようにコンテナは同じマシンにあるプロセス。にもかかわらず、独立したネットワークが割り当たっている。

```
root@88b7e296fe92:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
9: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
        valid_lft forever preferred_lft forever
```

```
vagrant@localhost:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:33:82:8a brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe33:828a/64 scope link
        valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:d0:cd brd ff:ff:ff:ff:ff:ff
    inet 192.168.98.202/24 brd 192.168.98.255 scope global enp0s8
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fed0:30cd/64 scope link
        valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0c:4d:e3:43 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:cff:fe4d:e343/64 scope link
        valid_lft forever preferred_lft forever
10: veth0910d7f@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether ae:ad:6a:50:8f:41 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::acad:6aff:fe50:8f41/64 scope link
        valid_lft forever preferred_lft forever
```

Namespace は他にもある

- ・つづいて、ホスト名も独立していることを確認しよう
コンテナ内部のホスト名も変更してみよう

```
$ docker exec -ti 88b7e296fe92 hostname  
$ hostname
```

Namespace は他にもある

- ・プロセスにはプロセスID (PID) があるが、
その採番が「独立している」ことを見てみよう
- ・同じプロセスに違うPIDが割り振られていることを確認しよう

```
$ docker exec -ti 88b7e296fe92 ps auxf  
$ ps auxf
```

Namespace は
どこで確認できる？

一緒にやってみよう

- ・a) コンテナの「ホストから見た」PIDを突き止めよう。

```
$ ps auxf | grep -A 6 docker[d] # ヒントとなるコマンド
```

- ・b) procfsについて調べ、以下のディレクトリを確認しよう

```
$ sudo ls -l /proc/$PID/ns # $PID はコンテナのPID
```

- ・c) ホストの /proc/\$PID/ns とどう違うか、目`diff`しよう

```
$ sudo ls -l /proc/self/ns
```



```
vagrant@localhost:~$ sudo ls -l /proc/5331/ns
total 0
lrwxrwxrwx 1 root root 0 Sep 25 02:18 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Sep 25 01:15 ipc -> ipc:[4026532132]
lrwxrwxrwx 1 root root 0 Sep 25 01:15 mnt -> mnt:[4026532130]
lrwxrwxrwx 1 root root 0 Sep 25 00:53 net -> net:[4026532135]
lrwxrwxrwx 1 root root 0 Sep 25 01:15 pid -> pid:[4026532133]
lrwxrwxrwx 1 root root 0 Sep 25 02:18 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Sep 25 01:15 uts -> uts:[4026532131]
```

```
vagrant@localhost:~$ sudo ls -l /proc/self/ns
total 0
lrwxrwxrwx 1 root root 0 Sep 25 02:20 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 net -> net:[4026531957]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Sep 25 02:20 uts -> uts:[4026531838]
```

一緒にやってみよう(続き)

- ・d) ネットワーク名前空間「のみ」にアタッチしてみよう。

アタッチ後、以下を確認しよう

- d1) ip a の実行結果はホスト/dockerと比べどうか？

- d2) hostname の実行結果はどうか？

```
$ sudo nsenter --net -t $PID
```

- ・→ 部屋にはさらに内部での「分類」がある

4.

cgroup

「仕事の材料」は
どのように割り振られる？

一緒にやってみよう

- ・a) さっきのコンテナの「メモリの割り当て」を確認しよう

```
$ CID=$(docker inspect -f '{{.ID}}' 88b7e296fe92)  
$ sudo cat /sys/fs/cgroup/memory/docker/$CID/memory.usage_in_bytes  
$ sudo cat /sys/fs/cgroup/memory/docker/$CID/memory.limit_in_bytes
```

- ・b) より、割り当ての少ないコンテナを作り、比較しよう

```
$ CID2=$(docker run --memory=4m -d kyusec-1); echo $CID2  
$ sudo cat /sys/fs/cgroup/memory/docker/$CID2/memory.usage_in_bytes  
$ sudo cat /sys/fs/cgroup/memory/docker/$CID2/memory.limit_in_bytes
```

さらにやってみよう

- ・c) メモリ割り当て制限なし、制限あり、のコンテナにそれぞれ入り、aptなどの操作をして、どちらが挙動が遅いか比較してみよう

```
$ docker exec -ti $CID bash  
$ docker exec -ti $CID2 bash
```

- ・d) 直接メモリの割り当てを変更し、メモリ逼迫的な挙動が改善することを確認してみよう

```
$ echo '128m' | sudo tee /sys/fs/cgroup/memory/docker/$CID2/memory.limit_in_bytes  
$ docker exec -ti $CID2 bash  
root@ebb02470253b:/# apt install ruby ...
```

さらにやってみよう(応用編)

・e) cgroupを自分で作って、値をセットしてみる。

その環境でOOM Killerを発動させてみよう

```
$ sudo mkdir /sys/fs/cgroup/memory/kyusec-2
$ sudo ls -l /sys/fs/cgroup/memory/kyusec-2
$ echo 0 | sudo tee /sys/fs/cgroup/memory/kyusec-2/memory.swappiness
$ echo 4m | sudo tee /sys/fs/cgroup/memory/kyusec-2/memory.limit_in_bytes

$ echo $$ | sudo tee /sys/fs/cgroup/memory/kyusec-2/tasks
$ ruby -e \
  'GC.disable;ha=Hash.new;loop{1000.times{ha[rand(2**20).to_s.to_sym]=rand(2**20).to_s.to_sym}}'
$ echo 1g | sudo tee /sys/fs/cgroup/memory/kyusec-2/memory.limit_in_bytes # 割り当てを変える
$ ruby -e \
  'GC.disable;ha=Hash.new;loop{1000.times{ha[rand(2**20).to_s.to_sym]=rand(2**20).to_s.to_sym}}'
# 即死しないことが確認できる
```

cgroupでコントロールできるリソース

- ・CPU、メモリ、ディスクIOの帯域、プロセス数、など...
- ・追加課題: pids controllerを利用し、コンテナ内部での「fork bomb」を防いでみよう。
(時間があれば、一緒にデモをやってみます...)



5.

簡単なコンテナを
自作してみよう

事前準備: Haconiwa のインストール

- Haconiwa: @udzura らにより開発された、mrubyで設定やフックを記述できるLinuxコンテナランタイム。
- Ubuntu に version 0.10.0.alpha2を入れよう



```
$ curl -s https://packagecloud.io/install/repositories/udzura/haconiwa/script.deb.sh \
| sudo bash
$ sudo apt-get install haconiwa=0.10.0~alpha2-1
$ haconiwa version
haconiwa: v0.10.0.alpha2
```

<https://packagecloud.io/udzura/haconiwa>

事前準備: イメージ (rootfs) を作る

```
$ sudo mkdir -p /tmp/haconiwa/kyusec-3  
$ docker export 88b7e296fe92 | \  
sudo tar -xv -f - --C /tmp/haconiwa/kyusec-3
```

```
$ sudo chroot /tmp/haconiwa/kyusec-3  
root@localhost:/# ls -l /  
total 76  
drwxr-xr-x 2 root root 4096 Sep 25 07:52 bin  
drwxr-xr-x 2 root root 4096 Apr 12 2016 boot  
drwxr-xr-x 4 root root 4096 Sep 25 07:53 dev  
...
```

簡単なコンテナを作つてみよう

```
$ sudo haconiwa init --bridge; haconiwa init kyusec-3.haco
# 以下の行を追加、編集

Haconiwa.define do |config| #...
  config.init_command = ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
#...
  root = Pathname.new("/tmp/haconiwa/kyusec-3")
#...
  config.network.container_ip = "10.0.0.10"
  config.network.namespace = config.name
#...
  config.capabilities.allow "cap_net_bind_service"
  config.capabilities.allow "cap_kill"
end
```

起動してみよう

```
$ sudo haconiwa run kyusec-3.haco
Create lock: #<Lockfile path=/var/lock/.haconiwa-902340a9.hacolock>
Container fork success and going to wait: pid=10494
AH00557: apache2: apr_sockaddr_info_get() failed for haconiwa-902340a9
AH00558: apache2: Could not reliably determine the server's fully qualified
domain name, using 127.0.0.1. Set the 'ServerName' directive globally to
suppress this message

## 別ターミナルで

$ curl -s 10.0.0.10 | grep '<title>'

<title>Apache2 Ubuntu Default Page: It works</title>
```

スクラッチで
作ってみよう

hacorb/hacoirb コマンド

```
$ hacorb --version  
mruby 1.4.0 (2018-1-16)  
$ hacoirb  
mirb - Embeddable Interactive Ruby Shell  
  
> puts Dir.pwd  
/home/vagrant  
=> nil
```



Container Programming

入門

一緒に
Rubyのコードを
書いてみます (vimで)

まずは、fork/exec/chroot するだけ

[udzura/section-2/mycon.rb]

```
pid = Process.fork do
  Dir.chroot "/tmp/haconiwa/kyusec-3"
  Dir.chdir "/"
  Exec.execve ENV, "/usr/sbin/apache2ctl", "-D", "FOREGROUND"
end
p(Process.waitpid2 pid)
```

```
$ sudo hacorb udzura/section-2/mycon.rb
AH00557: apache2: apr_sockaddr_info_get() failed for localhost
...
# 別ターミナル

$ curl localhost -s | grep title
<title>Apache2 Ubuntu Default Page: It works</title>
```

ネットワークの準備

```
$ sudo ip netns add kyusec-3-demo  
$ sudo ip link add kyusec-3-host type veth peer name kyusec-3-guest  
$ sudo brctl addif haconiwa0 kyusec-3-host  
$ sudo ip link set kyusec-3-guest netns kyusec-3-demo up  
$ sudo ip link set kyusec-3-host up  
$ sudo ip netns exec kyusec-3-demo ip link set lo up  
$ sudo ip netns exec kyusec-3-demo ip addr add 10.0.0.20/24 dev kyusec-3-guest  
  
$ ping 10.0.0.20
```

Namespaceを独立させる

[udzura/section-2/mycon.rb] diff

```
--- sample1/mycon.rb.rev1      2018-09-25 23:27:55.342689233 -0700
+++ sample1/mycon.rb       2018-09-25 23:30:32.719900439 -0700
@@ -1,4 +1,8 @@
 pid = Process.fork do
+ Namespace.setns(
+   Namespace::CLONE_NEWNET,
+   fd: File.open("/var/run/netns/kyusec-3-demo", 'r').fileno
+ )
 Dir.chroot "/tmp/haconiwa/kyusec-3"
 Dir.chdir "/"
 Exec.execve ENV, "/usr/sbin/apache2ctl", "-D", "FOREGROUND"
```

```
$ curl localhost
curl: (7) Failed to connect to localhost port 80: Connection refused
$ curl 10.0.0.20 -s | grep title
<title>Apache2 Ubuntu Default Page: It works</title>
```

cgroupでメモリ制限をする

[udzura/section-2/mycon.rb] diff

```
--- sample1/mycon.rb.rev2      2018-09-25 23:33:37.330883578 -0700
+++ sample1/mycon.rb       2018-09-25 23:37:44.894926446 -0700
@@ -1,4 +1,9 @@
+limit = ENV['MEMORY_LIMIT'] || "128m"
 pid = Process.fork do
+ Dir.mkdir "/sys/fs/cgroup/memory/kyusec-3-demo" rescue nil
+ system "echo 0 > /sys/fs/cgroup/memory/kyusec-3-demo/memory.swappiness"
+ system "echo #{limit} > /sys/fs/cgroup/memory/kyusec-3-demo/memory.limit_in_bytes"
+ system "echo #{Process.pid} > /sys/fs/cgroup/memory/kyusec-3-demo/tasks"
 Namespace.setns(
   Namespace::CLONE_NEWNET,
   fd: File.open("/var/run/netns/kyusec-3-demo", 'r').fileno
```

```
$ sudo hacorb udzura/section-2/mycon.rb
$ sudo env MEMORY_LIMIT='1m' hacorb udzura/section-2/mycon.rb # 比較しよう
```

スクリプト全体

[sample/mycon.rb] all

```
limit = ENV['MEMORY_LIMIT'] || "128m"
pid = Process.fork do
  Dir.mkdir "/sys/fs/cgroup/memory/kyusec-3-demo" rescue nil
  system "echo 0 > /sys/fs/cgroup/memory/kyusec-3-demo/memory.swappiness"
  system "echo #{limit} > /sys/fs/cgroup/memory/kyusec-3-demo/memory.limit_in_bytes"
  system "echo #{Process.pid} > /sys/fs/cgroup/memory/kyusec-3-demo/tasks"
  Namespace.setns(
    Namespace::CLONE_NEWNET,
    fd: File.open("/var/run/netns/kyusec-3-demo", 'r').fileno
  )
  Dir.chroot "/tmp/haconiwa/kyusec-3"
  Dir.chdir "/"
  Exec.execve ENV, "/usr/sbin/apache2ctl", "-D", "FOREGROUND"
end
p(Process.waitpid2 pid)
```

ここまでまとめ

- ・コンテナは、OSから見ると通常のプロセスである
- ・OSのコンテナ化の機能を組み合わせて、プロセスの独立性を高め、
独自にリソースを割り当てて、VMのように使うことができる
- ・コンテナ化の機能はたくさんある
- ・基本「システムコール」であるので、プログラミングでいじれる





休憩

s³

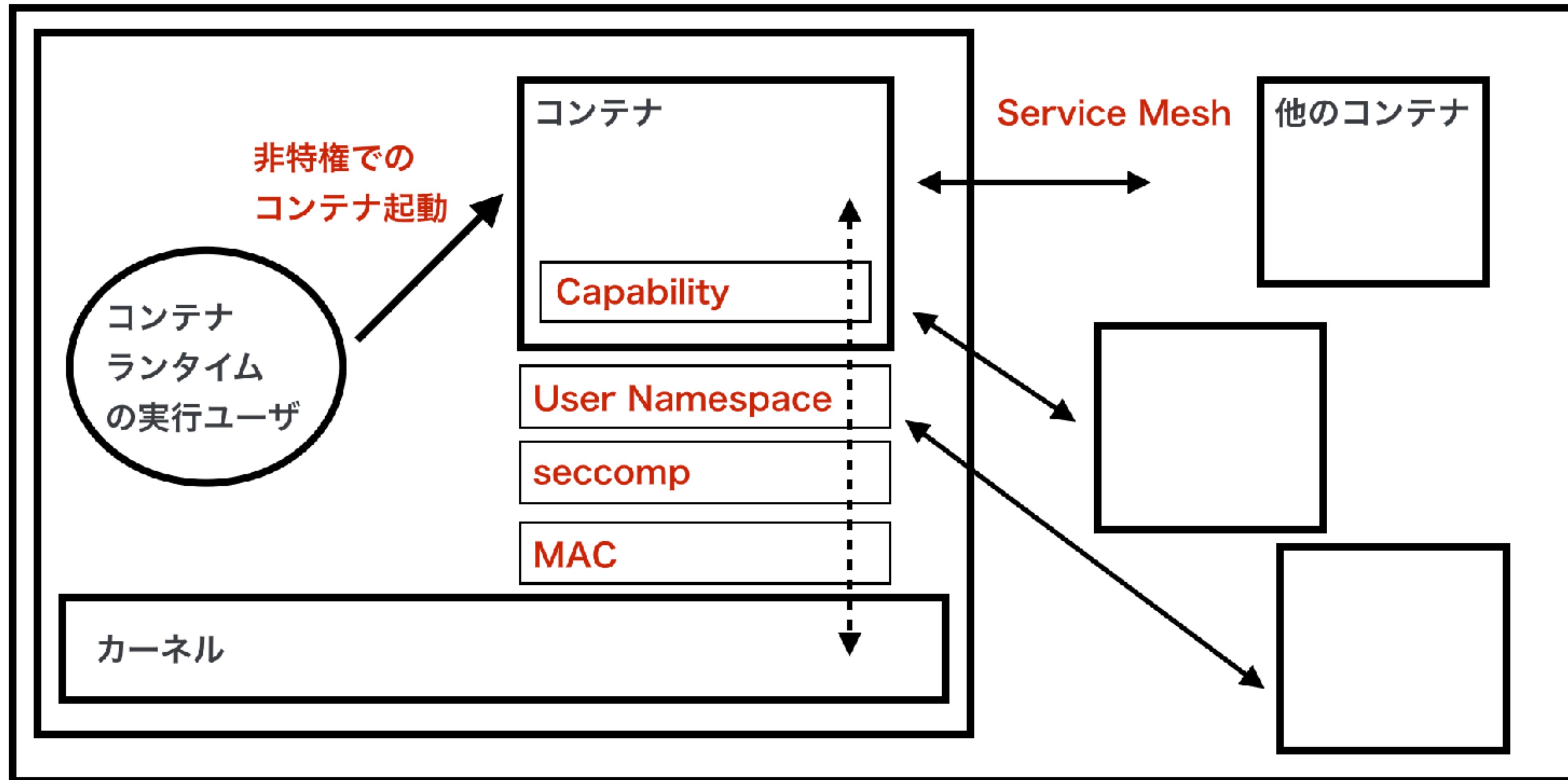
コンテナに対する攻撃の考え方

コンテナの中身は完全に理解した

- ・あらためて上から下まで眺めてみよう

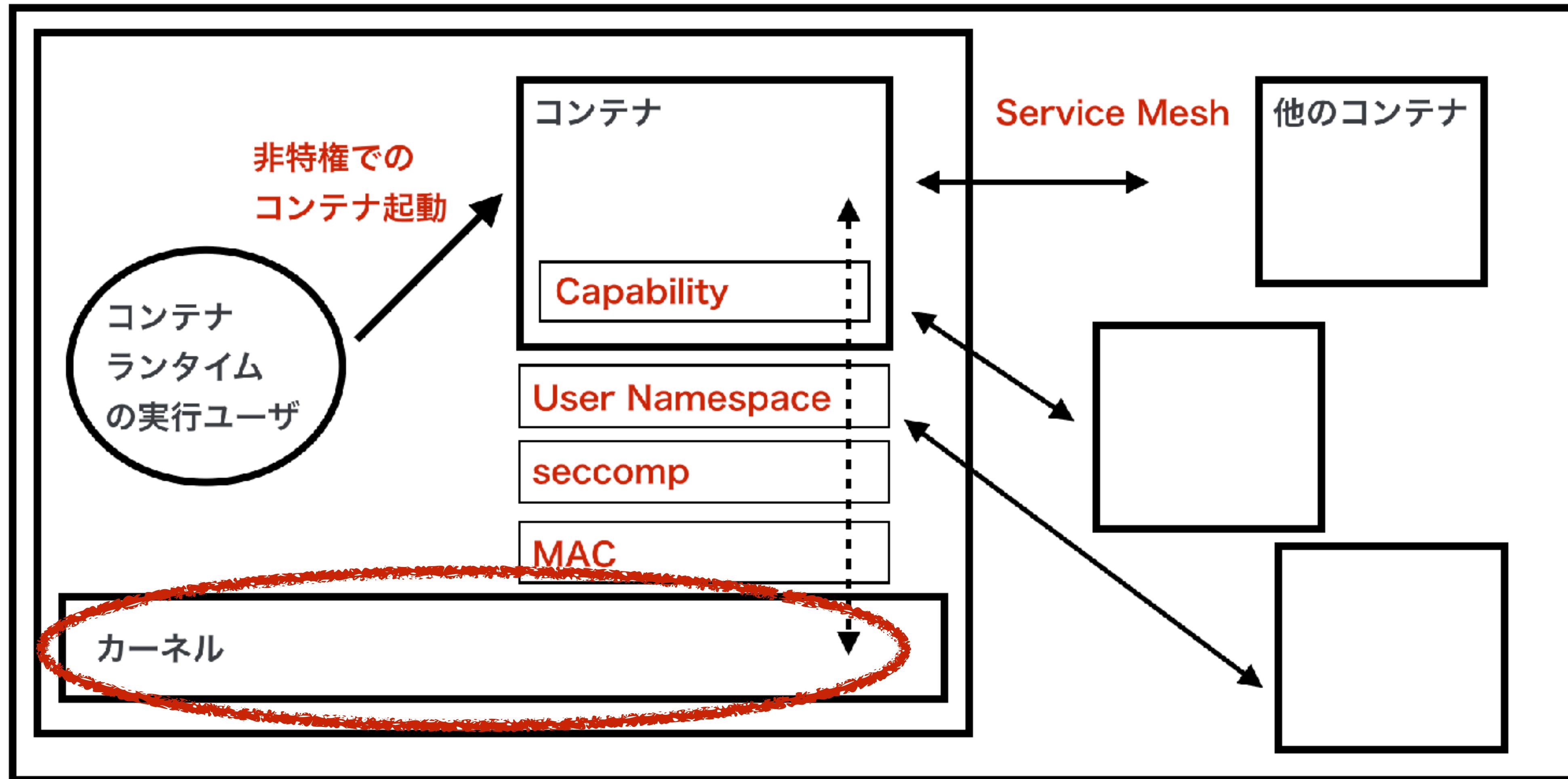


udzura@WSA研 #2 の資料より



どのような観点で
攻撃対策を考えれば良いか

OS/カーネル自身の脆弱性をつく



OS/カーネル自体の脆弱性をつく

- ・コンテナは一般的のプロセスの属性を変更して実現される。すなわち、ホストOSから見ると通常のプロセスに過ぎない。
- ・よって、ホストOS/カーネルに脆弱性があると、その影響を受ける。
- ・例: カーネルエクスプロイト: 権限昇格で見えないものが見えてしまう
- ・例: DoS: コンテナの中から、ホストの可用性に影響を与えると、ホスティングの場合などに他のユーザに影響を与えてしまう

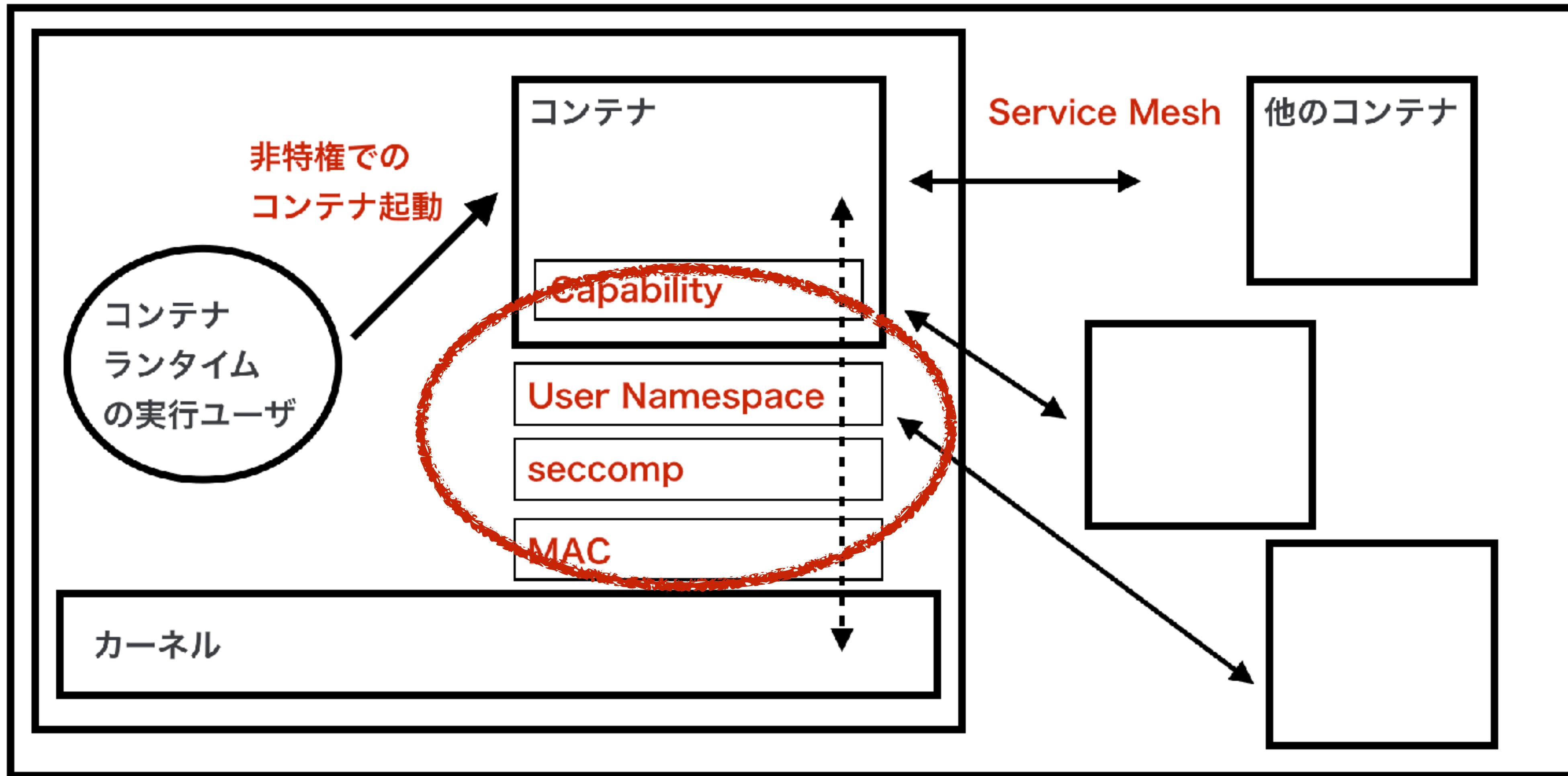


FYI: Container-Optimized OS

- ・GKEの例:
 - ・<https://cloud.google.com/container-optimized-os/docs/concepts/security?hl=ja>
- ・ミドルウェア、ユーザ、カーネルオプションなどを最低限の設定にしている。これにより、更新を容易にする、余計な攻撃を受けづらくする、など多くのメリットがある



コンテナ化の機能の不備、抜け穴をつく



コンテナ化の機能の不備、抜け穴をつく

- ・コンテナは、説明した通り、いくつかの権限分離、独立化のための機能の組み合わせである。
- ・これらの機能の理解が甘いと、不用意に設定を緩和してしまい攻撃につながる場合がある
 - ・e.g. docker run --privileged
- ・原則としてコンテナ側で防ぐべき「穴」を防げていなかった場合もある。これはランタイム側の脆弱性となってしまう
 - ・e.g. コンテナ内部からの /proc/acpi への攻撃の例

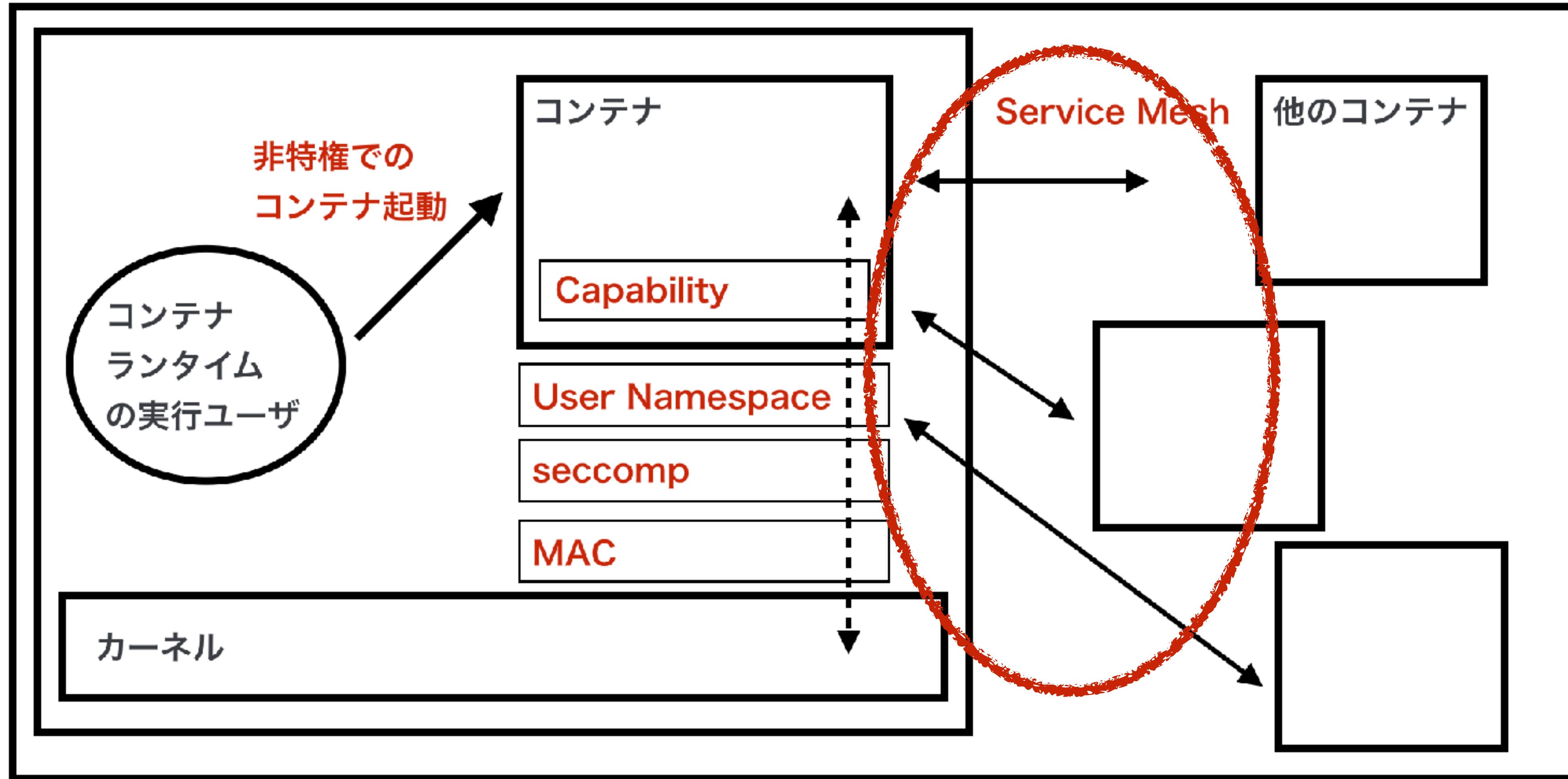
<https://jvndb.jvn.jp/ja/contents/2018/JVNDB-2018-007686.html>

Introduction to Container Security

- ・Docker 公式の、セキュリティ観点からのコンテナ機能の解説。
- ・今日話したような内容が詳しく載っている
 - ・https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf
- ・近藤うちおさんの資料でもまとめている。参照のこと
 - ・<https://udzura.hatenablog.jp/entry/2018/05/22/234036>



ネットワークなどの設定不備をつく



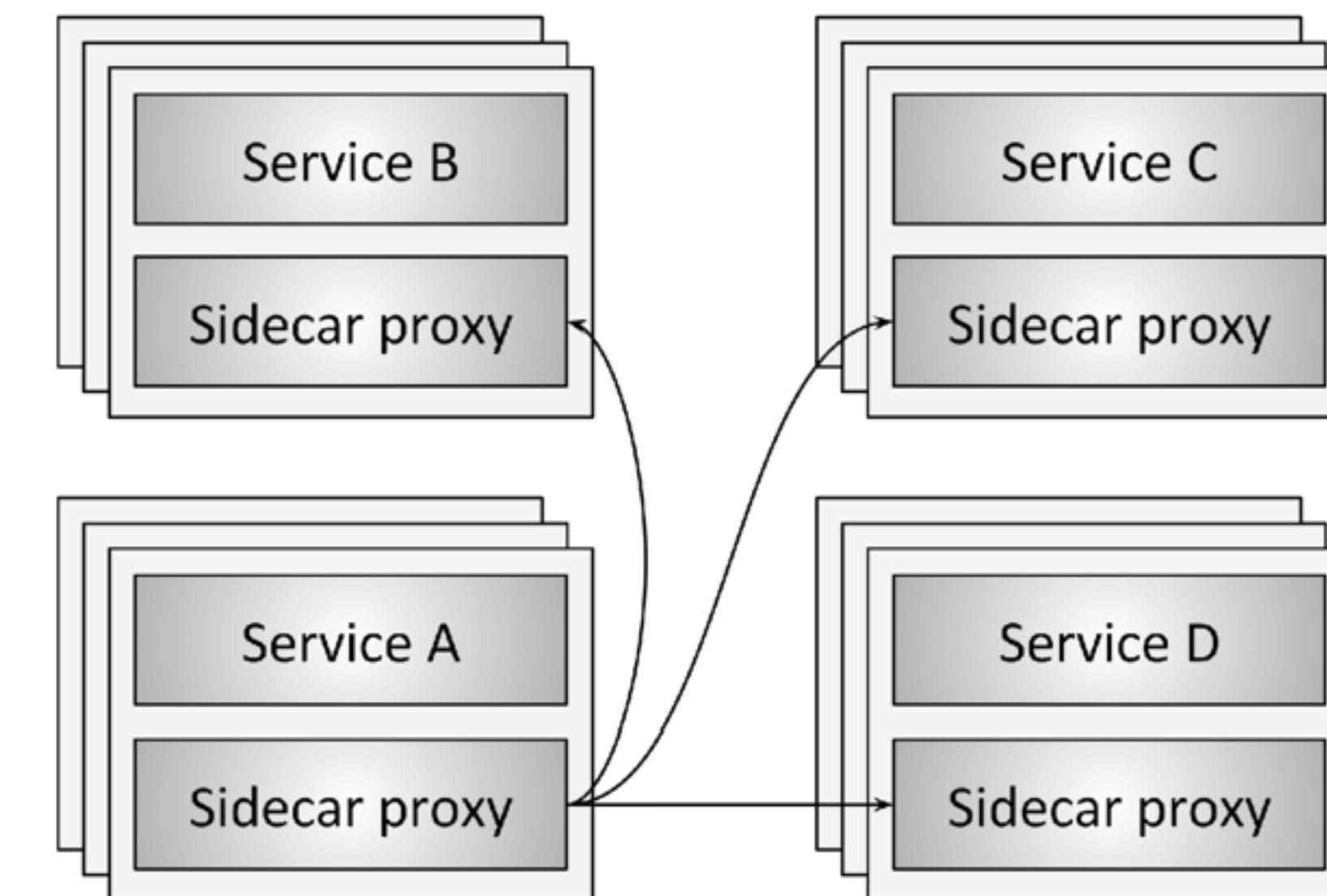
ネットワークなどの設定不備をつく

- ・コンテナ同士は、複数のコンテナを組み合わせて利用するのが普通
- ・そのため（クラウドプラットフォームなど同様）アクセス権限の設定をする必要がある。c.f. セキュリティグループ
- ・ネットワーク的なものでは：
 - ・Kubernetesの、RBACベースのネットワーク制限
 - ・サービスメッシュの導入によるアクセス制限、流量制限
 - ・cgroupのnet_clsコントローラ+tc/iptables

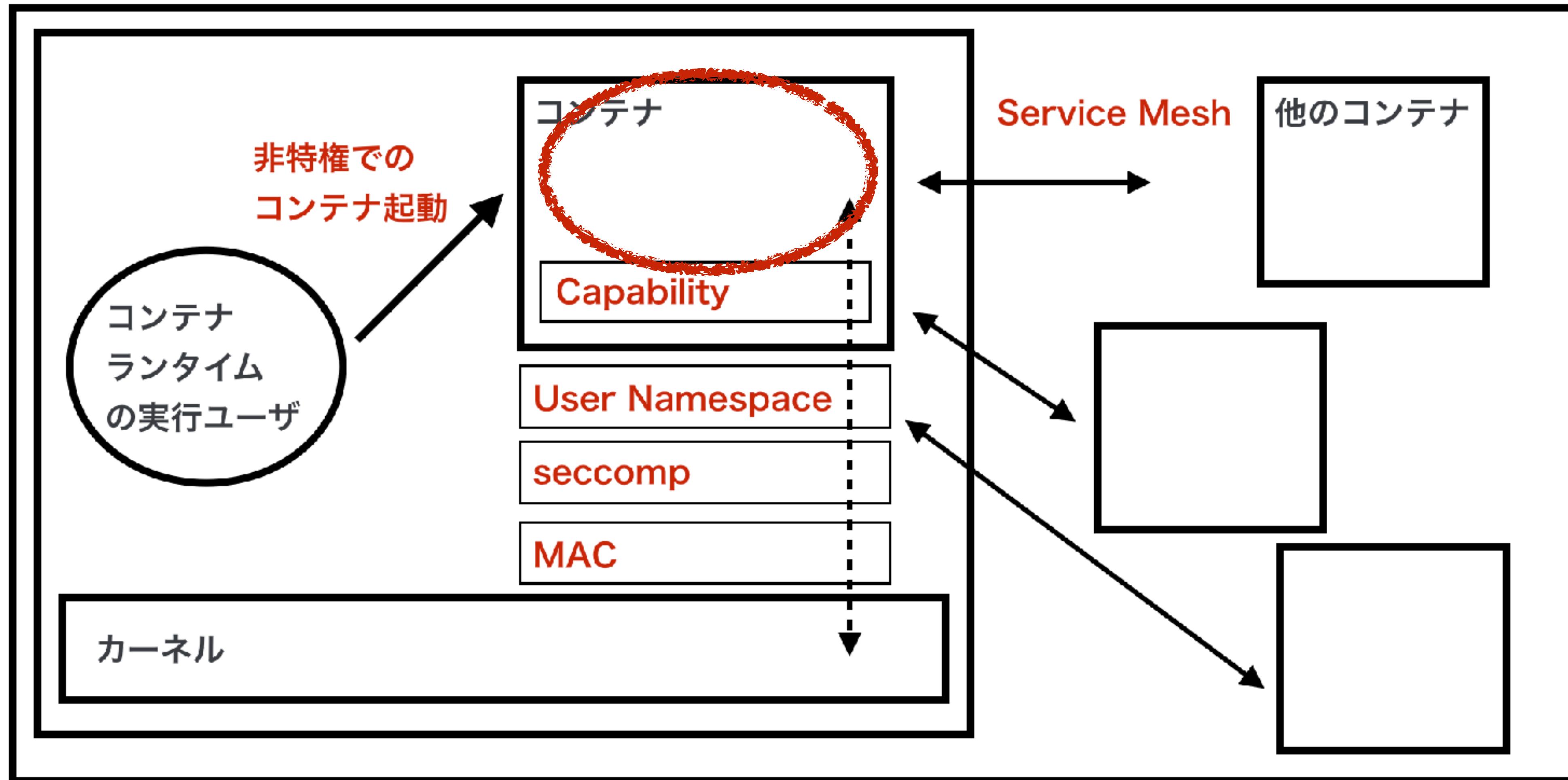


FYI: サービスマッシュ

- ・ログ、アクセスコントロール、トレーシングなどのビジネスロジックから離れた問題を、コンテナ間のネットワーク層で解決する考え方。
- ・例えばサイドカーコンテナ（コンテナに付属するコンテナ）などにプロクシを立て、すべてのHTTPアクセスを経由させることで、ネットワークレベルでのアクセス制限や流量制御をおこなう。



(アプリケーション自体の脆弱性をつく)



※ この場合、一般のアプリケーションにある脆弱性と同様の議論となる。省略

FYI: スイスチーズモデル

- ・コンテナに関するセキュリティ機構は、一部機能として重複しているものもある (ex. Capability と seccomp の両方で、chroot操作を禁止できるなど)
- ・一方で、複数の機構を用いて操作を制限することで、ある機能が脆弱性などでバイパスされてしまっても、別の機能で攻撃を防ぐ/緩和することができる



SA

実践！
コンテナ攻撃ワークショップ

<https://flic.kr/p/cg2J5N>



@mrtc0 にバトンタッチ

