systemd エッセンシャル

レッドハット株式会社 ソリューションアーキテクト
森若和雄 <kmoriwak@redhat.com>
2020-03-22

この資料の位置づけ

- 対象 : systemd をとりあえず使えているけど中で何をやっているかブラックボックスで気持ち悪い or 納得感がない人
- 目的 : RHEL8 で systemd の出力やログを読んで理解できないときに、どこを調べたらいいか見当がつくようになること
- 前提 : fork, exec システムコール、シグナル、ソケットくらいの言葉がなんとなくわかる程度の UNIX 系 OS についての知識

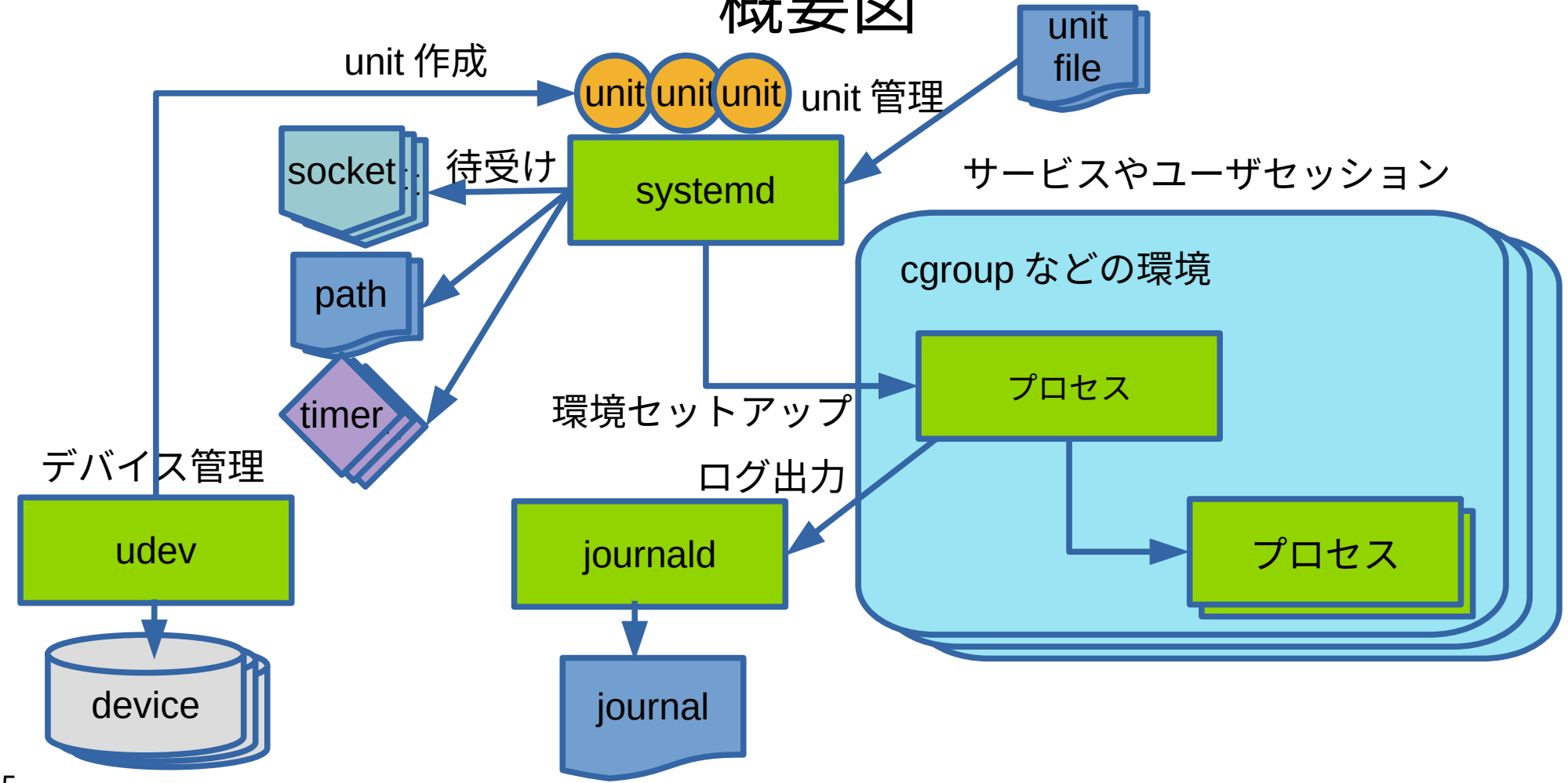
この資料で扱わないこと

- RHEL8 より前の systemd から何が変わったかの差分
- unit 作成方法や コマンドの how to use
- ディレクティブの網羅的な一覧や解説 (systemd.directives から systemd の man page を探すのがおすすめです)
- dracut を含む起動処理の詳細
- Optional な機能の一部
  - systemd-networkd, systemd-resolved, systemd-timesyncd
  - systemd-nspawn, machinectl
  - systemd-boot, Portable Service
  - デスクトップ環境とのインテグレーション全般
- 個別のトラブルシュート (Red Hat の Knowledge base を検索するのがおすすめです)

agenda

- systemd は何をやるもの？
- systemd の unit
  - target unit: 同期ポイントを提供
  - service unit: サービス管理
  - socket, path, timer unit: イベントによる Activation
  - device, mount unit: ファイルシステムの mount
  - slice, service, scope unit: cgroup 管理
- ユーザセッション用 systemd
- アドホックなコマンド実行の管理 systemd-run
- journald によるロギング
- sysvinit からの移行
- systemd の動作を眺める
- 周辺ツールなど
- 参考資料

概要図



systemd は何をやるもの？

systemd って何？

- システムとサービスの管理をおこなうたくさんのサービス、ユーティリティ、ライブラリ群。 PID 1 の init 実装を含む。
- 「 sysvinit の置き換え」とよく言われますが単純な置き換えではなく linux の機能を活用する基盤として作り込まれています
- 目標は？
  - 起動の高速化
  - ディストリビューション独自実装の統廃合
  - ベストプラクティスの統合

systemd の普及

- 2010 年 systemd の最初のリリース
- 2011 年 Fedora に systemd を統合
- 2013 年 Debian に統合 (複数ある init 実装の一つとして)
- 2014 年 RHEL, SLES が systemd を統合
- 2015 年 Debian のデフォルト init 実装に選出

世にでてから約 10 年、一般的なディストリビューションで
標準的に利用されるようになってから約 5 〜 6 年

systemd はどのあたりをカバーする？

- 起動・終了・再起動に必要な処理全般
  - デバイスの検出・命名・初期化
  - fs の mount, autmount, 暗号化 block device 対応
  - サービス起動・管理
  - ロギング
  - 起動失敗時のレスキュー処理
  - パスワード確認
  - システムの locale, TimeZone, キーボード , 仮想コンソール
  - 電源管理 (サスペンド・ハイバネート・レジューム)
- サービス管理で典型的に必要な属性全般の管理
  - control groups, namespace, ulimit, 通知 , /tmp の掃除
  - ユーザセッション管理

それぞれをカバーする systemd 関連プログラム

- 起動・終了・再起動に必要な処理全般
  - デバイスの検出・命名・初期化　← udev
  - fs の mount, autmount, 暗号化 block device 対応 ← systemd-fstab-generator
  - サービス起動・管理　← systemd 本体
  - ロギング　← journald
  - 起動失敗時のレスキュー処理　← rescue.service
  - パスワード確認　← systemd-ask-password-*
  - システムの locale, TimeZone, キーボード , 仮想コンソール ← localectl, timedatectl
  - 電源管理 (サスペンド・ハイバネート・レジューム)　← systemd-sleep, udev, systemd-inhibit
- サービス管理で典型的に必要な属性全般の管理
  - control groups, namespace, ulimit, 環境変数 , /tmp の掃除 ← systemd 本体 , systemd-tmpfiles*
  - ユーザセッション管理 ← systemd-logind

宣言的なサービスの定義

- sysvinit ではシェルスクリプトを順次実行することで起動処理を行う
  - 編集すると rpm パッケージ等の更新時にトラブルが発生しがち
  - ulimit 等のリソース割り当てや設定方法が標準化されていない。
    - そのため自動でのカスタマイズや、それを更新時に維持することなどが難しい。
- systemd では宣言的にサービスを定義し、実行は shell ではなく systemd が行う

```
[Unit]
Description=Vsftpd ftp daemon
After=network.target

[Service]
Type=forking
ExecStart=/usr/sbin/vsftpd /etc/vsftpd/vsftpd.conf

[Install]
WantedBy=multi-user.target
```

従来からの互換性の維持

中身は大きく変わっていますが、操作の互換性を維持する工夫を実施

- sysvinit, LSB の init scripts から unit ファイルを自動生成
- インタフェースの互換性維持
  - 「ランレベル」への対応 : runlevelX.target, カーネルオプション等の維持
  - ソケット : /dev/initctl, /dev/log 互換ソケット
  - dbus: ConsoleKit 等の dbus インタフェースを維持
- 互換コマンドの提供
  - halt, init, poweroff, reboot, runlevel, shutdown, telinit など
  - service コマンドを実行すると systemctl を呼びだす

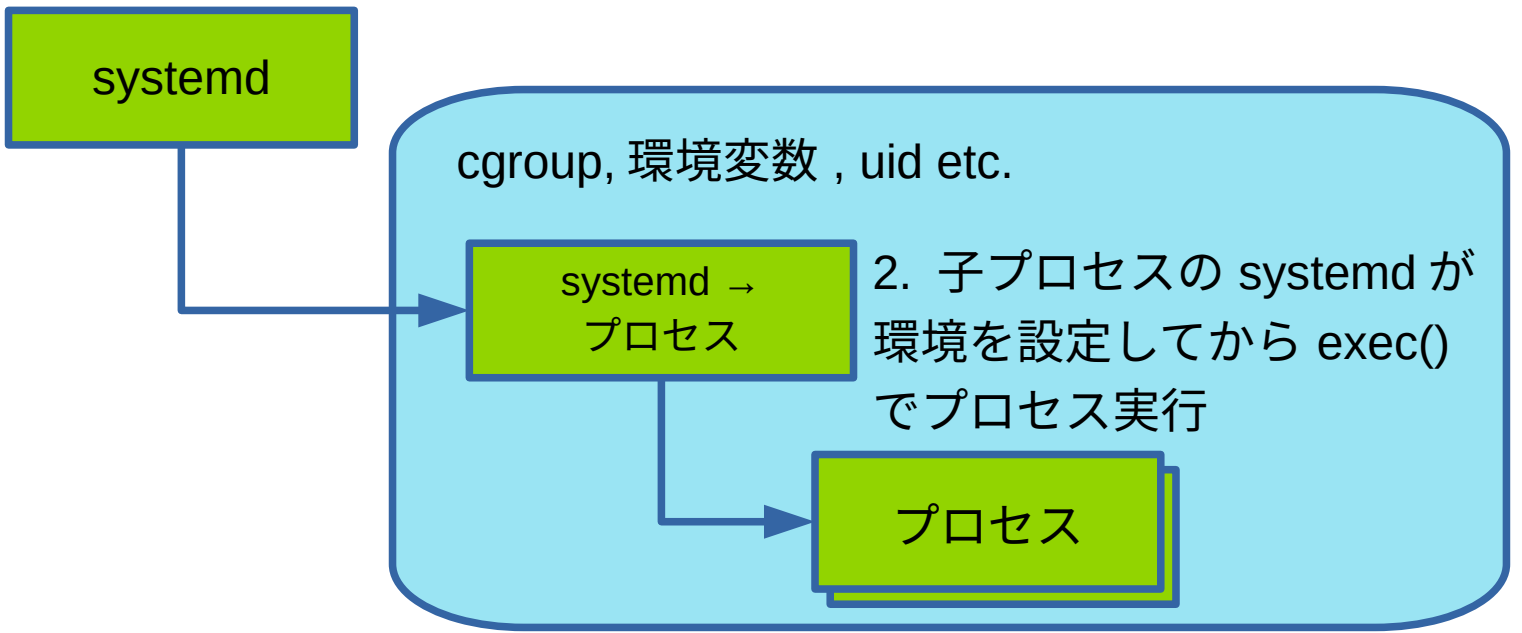
```
# service cups start
Redirecting to /bin/systemctl start  cups.service
```

systemd はプロセス実行環境を用意する

- systemd の主要な機能のひとつは実行環境の用意
  - UID, GID, 環境変数 , cwd, chroot, ulimit, capability, nice, cgroup, namespace, seccomp, taskset などを設定した上でプロセスを実行
- systemd は linux kernel が提供する多様な機能を統一的なインタフェースで提供

systemd がプロセス実行環境を用意する方法

1. systemd は fork() して、子プロセスを cgroup に入れる



3. プロセスがデーモン化や子プロセスの
生成などを行っても cgroup で追跡する

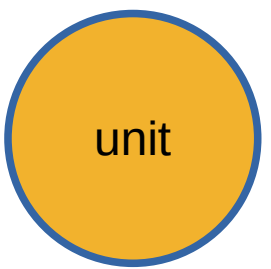
使い捨て unit の作成 systemd-run

- systemd が用意する環境を試すには、使い捨ての unit を作る systemd-run コマンドが便利
- systemd-run [オプション] < コマンド >
  - 例 1: # systemd-run -t env
    - サービス用に用意された環境変数を表示
    - -t をつけないと端末ではなくログへ出力される
  - 例 2: # systemd-run -t --uid=apache id
    - uid が変わっている
  - 例 3: # systemd-run -S -p ProtectHome=yes
    - service 用環境で shell を起動する。この例では /home 以下にアクセスできない (何もなくなっているように見える)
    - systemd で制限をかけた環境でプログラムをテストするときに便利。

unit
systemd の管理するモノ

unit とは？

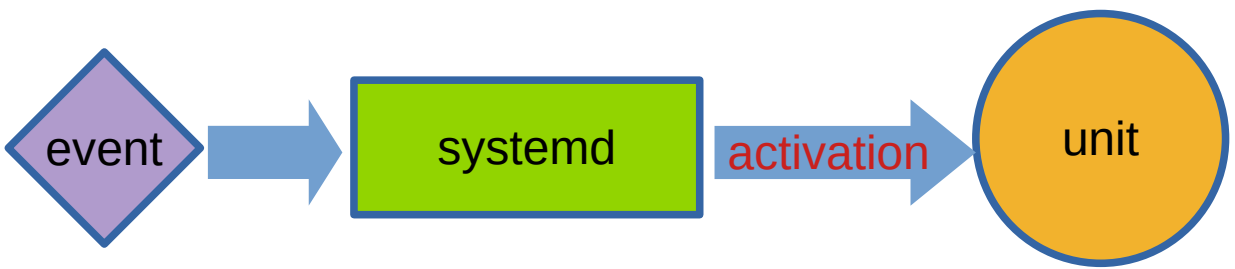
- システム管理に登場するいろいろなモノを抽象化した、 systemd で使われる概念
- 例：以下の「」内にあるものは全て unit
  「**ブロックデバイス sda2** 」を「 **/var に mount** 」して
  「**パス /var/cache/cups/org.cups.cupsd が存在**」すれば
  「**サービス cupsd** 」を起動して「**ソケット /run/cups/cups.sock** 」で待ちうける



systemd は unit 群を管理する

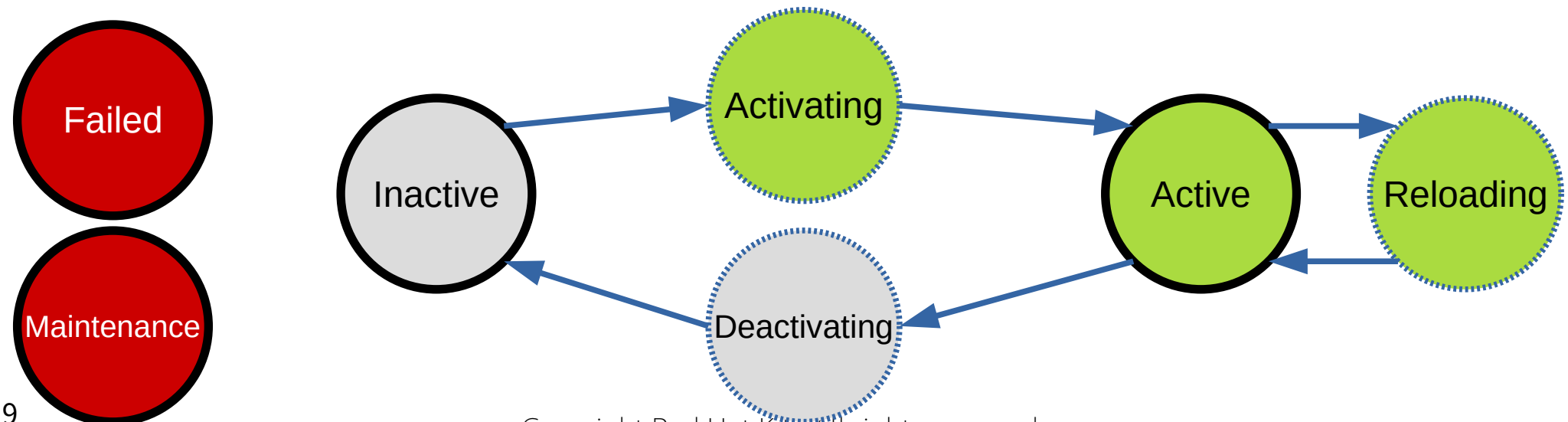
systemd はおおむね以下のような動作を無限に繰り返します：

- システムの起動、ハードウェアの挿抜、ユーザコマンド、タイムアウト、 unit 状態変化などのイベントを取得する
- イベントに対応する unit があれば、あらかじめ定義された制限に従って有効化 (Activation) や無効化などの job を行う。 unit の内容や依存関係から複数の job になることもある。
- systemd は job queue を持っていて複数の job を適切な順序で実施する



unit がとる状態

- 基本的な 2 状態 : Active, Inactive
- 中間的な状態 : Activating, Deactivating, Reloading
- 例外 : Failed, Maintenance



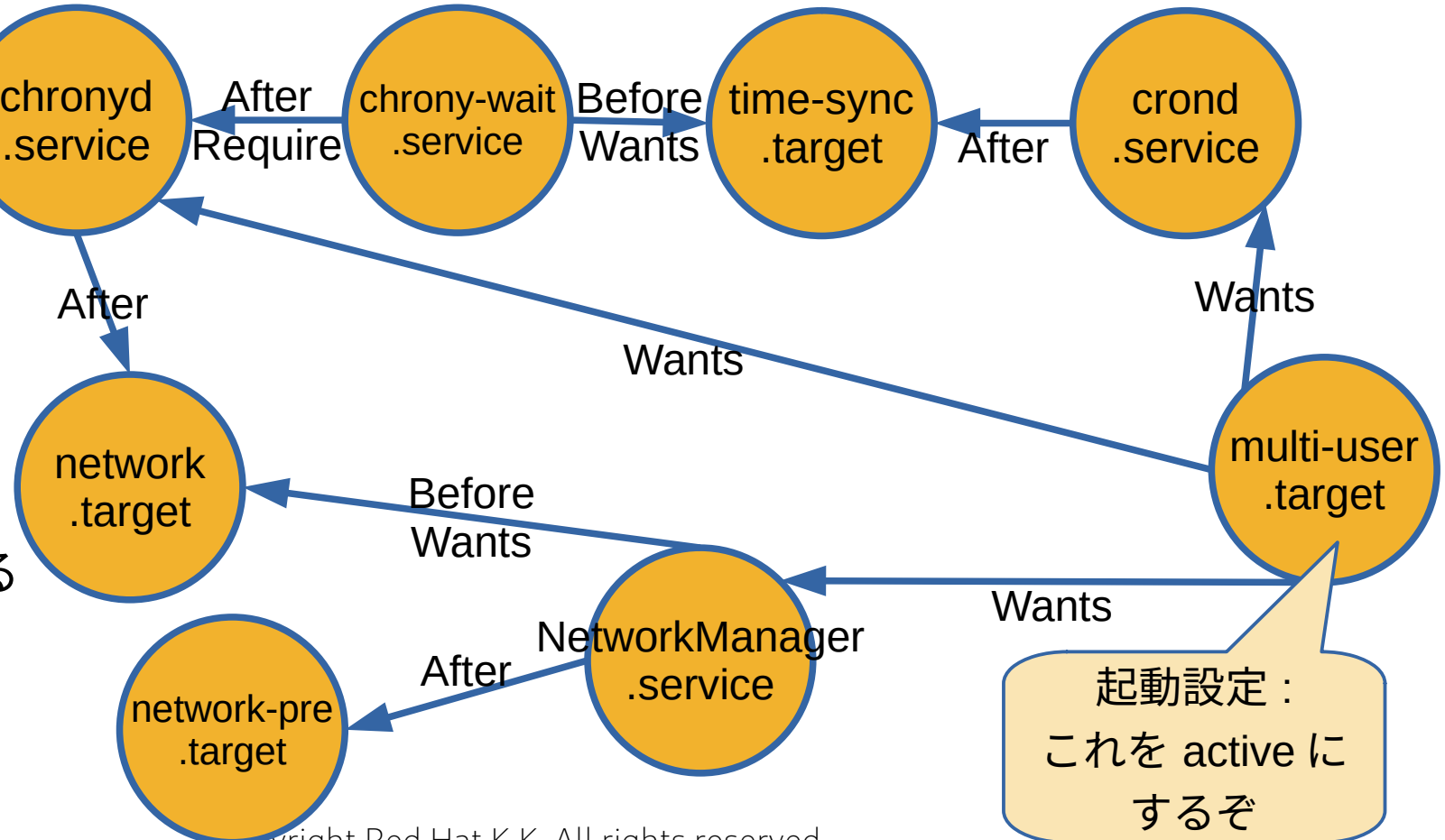
unit 間の依存関係

unit は他の unit と依存関係をもてる

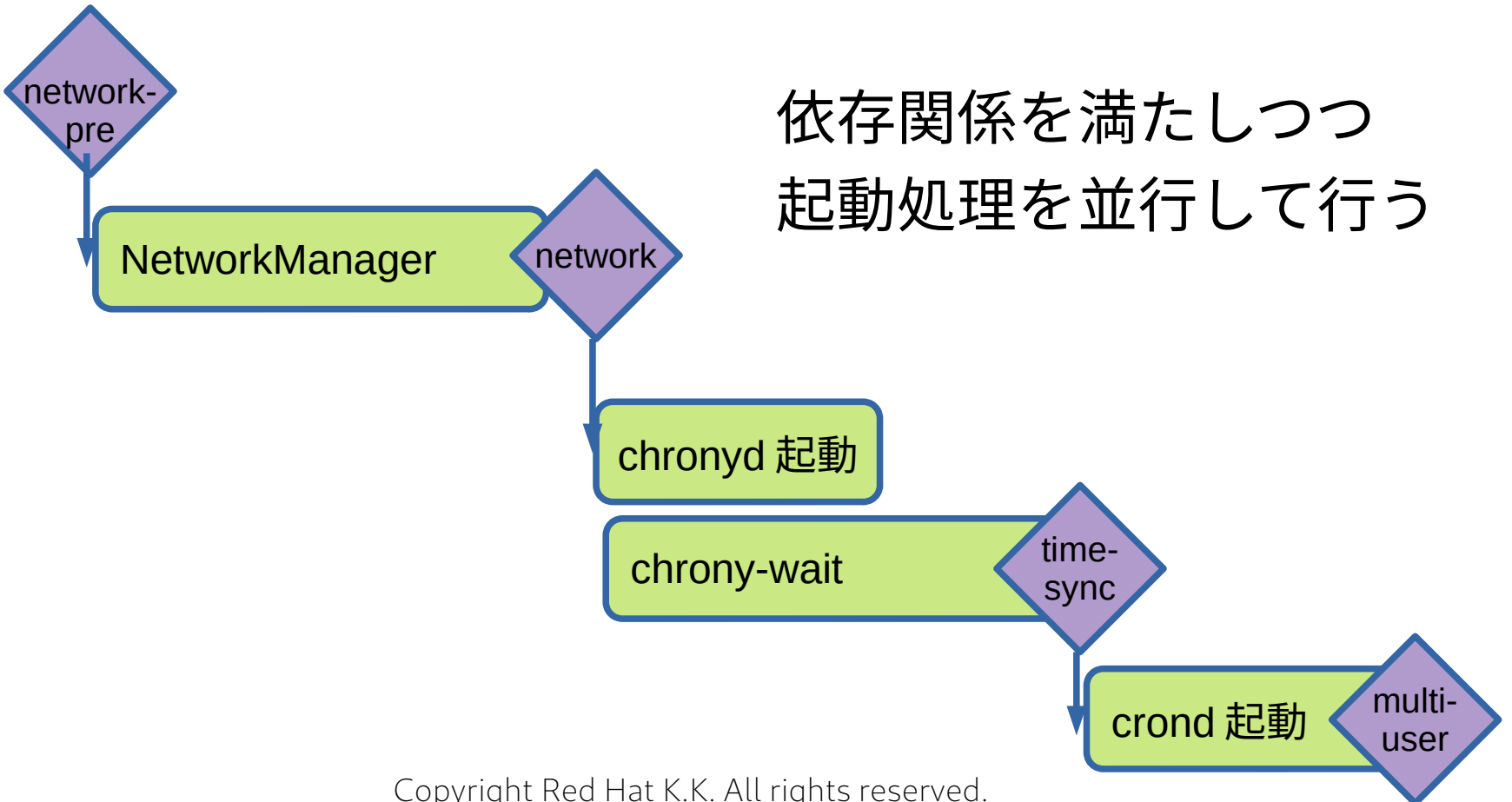
- **Wants**: B を active にするなら A も active にしたいが A が無い場合や失敗しても OK
- **Requires**: B を active にするなら A も active にしする。 A が成功しないと B は失敗
- **After**: B を activating にするのは A を activating にしたあと
- **Conflicts**: A と B は同時に Active になれない

systemd の unit 初期化イメージ

systemd は
多数の unit 間の
依存関係を見て
何を active に
するか、順序を
どうするか決める



systemd の unit 初期化イメージ (続)

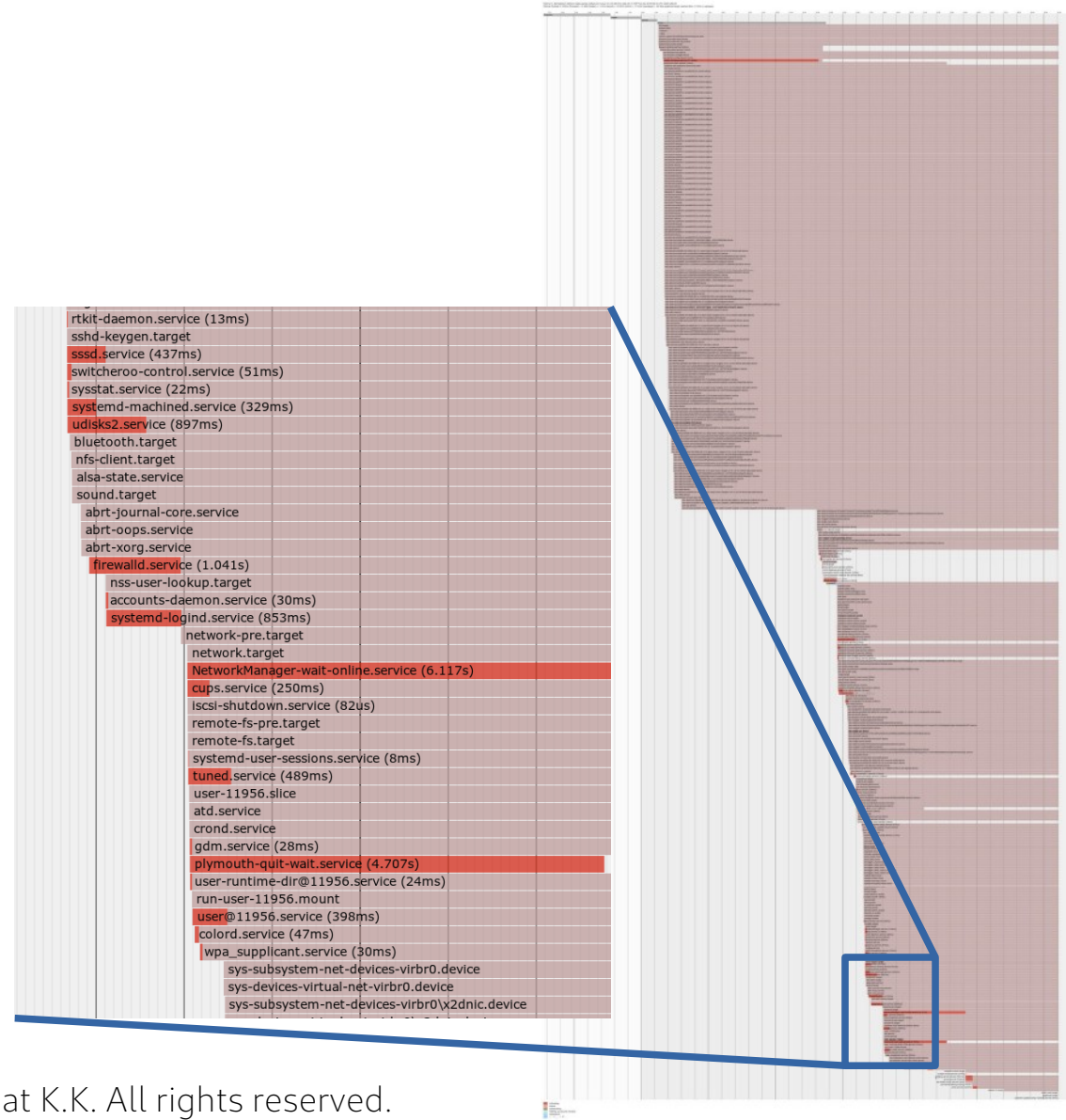


依存関係を満たしつつ
起動処理を並行して行う

実際の起動処理の様子

```
$ systemd-analyze plot
```

SVG 形式で起動時の各 unit の初期化開始・終了のタイミングを図示する。



systemd --test

- /lib/systemd/systemd --test --system とすると実際に起動用の unit や設定を読んで実行するべき job を計算した様子を出力する。(実際の起動処理はおこないません)

主要な Activation のきっかけ

- **Activation on boot**: 起動処理。通常は default.target unit を有効化する。
- **Socket-based Activation**: systemd が xinetd のようにソケット待ちうけを行い着信を契機に対応する service の unit を有効化してソケットを引きつぐ。socket unit と service unit の組み合わせで定義する。
- **Timer-based Activation**: systemd が時刻やタイムアウトなどを契機に service unit などを有効化する。 timer unit と service unit の組み合わせで定義する。
- **Device-based Activation**: linux kernel が検出したハードウェアの挿抜や変更を契機として device unit を作成し、そこからの依存関係で mount unit や service unit などを有効化する。 fstab での mount 処理などは device-based activation で実施される。

ユーザからの操作 : systemctl

- systemctl は systemd への主要な操作インタフェース
  - よく使うコマンド : 見る (show) 、状態 (status) 、有効化 (enable) 、無効化 (disable) 、開始 (start) 、終了 (stop) 、リロード (reload)

例 : cron デーモンに対応する unit crond.service の状態表示

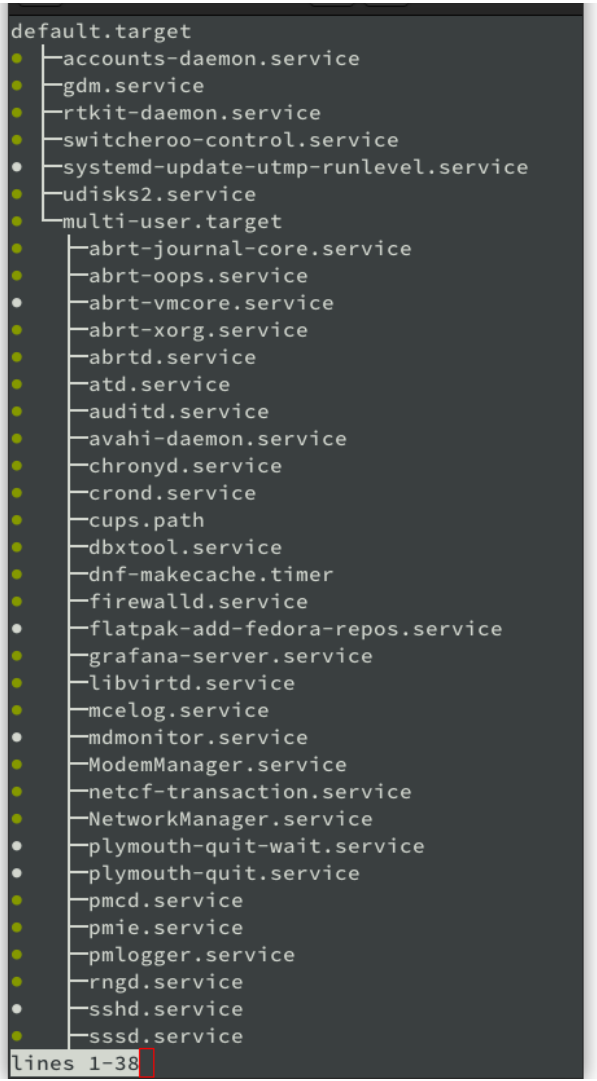
```
$ systemctl status crond.service
● crond.service - Command Scheduler
   Loaded: loaded (/usr/lib/systemd/system/crond.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2020-02-03 14:42:02 JST; 2 days ago
 Main PID: 1245 (crond)
    Tasks: 1 (limit: 28413)
   Memory: 10.0M
      CPU: 30.309s
   CGroup: /system.slice/crond.service
           └─1245 /usr/sbin/crond -n

Feb 06 11:10:01 snake.usersys.redhat.com CROND[147722]: (root) CMD (/usr/bin/redhat-interna>
Feb 06 11:15:01 snake.usersys.redhat.com CROND[148019]: (root) CMD (/usr/bin/redhat-interna>
```

unit の start/stop 操作

- systemd が unit を
  active にする (start) / inactive にする (stop)
  - `systemctl start crond.service`
  - `systemctl stop crond.service`
- 状態を変える必要がある場合だけ動作を行う
  - unit が既に active である場合、 start しようとしても何も実施しない。 unit が inactive な場合の stop も同様。
- 依存関係があれば指定以外の unit も操作を行う場合がある

実際のシステムで依存関係を見る

- `systemctl list-dependencies`
  - デフォルトでは起動処理に対応する default.target の依存関係を表示
- default.target に使われる unit
  - multi-user.target (runlevel 3 相当)
  - graphical.target (runlevel 5 相当)



unit の種類

- unit にはいくつかの種類がある。よく使うものを例示する。
  - **target**: 何もしない。依存関係や前後関係を定義するために使う
  - **service**: 何かのプロセスを実行してサービスを提供する
  - **socket**: TCP, UDP, IPv4, IPv6, socket, dbus, fifo などで待ちうける
  - **path**: ファイルやディレクトリが存在することや操作されたことを待ちうける
  - **timer**: サービス開始時から x 秒経過、前回の timer 起動時から x 秒経過、カレンダー上の日時などの時刻を待ちうける

関連する man page

- systemd(1) Concepts 節 : systemd の主なコンセプト、 unit の種類、 kernel コマンドラインオプション
- daemon(7) Activation 節 : Activation の説明
- systemd.special(7): systemd であらかじめ定義されている特殊な unit の紹介。 default.target など予約語のように扱われるものも多い。

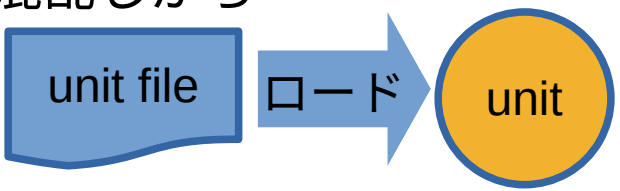
やってみよう

- systemctl list-units で unit の一覧を見る
- systemctl status <unit 名 > で unit の状態を見る
- systemctl show <unit 名 > でもっと細かい unit の状態を見る

unit file
unit の設定ファイル

unit の定義ファイル unit file

- systemd の unit 群のほとんどはファイルで定義される
- unit と unit file は明確に区別されるが慣れるまでは混乱しがち
  - `systemctl list-units` (unit の一覧)
  - `systemctl list-unit-files` (unit file の一覧)
  - `systemctl cat crond.service` (ある service に関連する unit file を出力)



- systemd は unit file をロードしてメモリ上の unit を作成する
  - systemctl daemon-reload (全 unit 定義ファイルを読み直す)
  - 不要な unit は自動的にアンロードされる (一覧を見るだけでもロードしなおされるので利用時にはあまり意識しない)

※unit file を書き変えただけでは動作に影響を及ぼさない場合があるので注意

unit file の例

/lib/systemd/system/httpd.service

```
[Unit]
Description=The Apache HTTP Server
Wants=httpd-init.service
After=network.target remote-fs.target nss-lookup.target httpd-init.service
Documentation=man:httpd.service(8)

[Service]
Type=notify
Environment=LANG=C
ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
KillSignal=SIGWINCH
KillMode=mixed
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

unit の例

$ systemctl show httpd

```
Type=notify
Restart=no
NotifyAccess=main
RestartUSec=100ms
TimeoutStartUSec=1min 30s
TimeoutStopUSec=1min 30s
TimeoutAbortUSec=1min 30s
RuntimeMaxUSec=infinity
WatchdogUSec=0
WatchdogTimestampMonotonic=0
RootDirectoryStartOnly=no
RemainAfterExit=no
GuessMainPID=yes
MainPID=0
ControlPID=0
FileDescriptorStoreMax=0
NFileDescriptorStore=0
```

```
StatusErrno=0
Result=success
ReloadResult=success
CleanResult=success
UID=[not set]
GID=[not set]
NRestarts=0
OOMPolicy=stop
ExecMainStartTimestampMonotonic=0
ExecMainExitTimestampMonotonic=0
ExecMainPID=0
ExecMainCode=0
ExecMainStatus=0
ExecStart={ path=/usr/sbin/httpd ; arg
ExecStartEx={ path=/usr/sbin/httpd ; a
（以下略）
```

unit file の文法

いわゆる INI ファイル形式

- [Section] でセクションを指定
- Key=Value 形式の行が並ぶ。 Key や条件により同じ Key を複数回書ける場合もある。
- Key はディレクティブと呼ばれる
- # または ; のあと改行まではコメント
- 改行は \ でエスケープできる
- 時間は”3min 10sec” のように単位をつけて書ける

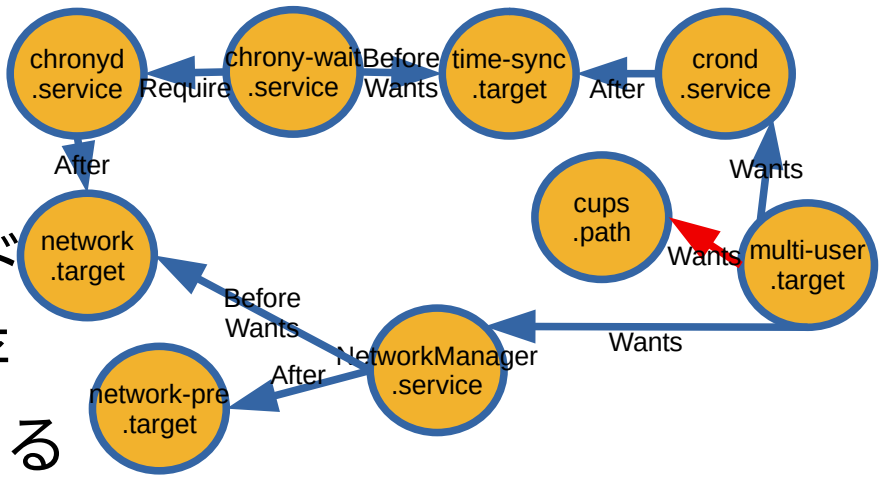
```
[Section A]
KeyOne=value 1
KeyTwo=value 2
# a comment
; another comment

[Section B]
Setting="something" "some thing" "..."
KeyTwo=value 2 \
       value 2 continued
TimeoutSec=”1hour 5min”
```

unit と unit file

- unit には unit file にある情報の他に以下が含まれる
  - デフォルト値
  - 他 unit からの依存関係の逆向きの依存関係
    - 他から After= で参照されていると Before= が作られるなど
  - 実行時に決まる情報 (時刻、実行結果、 PID など)

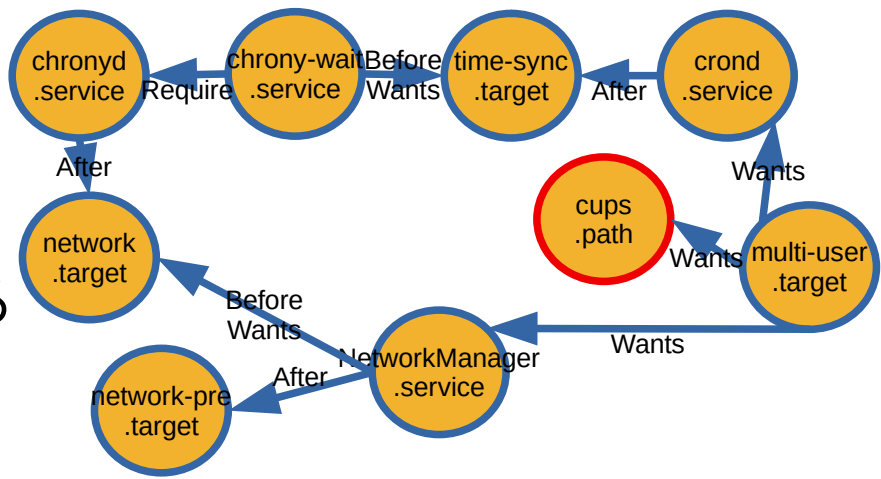
unit file の enable/disable 操作

- unit file に対して enable か、 disable かを指定する
  - enable すると Install 処理が行われる (典型的には multi-user.target などからの依存関係を定義する)
  - disable すると Uninstall 処理を行い Install での処理を元に戻す
- disable していても unit file はロードされ unit が作成される。他から依存されていれば active になる場合もある



unit file の mask/unmask 操作

- どういう依存関係を経由しても unit file から unit を作りたくないときは mask を行う
  - mask すると systemd はその unit file を無視してロードしない
  - unmask すると元にもどす
- mask した unit file で定義されていた unit へ、他の unit が依存していると依存関係を満たせず失敗する場合もある



unit file の参照パス

- systemd が unit file を探すディレクトリは複数ある
- RHEL の場合 systemd 関連の設定ディレクトリは以下。
  同名のファイルがあれば上が優先
- /etc/systemd (管理者による設定)
  → /run/systemd (実行時の自動設定など)
  → /lib/systemd (rpm パッケージで提供)
- 設定変更がパッケージ更新で壊される事故が防げる

unit file のカスタマイズ

- unit file の典型的なカスタマイズはファイル内容の編集ではなく、別ファイルの配置やシンボリックリンクの作成でおこなう。
- *foo*.service という unit file に対して以下のディレクトリが使える
  - *foo*.service**.d** (drop-in ディレクトリ。 *.conf のファイル名で設定を書くと *foo*.service の同じディレクティブを上書きしたものとして解釈される)
  - *foo*.service**.wants**, *foo*.service**.requires** (このディレクトリ内に他の unit file へのシンボリックリンクを置くことで Wants= および Requires= に追記したものとして解釈される)

drop-in の利用例

/lib/systemd/system/httpd.service

```
[Unit]
Description=The Apache HTTP Server
Wants=httpd-init.service
After=network.target remote-fs.target nss-
lookup.target httpd-init.service
Documentation=man:httpd.service(8)

[Service]
Type=notify
Environment=LANG=C

ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
# Send SIGWINCH for graceful stop
KillSignal=SIGWINCH
KillMode=mixed
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

/etc/systemd/system/httpd.service.d/limit.conf

```
[Service]
LimitNOFILE=10240
```

- drop-in には追加または変更したい部分だけ書く
- `systemctl cat httpd.service` のようにすると指定した unit に関係する設定ファイルを出力する (*.wants, *.requires は systemctl cat の出力に含まれない)
- `systemd-delta` コマンドは unit カスタマイズの概要を表示する

関連する man page

- systemd.unit(5): unit file 、 unit file のロードやアンロードの条件、依存関係や全 unit 共通の設定、設定ファイルを配置する path など
- systemd.syntax(7): unit file の文法
- systemd.time(7): systemd 内で使う日時で使える構文
- systemd.directives(7): ディレクティブから man ページへの索引 (既存の unit file を読むときに便利)

やってみよう

- systemctl **list-units** と systemctl **list-unit-files** を比較
- systemctl cat sys-subsystem-net-devices-*eno1*.device のようにして対応する unit file が存在しないことを確認する
- systemctl **cat** default.target と systemctl **show** default.target を比較
- パッケージに含まれる **/lib**/systemd/system/ 以下の unit file と、自動または手動で作られる **/etc**/systemd/systemd/ 以下のリンクやファイルを見比べて systemctl show の出力と比較する
- systemctl enable/disable crond.service で unit を install/uninstall する。この操作による設定が /etc/systemd/system/ 以下に反映されていることを確認する

target unit
同期ポイントを提供

target unit

- target unit 自体は何もしないで状態が変わるだけの unit
  - 前後関係や依存関係をまとめるために利用される
- システム起動、シャットダウン、異常時のレスキューなどで典型的に必要な target はあらかじめ定義されている。自由に追加することもできる。
  - systemd.special(7) を参照

multi-user.target の例

- multi-user.target は主なサービス群を動作させたい従来のランレベル 3 に相当する。
- 定義ファイルを見ると説明と依存関係のみ

/lib/systemd/system/multi-user.target

```
( 略 )
[Unit]
Description=Multi-User System
Documentation=man:systemd.special(7)
Requires=basic.target
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
AllowIsolate=yes
```

multi-user.target の依存関係

`systemctl show multi-user.target` で unit を見ると多数のサービスに Wants と After で依存。 active にすることで多数のサービスが起動する。

```
$ systemctl show multi-user.target
( 略 )
Requires=basic.target
Wants=dbus.service plymouth-quit.service chronyd.service sshd.service tuned.service
rsyslog.service sssd.service pmlogger.service plymouth-quit-wait.service rhsmcertd.service
systemd-ask-password-wall.path pmcd.service systemd-user-sessions.service atd.service systemd-
logind.service firewalld.service NetworkManager.service remote-fs.target getty.target dnf-
makecache.timer crond.service auditd.service systemd-update-utmp-runlevel.service
RequiredBy=graphical.target
Conflicts=shutdown.target rescue.service rescue.target
Before=shutdown.target systemd-update-utmp-runlevel.service graphical.target
After=atd.service sshd.service sssd.service rescue.target chronyd.service tuned.service
NetworkManager.service pmlogger.service systemd-logind.service plymouth-quit-wait.service
plymouth-quit.service rsyslog.service firewalld.service crond.service pmcd.service systemd-user-
sessions.service dnf-makecache.timer dbus.service basic.target rhsmcertd.service getty.target
rescue.service
( 略 )
```

Install 処理での依存関係の追加

- multi-user.target の unit file に存在しない Wants は以下のディレクトリにあるリンク群に由来する
  /lib/systemd/system/multi-user.target.wants/
  - パッケージ内で依存関係を定義している場合に使う

  /etc/systemd/system/multi-user.target.wants/
  - unit file の enable 時に [Install] 節の定義によりリンクを作成する
- /etc/systemd/system/*.wants および *.requires ディレクトリは unit file を拡張する
  - 例 : シンボリックリンク multi-user.target.wants/tuned.service は Wants=tuned.service の追記に相当。
    さらに依存関係解決により、 tuned.service 内の Requires= で参照している dbus.service, polkit.service も multi-user.target の Wants= へ追加される

余談 : デフォルトでの enable/disable

あるサービス (例 : sshd) の rpm や deb などのパッケージを導入したときに、デフォルトでそのサービスを enable にするか、 disable にするか ?

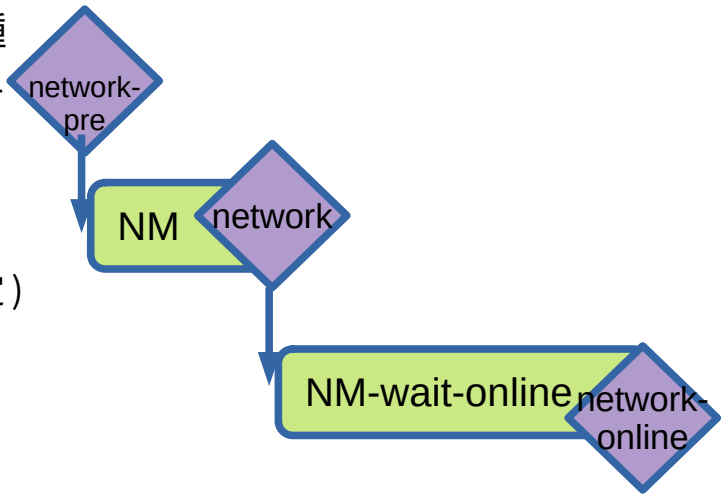
- systemd では、 preset という概念を導入してポリシーを各パッケージから分離した。仕組みは man systemd.preset(5) を参照。
  - “workstation 用途”と“ server 用途”でデフォルト設定を替えたい場合などに便利。
- 各パッケージの導入時に systemctl preset <unit 名 > を実行することで preset にあわせて enable または disable される。

ネットワーク初期化での .target 例

RHEL 8 ではネットワーク初期化に NetworkManager(NM) を使うが、各種サービスは直接 NetworkManager に依存せず network.target や network-online.target へ依存している。

- network-pre.target
  - ネットワーク初期化をはじめる準備ができた (NM.service の After= に指定)
- network.target
  - ネットワーク初期化をはじめた (NM.service の Before= に指定)
- network-online.target
  - ネットワークで外部と通信できるようになった (NM-wait-online.service の Before= に指定)

ネットワーク初期化に NM 以外を利用する場合も、 target は同じなので
サービスは target との依存関係だけを持てばよい。



関連する記事

- Running Services After the Network is up
  https://www.freedesktop.org/wiki/Software/systemd/NetworkTarget/

関連する man page

- systemd.target(5): target unit の説明
- systemd.special(7): systemd がデフォルトで提供する *.target の説明
- bootup(7) ： systemd がデフォルトで提供する *.target の依存関係を表現した図
- systemd.preset(5): systemd の preset 設定の説明

やってみよう

- systemctl list-units -t target -a でどんな target があるか一覧を見る
  - ※LOAD 欄が not-found となっている unit は他から参照されているが unit file は存在しない
    - 例 :syslog.target は systemd 202 よりあとには存在しない
- systemctl show shutdown.target の ConflictedBy= 行を見る。shutdown.target を active にしようとするとここに記載された unit 群は競合するため inactive にされる。
- /etc/systemd/system/multi-user.target.wants/ 以下にあるリンクから中身を見て、 [Install] 節を確認する。

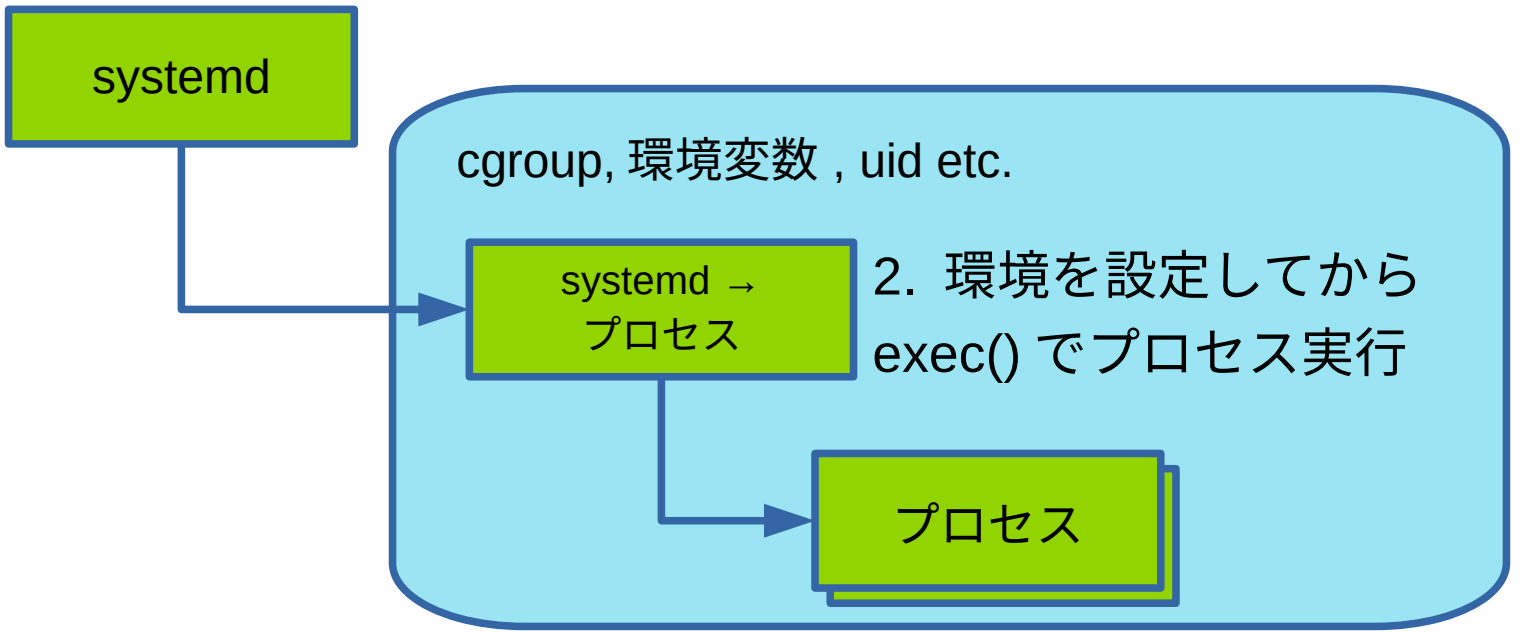
service unit
サービス管理

systemd の service って何するの？

- サービスが動作する環境を用意
- サービスを起動 (通常何らかのプロセスを開始)
- 起動したサービスを監視
  - stop やシャットダウン時などの操作
  - 異常終了の検出、再起動などの対応

サービスのイメージ図

1. systemd は fork() して、子プロセスを cgroup に入れる



3. プロセスがデーモン化や子プロセスの
生成などを行っても cgroup で追跡する

サービスが動作する「環境」って？

- 環境変数
- ulimit
- UID, GID
- nice 値
- numa ポリシー
- CPU アフィニティ
- Capability
- SELinux context

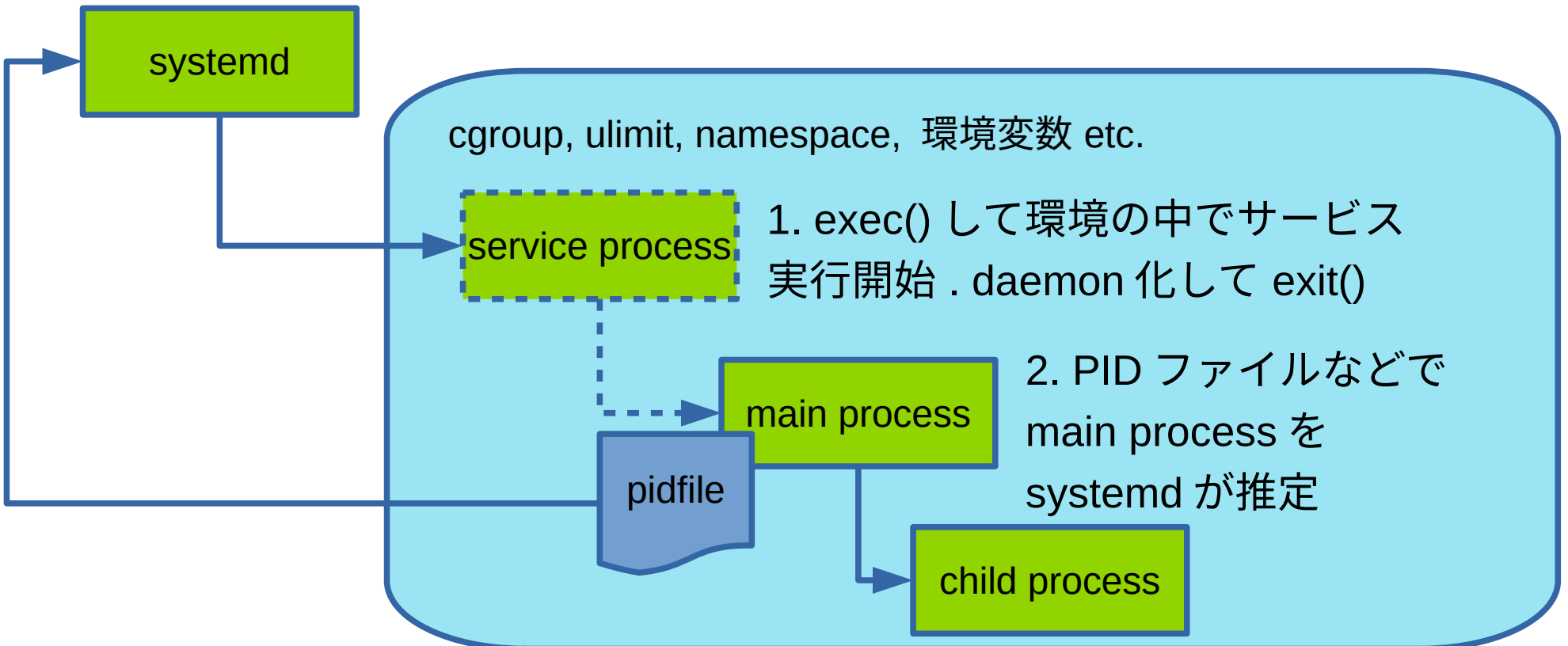
- cgroups でのリソース制御
  - CPU, メモリ , ストレージ I/O, 直接アクセスできるデバイス
- seccomp でのシステムコール制限
- namespace を使ってファイルシステムの一部を bind mount して変更
- サービス毎の firewall 設定

など

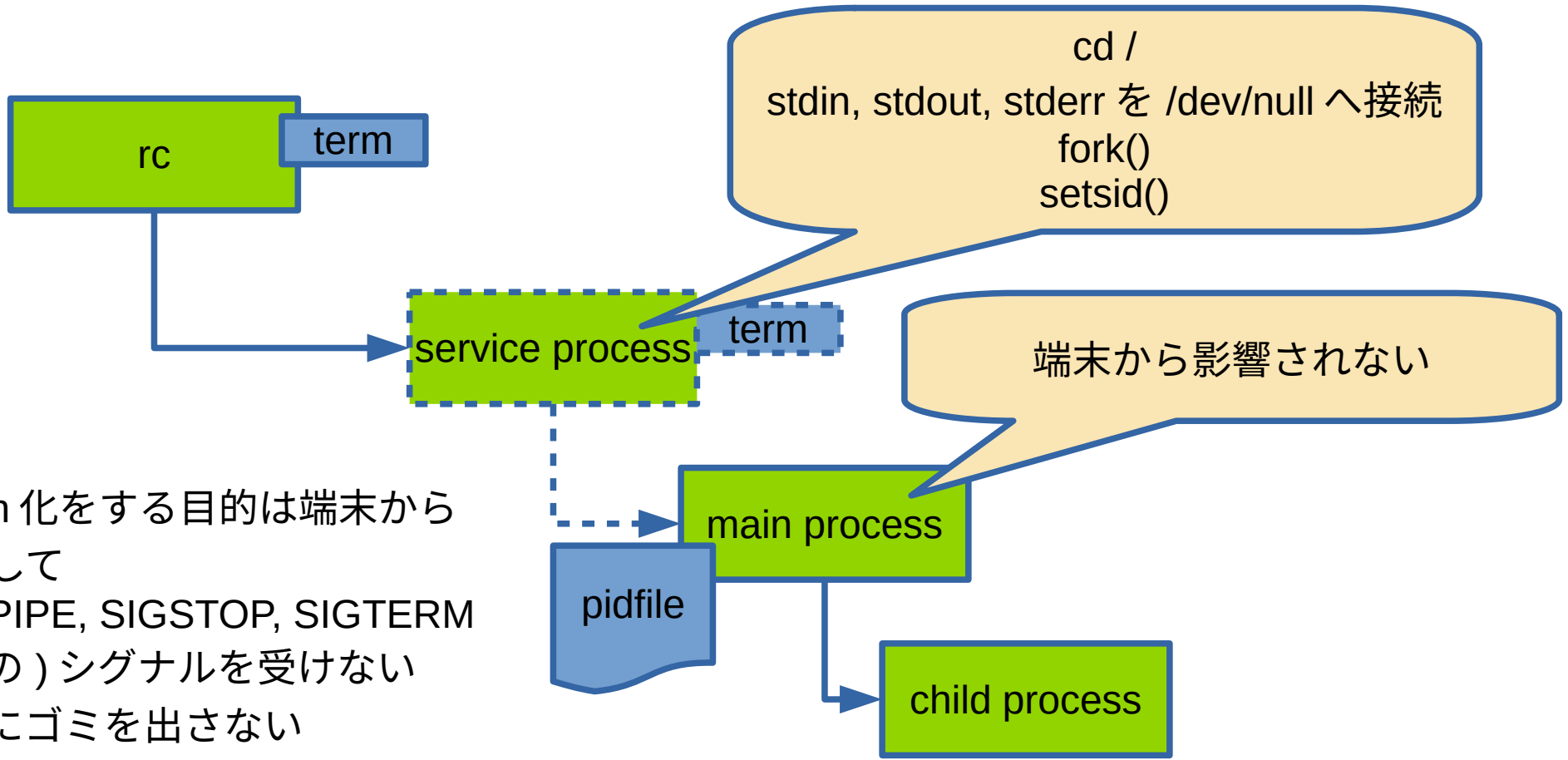
サービスを起動

- 基本的には fork(), exec() でプロセスを実行する
- プログラムによりいろいろな動作があるのでいくつかの type を提供している
  - デーモン化するもの、デーモン化しないもの
  - プロセスが生き続けてサービスを提供しつづけるもの、最初と最後に設定変更をするだけで動作しつづけるプロセスがないもの
  - (systemd がうまく扱える) socket や dbus で待ちうけするもの
  - systemd に対応して通知してくれるもの、 systemd への対応はないもの
- service は type により「どうなったら active とみなすか」「 systemd は何をサービスの停止とみなすか」などの詳細がことなる

daemon 利用時のイメージ



おさらい : daemon 化による端末との切り離し



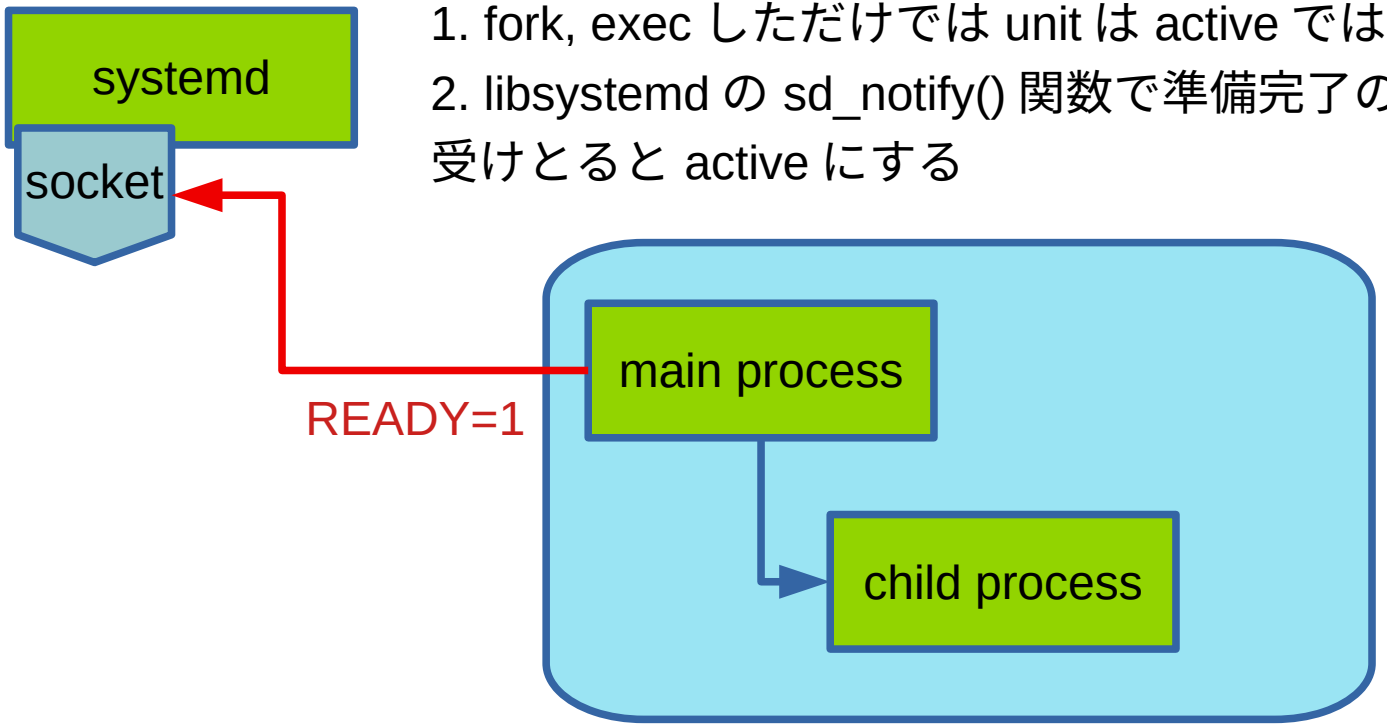
daemon 化をする目的は端末から切り離して

- (SIGPIPE, SIGSTOP, SIGTERM などの) シグナルを受けない
- 画面にゴミを出さない

systemd の環境と daemon 化

- systemd は service に割りあてる stdin, stdout, stderr をどう構成するか明示的に設定できる
  - /dev/null, 端末 , socket, ファイル , journal など
  - systemd.exec(5) の Logging and Standard Input/Output
- 可能 (プログラムに“ foreground mode” などのオプションがある) であれば **Type=simple** を使う。この場合 stdout や stderr へメッセージを出すなら journal に接続して記録できる。
- daemon 化を避けられない場合は **Type=forking** にして PIDFile で監視するべきプロセスを伝える。 stdout や stderr はセットアップされるが、 daemon 化で /dev/null へ接続されなおすのでその後は使われない。

Type=notify 利用時のイメージ

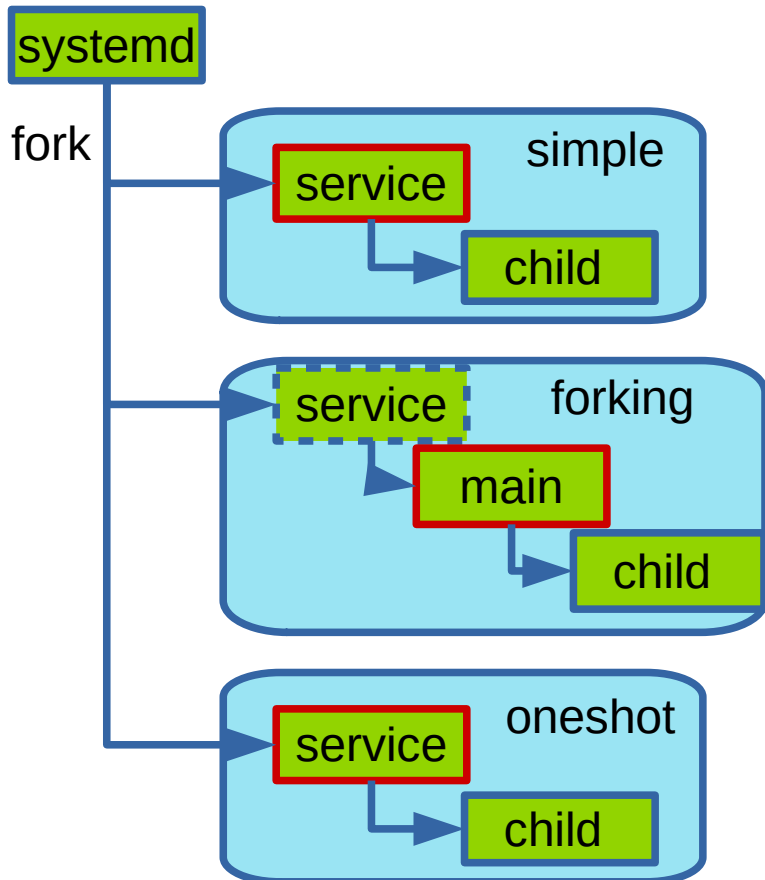


1. fork, exec しただけでは unit は active ではなく activating
2. libsystemd の sd_notify() 関数で準備完了のメッセージを受けとると active にする

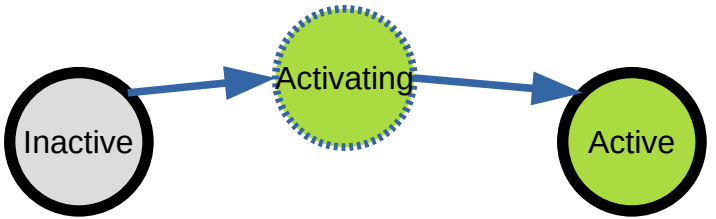
よく使われる type と使い分け

よく使う service の type は以下 3 種類

- simple
  - 実行されたプロセスが直接サービスを提供するタイプのプログラム。
- forking
  - 実行されたプロセスがさらに子プロセスを作りデーモン化するタイプ。古典的な daemon はこの type 。 stdout,stderr 経由の log が使えないため選べるなら simple のほうがよい。
- oneshot
  - 何かの初期化など一時的に実行するだけですぐに終了するもの。



service が active になる条件



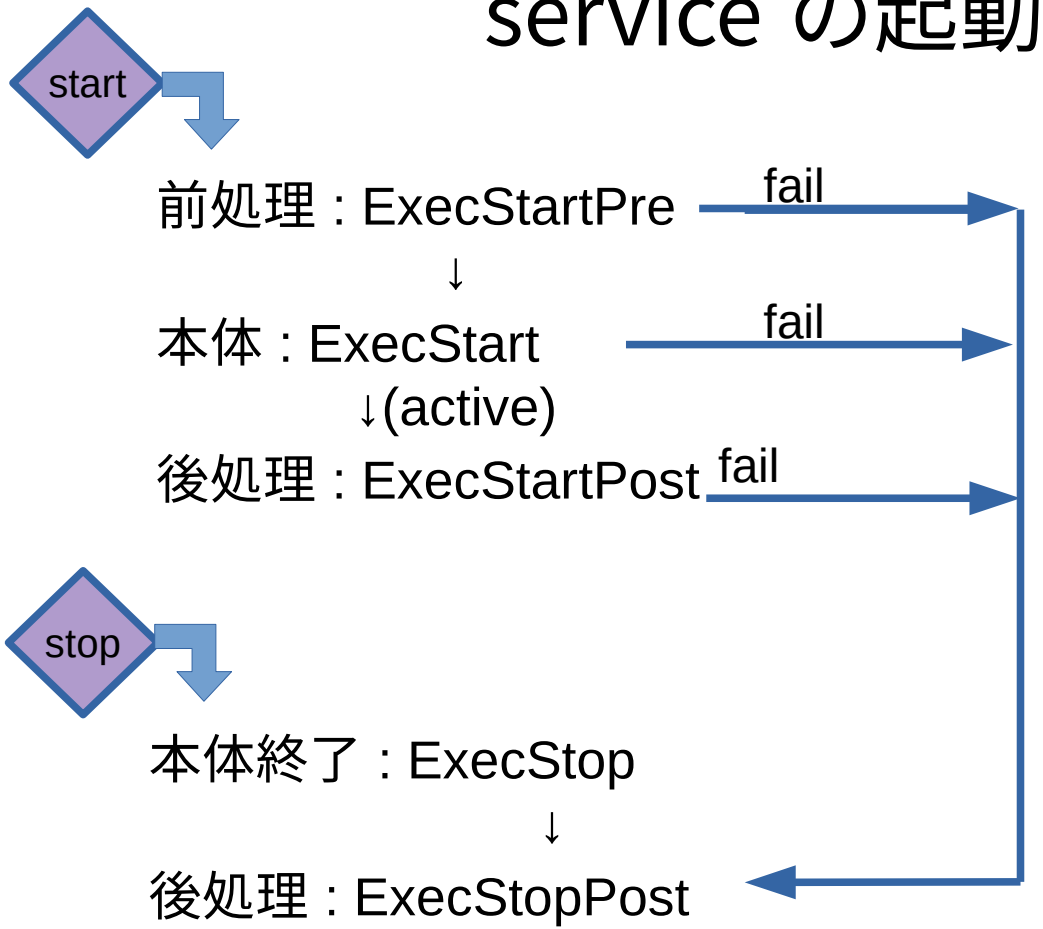
- simple, idle: fork() が成功すると active
- exec: fork(), exec() が成功すると active
- forking: プロセスを起動して main process の PID を確認すると active
- oneshot: fork() が成功して activating 、 exit() すると inactive 。 **active にならない**。
- oneshot(RemainAfterExit つき): fork() したプロセスの exit() が成功だと active
- dbus: dbus 上で指定した名前の待ちうけが行われると active
- notify: sd_notify で通知が行われると active

unit が「 active にならない」とどうなる？

- systemctl start hoge.service を実行すると……
  - hoge が active の場合 → 何もしない
  - hoge が inactive の場合→ active にしようと ExecStart する

  → unit の active/inactive 状態により**二重起動を防止**している
- hoge.service が oneshot で active にならない場合、複数回 systemctl start hoge を実行すると実行した回数だけ ExecStart のコマンドが実行される。

service の起動・終了フロー



- 上から順に実行して、成功の場合は次へ進む
  - service の Type によりいつ active とみなされるかと ExecStartPost の実行開始タイミングが異なる
  - Type=simple と Type=idle では ExecStartPost は ExecStart より先に実行される場合もある
- 開始処理中に失敗した場合、 ExecStopPost を実行する
  - 終了処理で失敗すると unit は failed 状態になる
- 起動・終了それぞれにデフォルトで 1 分 30 秒のタイムアウトが設定されているため、 Exec* はどれも失敗する可能性がある

active な service の終了検出

- systemd は service の終了を検出できる
  - プロセスが exit() した場合、 systemd は PID 1 なので SIGCHLD を受信し、 signal や exit code がわかる。
  - systemd の notify に対応している場合
    - 定期的な watchdog 送信
    - sd_notify でのサービス状態の通知
- ExecStopPost= での終了処理を行う。その後 Restart= により再度 service を active にするよう指定もできる。

service への start 以外の操作 (一部)

- stop
  - service が active であれば ExecStop, ExecStopPost を実行する
  - service が signal などで殺された場合 ExecStopPost を実行する
  - service が inactive であれば何もしない (sysvinit と異なる振舞いなので注意)
- kill
  - cgroup を利用して指定した service に関係するプロセス全てへシグナルを送る

    例) libvirtd が dnsmasq を起動して stop しても dnsmasq はそのまま残る。関連プロセスを全て落とすには kill を使う。
- reload
  - 管理されているサービスに設定ファイルを読み直させる (unit file の読み直しではない)
  - ExecReload= を指定している場合のみ実行可

起動終了処理のカスタマイズ

- 条件確認
  - Condition*= (各種条件を確認し、条件を満たす場合に実行する。ConditionPathExists=, ConditionVirtualization= など)
  - ExecCondition= (何かを実行して実行結果で条件確認)
  - StartLimit*= (start の頻度に制限をかける)
- 実行の成功失敗判断
  - Exec*= に指定する実行ファイル名の直前に”-” をつけると exit code を無視して成功とみなす。
  - SuccessExitStatus= exit code のうち成功とみなすものと、受信シグナルのうち成功とみなすものを追加する

サービスの起動順序が意図と違う？

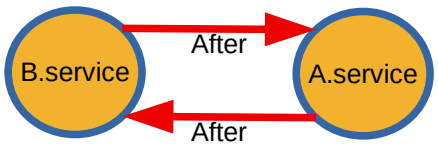
“ 意図通りの実行順にならない” ケースの多くは以下 2 つ

- 依存関係 (Wants, Requires) で指定したものが先に実行されると思った
  - 指定されたものの activation は job queue に入る (だけ) 。
  - これらで指定したものが起動しているか、サービス開始しているかは関係なくサービスを起動しはじめる
  - After, Before での前後関係を定義することで解決できる
- 前後関係 (After, Before) がサービス開始の前後関係であると思った
  - プロセス起動の前後関係は守られる
  - サービス開始したかはわからない (Type=dbus か notify でなければプロセスが起動したことしか見ていないから)

余談：前後関係でループができたら？

設定ミスによりループができるケースがある

このような場合 systemd は (Wants= だけで参照されているような) 必須ではない unit を無視・停止してループを消そうとする。



プロセスの起動順序ではなく
サービスの起動順序を守らせるには？

短時間 (～ 1 分) で初期化が終わる場合は ExecStartPost で対応

- きっちりやる派 (確認プロセスの終了ステータスで伝える):
  Type=forking または simple にして
  ExecStartPost= で「サービス検出を確認したら exit(0) する」プログラムを実行
  - service unit のタイムアウトによる失敗処理があるので確認プロセスは無限に待つ素朴なプログラムでよい
- だいたい動けば OK 派 (ちょっと待つ):
  ExecStartPost=/usr/bin/sleep 30s

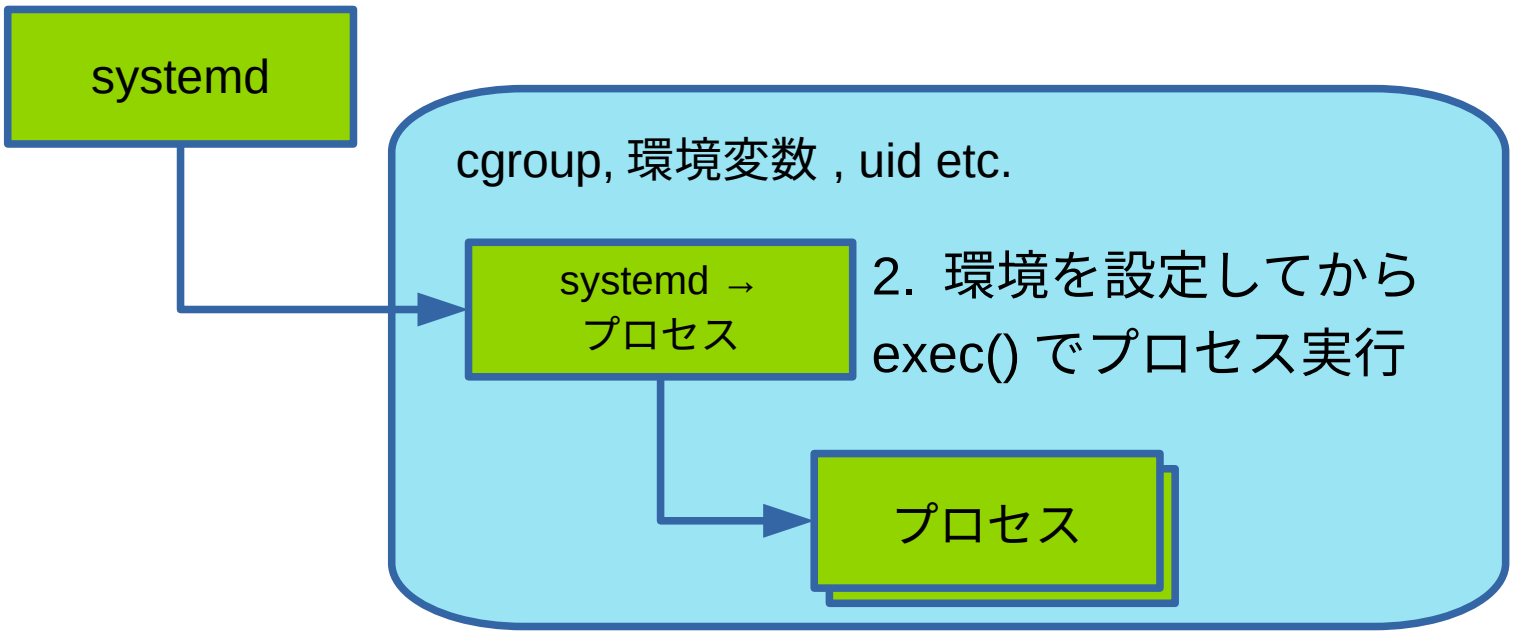
プロセスの起動順序ではなく サービスの起動順序を守らせるには ?(続)

プロセスの起動からサービス開始まで長時間 (1 分〜) かかる場合はプロセス起動のタイムアウト処理とサービス開始の検出を分離する。

- プログラムを systemd に対応させる派 (notify)
  - Type=notify として、サーバプロセスから準備 OK の通知をもらう (起動されるプログラム内での対応が必要)
- サービスの動作確認をするだけの別サービスを用意する
  NetworkManager.service→NetworkManager-wait-online.service
  - Type=oneshot で TimeoutStartSec=infinity, RemainAfterExit=true として ExecStart= でサービスの動作を確認するプロセスを起動する

プロセスの実行環境 (再掲)

1. systemd は fork() して、子プロセスを cgroup に入れる



3. プロセスがデーモン化や子プロセスの
生成などを行っても cgroup で追跡する

systemd-run でサービスの実行環境を調べる

- 例「 PrivateTmp=yes とされている環境ではどうなるか調べたい場合」
- `systemd-run -S -p PrivateTmp=yes`
  - echo $$ で自分の PID を確認する
  - lsns -t mnt で shell が独自の mnt namespace に入っていることを確認する
  - /tmp, /var/tmp に何もないことを確認する
    - /var/tmp は /var/tmp/systemd-<boot ID>-<unit 名 >/tmp を bind mount される。 /tmp は外から見えない

関連する man page

- systemd.service(5): service についての unit file でデフォルトで含まれる依存関係、各 type の動作、 Exec* で使える特殊な記法、 service 特有のディレクティブ、コマンドライン風のパイプおよびリダイレクト処理
- systemd.exec(5): (cgroup 以外の) プログラムの実行環境を準備するためのディレクティブ
- sd_notify(3): systemd へ通信をする C 言語での関数
- systemd-notify(1): sd_notify() を呼ぶコマンド。シェル等で利用する。

やってみよう

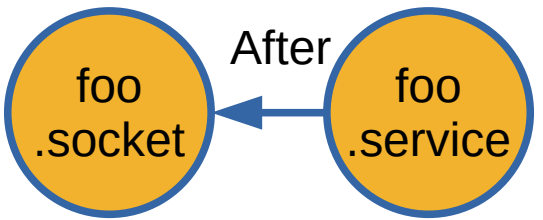
- /lib/systemd/system/ で **simple**, **oneshot**, **forking**, **notify** の例を探す。
  ※ Type= がない service はデフォルトの simple
- Type=oneshot の例をコピー・改変して /bin/echo foobar するだけの unit file foobar.service を作成する
  - /etc/systemd/system に配置、 systemctl daemon-reload で読み込み
  - systemctl start foobar.service してログに foobar が記録されていることを確認する
  - systemctl status foobar.service して状態を確認
  - RemainAfterExit= の設定を反対にして同じことを繰り返して動作の違いをみる
- 複雑な service unit の例として nfs-server.service か libvirtd.service を見てみる

socket, path, timer unit
イベントによる service 起床

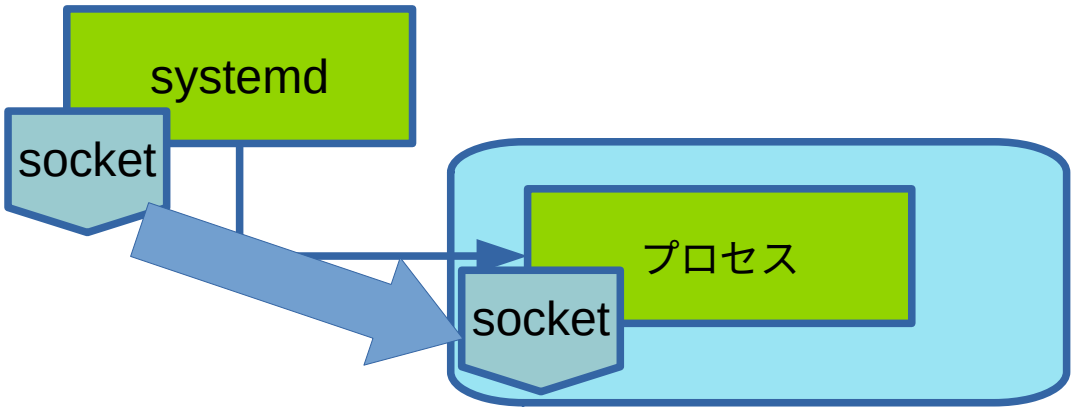
何かのイベントを契機に service を起動する

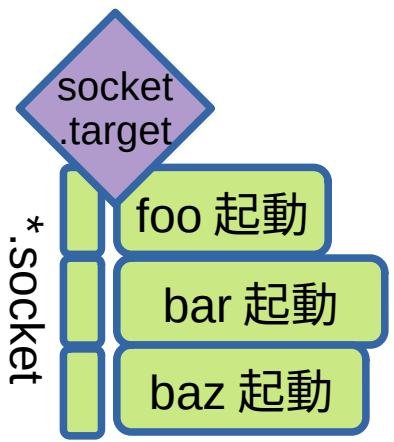
systemd では「イベントを契機にして service を起動する」仕組みを提供し、 unit の組み合わせとして管理する

- socket, path, timer
- イベントを検出するとデフォルトでは同じ名前の .service を active にする
  - Service= で任意の service unit を指定できる



socket unit





- systemd が socket を listen する。
  - 起動のごく初期に listen を開始するので一般のプロセスによるランダムなポート open との競合を回避する
  - socket の listen と service の立ち上げを分離すると、 socket だけ先に作っておけば互いに socket で通信する service 群の起動を並列化できる。
  - socket の利用を開始すれば自動的に相手を待つ。
- 対応する service unit の起動時に socket をひきつぐ。
  - foobar.socket には foobar.service が対応
  - 別名の service への対応づけも指定可

「socket の引き継ぎ」ってどうやるの？

systemd が socket unit で待ちうけていた socket を service unit へ渡す方法は……？

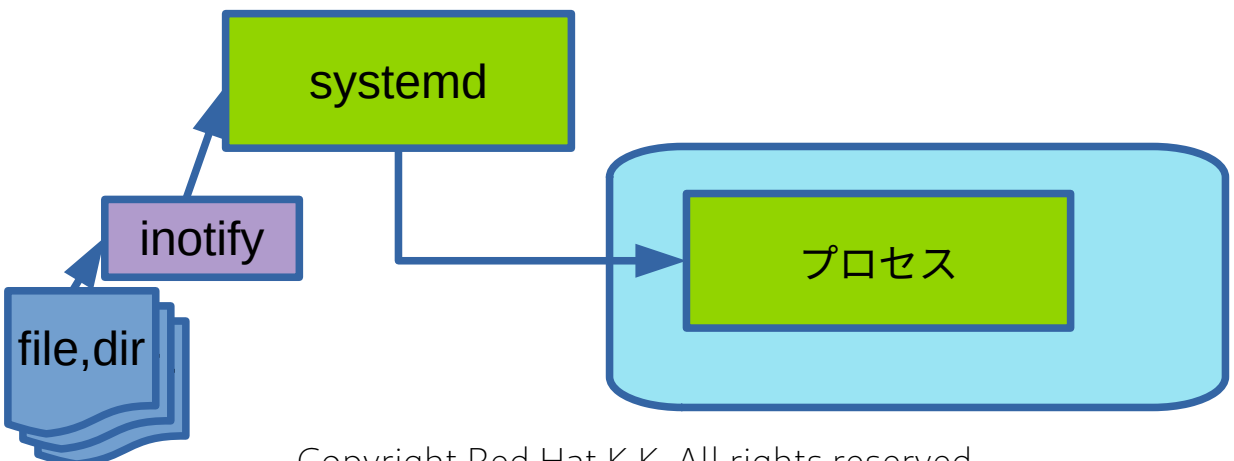
- inetd 方式 : service unit で StandardInput=socket として標準入出力を socket とのやりとりに使う
- libsystemd 方式 : systemd が service 実行時に fd 群を設定する。サービスを実行するプロセスは sd_listen_fds() 関数で fd の数や属性を取得する。

socket activation

- 起動直後の sockets.target への依存が暗黙に定義されるので有効な socket unit は起動直後に active になり、 socket や port を確保する
- systemd が socket への着信を検出すると対応する service を起動する
  - 1 回目の通信には遅延が発生する
- service の [Install] 節に通常の service と同様 WantedBy= 等を挿入するよう定義することで、ユーザーが unit の動作を選べるようにもできる
  - enable してシステム起動時の起動対象にする
  - disable して socket activation の対象にする
- service で Also= として socket をインストールすることで service の enable/disable とセットでポートやソケットの確保を指定できる

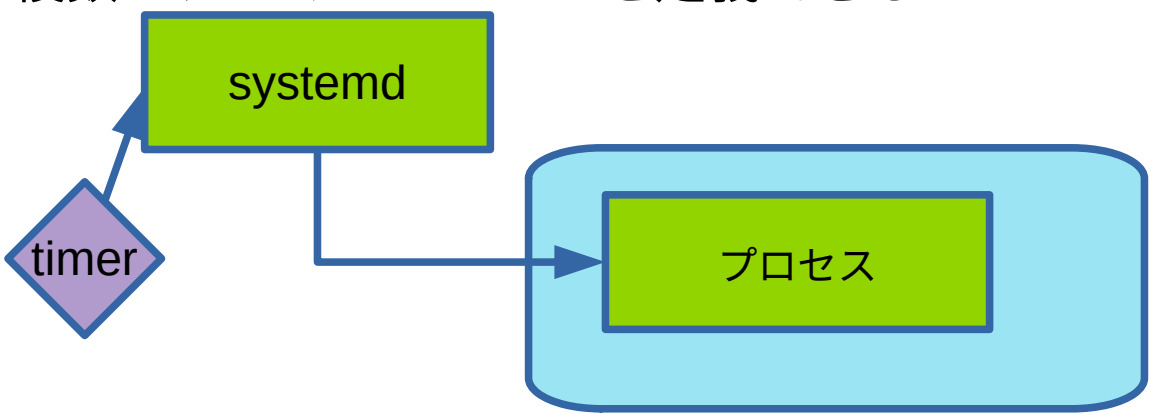
path unit

- systemd が path の存在・変更等を inotify() で監視する
- socket activation と同様に指定されたイベントがあれば対応する service を起動する
- service は特にイベントを受信したり共有したりはしない



timer unit

- timer はいくつかのイベントから xx マイクロ秒経過のような表現で定義
- socket activation と同様に指定されたイベントがあれば対応する service を起動する
- service は特にイベントを受信したり共有したりはしない
- timer unit は複数のタイマーイベントを定義できる



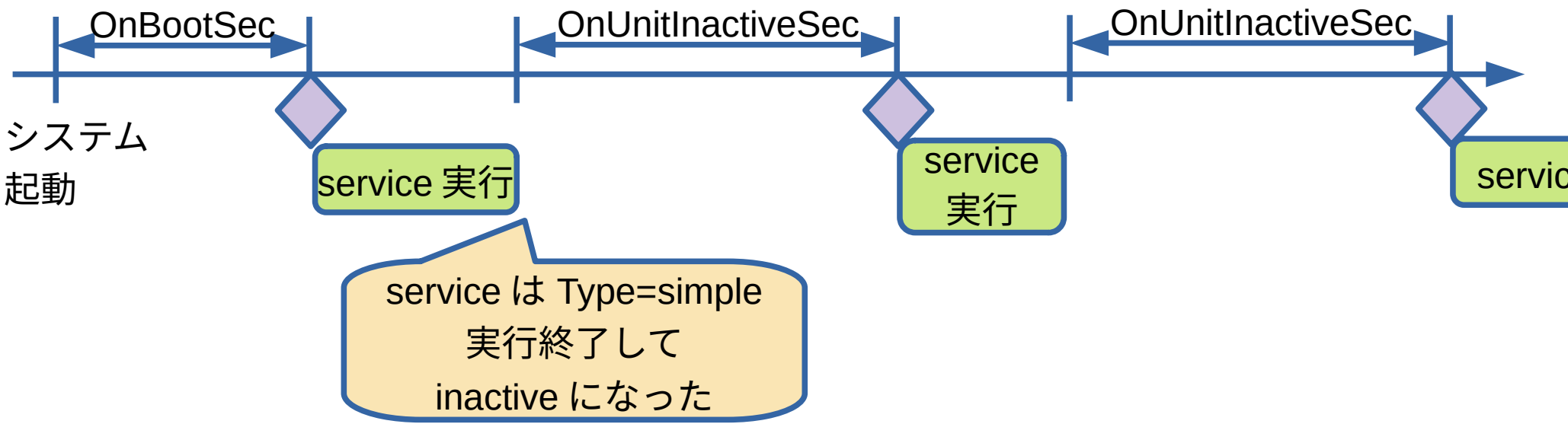
余談 : timer の扱うイベント

timer はさまざまなイベントからの相対時間を指定できる

- OnBootSec= システムが起動してからの相対時間
- OnStartupSec= systemd が起動してからの時間。システム全体を管理する systemd では OnBootSec とほぼ同じ。ユーザセッションではログイン直後のタイミングからの相対時間になる。
- OnActiveSec= timer unit 自身が active にされてからの相対時間
- OnUnitActiveSec= timer に対応する service unit が active になってからの時間
- OnUnitInactiveSec= timer に対応する service unit が inactive になってからの時間
- OnCalendar= カレンダー上の時刻。 cron のような指定ができる

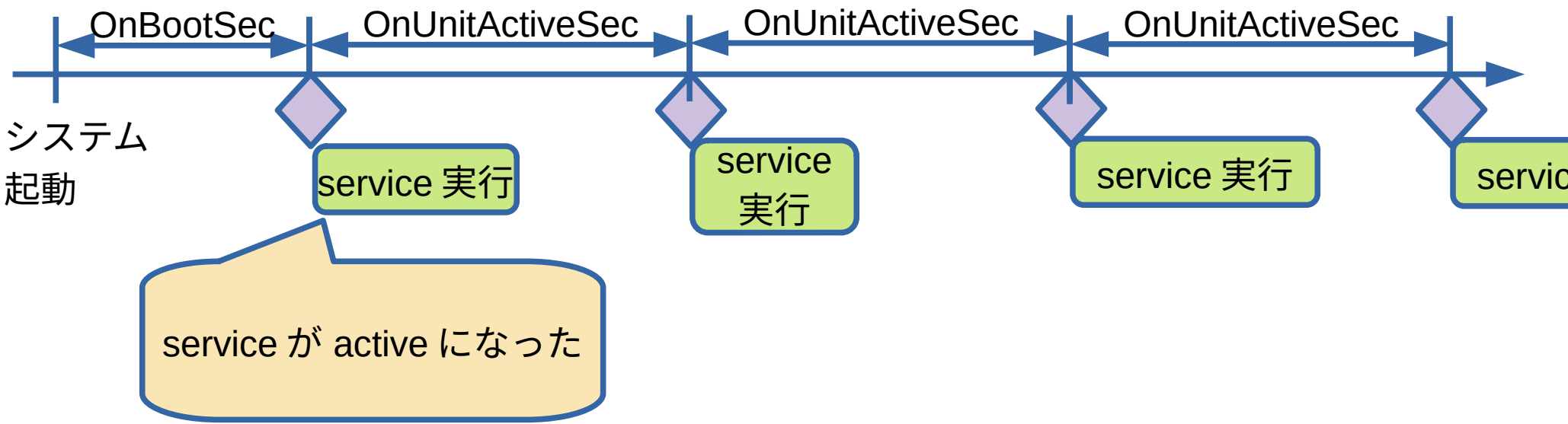
timer の繰り返し (1)

繰り返し実行させたい場合には、 OnBootSec= と OnUnitInactiveSec= を組みあわせて実現



timer の繰り返し (2)

繰り返し実行させたい場合には、 OnBootSec= と OnUnitActiveSec= を組みあわせて実現。実行時間が OnUnitActiveSec より長い場合は終了後すぐ実行する。



timer をわざとずらす

- AcculacySec= timer で指定された時刻をどれだけの精度で実現するか。デフォルトでは 1 分。すこしずつ異なるタイマーイベント群にもとづくアクションの実施タイミングを揃えることで消費電力削減に役立つ。
- RandomizedDelaySec= timer で指定された時刻をランダムに最大どれだけ遅延させるか。デフォルトは 0 。クラスタ環境で同じプログラムが同時に実行されることにより、負荷が過度に集中することを避けるために利用する。

path, socket, service 連携例 cupsd (1)

cups.service

```
[Unit]
Description=CUPS Scheduler
After=sssd.service network.target ypbind.service

[Service]
ExecStart=/usr/sbin/cupsd -l
Type=notify
Restart=on-failure

[Install]
Also=cups.socket cups.path
WantedBy=printer.target
```

- ExecStart= 起動時に実行するコマンド。 ExecStop 等の指定がないので終了時はここで作成されたプロセスへ「 SIGTERM 送信→タイムアウト待ち→ SIGKILL 送信」による終了処理を行う
- Also= この unit を有効にする時に指定した unit も有効化する
- WantedBy= enable すると依存関係を挿入する。 printer.terget を有効化すると cups.service も有効化の対象になる

path, socket, service 連携例 cupsd (2) Socket Activation

cups.socket

```
[Unit]
Description=CUPS Scheduler
PartOf=cups.service

[Socket]
ListenStream=/var/run/cups/cups.sock

[Install]
WantedBy=sockets.target
```

- ListenStream= 指定したソケットを systemd が待ちうける
- systemd がソケットの listen を行い、はじめて接続がおこなわれた時点で対応するサービスを起動する (Socket Activation)
  - この例では cups.socket を待ちうけて着信があれば cups.service を起動

path, socket, service 連携例 cupsd (3)
Path based Activation

cups.path

```
[Unit]
Description=CUPS Scheduler
PartOf=cups.service
[Path]
PathExists=/var/cache/cups/org.cups.cupsd
[Install]
WantedBy=multi-user.target
```

- ファイルやディレクトリの存在や変更を検知して対応するサービスを起動する (Path based activation)
- この例では /var/cache/cups/org.cups.cupsd が存在すれば cups.service を起動する

timer と service の連携例 systemd-tmpfiles-clean.timer

systemd-tmpfiles-clean.timer

```
[Unit]
Description=Daily Cleanup of Temporary Directories
Documentation=man:tmpfiles.d(5) man:systemd-tmpfiles(8)

[Timer]
OnBootSec=15min
OnUnitActiveSec=1d
```

- 時間の経過を契機として service を起動する
- この例ではシステム起動後 15 分、その後 1 日おきに systemd-tmpfiles-clean.service を起動する

関連する man page

- systemd.socket(5), systemd.path(5), systemd.timer(5): 各種類の unit の説明、利用できるディレクティブなど
- systemd.time(7): 時刻、時間の表記法
- systemd for Developers I
  - http://0pointer.de/blog/projects/socket-activation.html

やってみよう

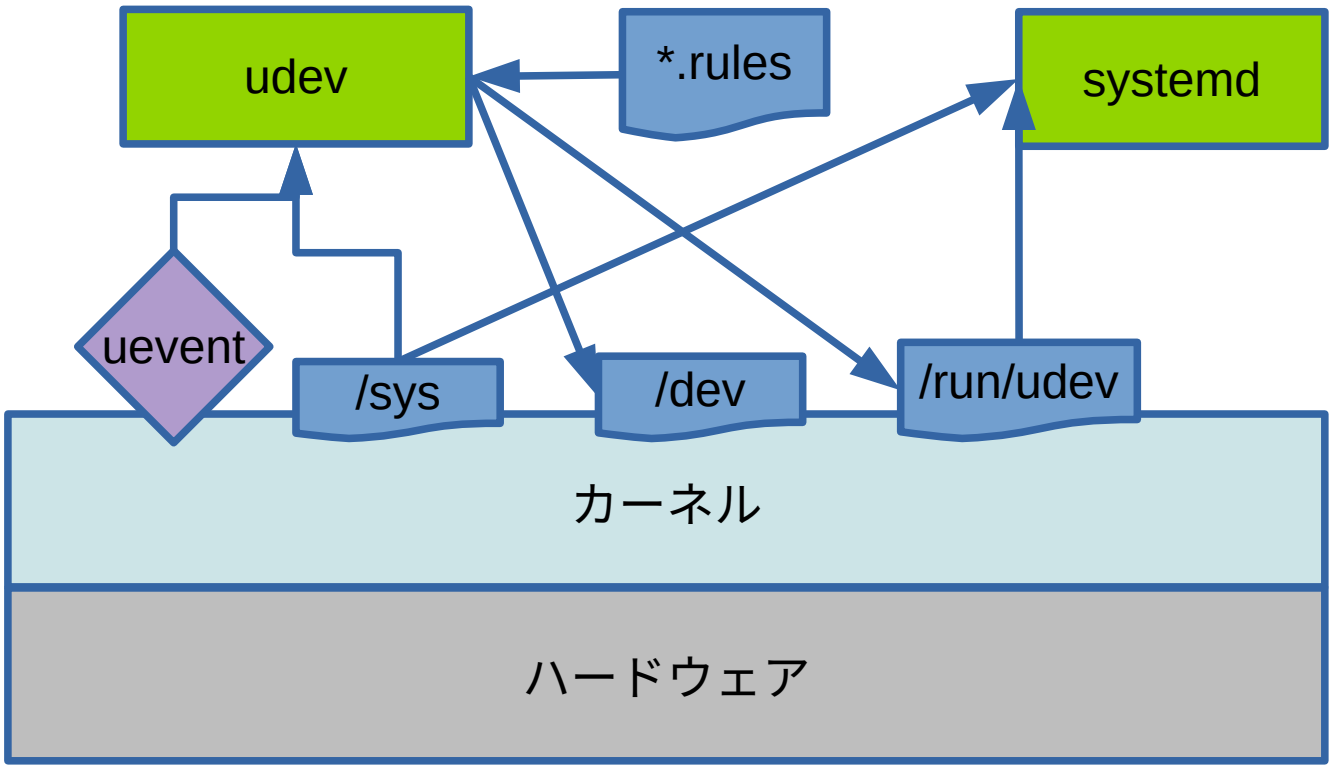
- lsof -p 1 で systemd が多くの socket を待ちうけていることを確認
  - systemctl -t socket で socket unit を確認して照らしあわせる
- systemctl list-timers で今予定されている timer の起床一覧を見る
- fstrim.timer は 1 週間に 1 回 service を起動する設定になっている。 drop-in を使って 1 日に 1 回に書き換えてみる。
- /lib/systemd/system/*.timer を見ていろいろなタイマーの使い方のパターンを見る

device unit と mount unit
ファイルシステムの mount

デバイス検出から mount まで

- 主な登場人物 : linux kernel, udev daemon, systemd, fstab
- 関係する unit
  - .device
  - .mount
  - local-fs.target, remote-fs.target

デバイス検出から .device unit 作成まで



1) ハードウェアの追加や変更を kernel が検出

2) uevent を udev が受信

3) udev の *.rules により /dev 以下にファイルを作成したり、 systemd 用の情報を付加したりする

4) systemd は udev により”systemd” と TAG が付与されたデバイスの .device unit を作成する

fstab から mount unit を生成

- systemd-fstab-generator
  - /etc/fstab を解析して、対応する mount unit 定義を生成する

例 : fstab の ”/dev/mapper/snake-home /home xfs   defaults   0 0” という行から生成された /run/systemd/generator/home.mount

```
# Automatically generated by systemd-fstab-generator

[Unit]
SourcePath=/etc/fstab
Documentation=man:fstab(5) man:systemd-fstab-generator(8)
Before=local-fs.target

[Mount]
Where=/home
What=/dev/mapper/snake-home
Type=xfs
```

余談 : systemd-fstab-generator 実行タイミング

- systemd の起動直後と、 systemd daemon-reload 実行時に systemd-fstab-generator は実行される。つまり……
  - initramfs 内での起動直後、 initramfs 内の fstab から unit 生成
  - root directory を変更して systemd を読み直した直後、 /etc/fstab から unit 生成
- fstab を編集して手動で mount を行う操作はひきつづき有効なので利用していても特に今までと違いはない

余談 : systemd-mount, systemd-umount

- systemd の mount unit はちょっと賢い
  - 親ディレクトリの mount や、ネットワーク接続を待つ
  - automount
  - その他任意の unit の依存関係が書けるので mount が成功するのを待ってサービスを起動する等
- 賢いのはいいが使うのに都度 fstab に書いて generator を実行して systemctl start foobar.mount するのは面倒なので……
  → 専用コマンド systemd-mount が用意されている

local-fs.target, remote-fs.target

- local-fs.target: ネットワーク不要で mount できるファイルシステム
  - fstab に書かれている local fs はこの target から依存される
- remote-fs.target: ネットワーク初期化後に mount できるファイルシステム
  - fstab で _netdev オプションが書かれているとこの target から依存される

関連する man page

- systemd.mount(5): fstab に書くオプションについてここに記載されている
- systemd.device(5): udev rules で定義する変数について記載がある
- udev(7), udevadm(8): udev database を見たりイベントをモニタしたいときは udevadm コマンドで見る

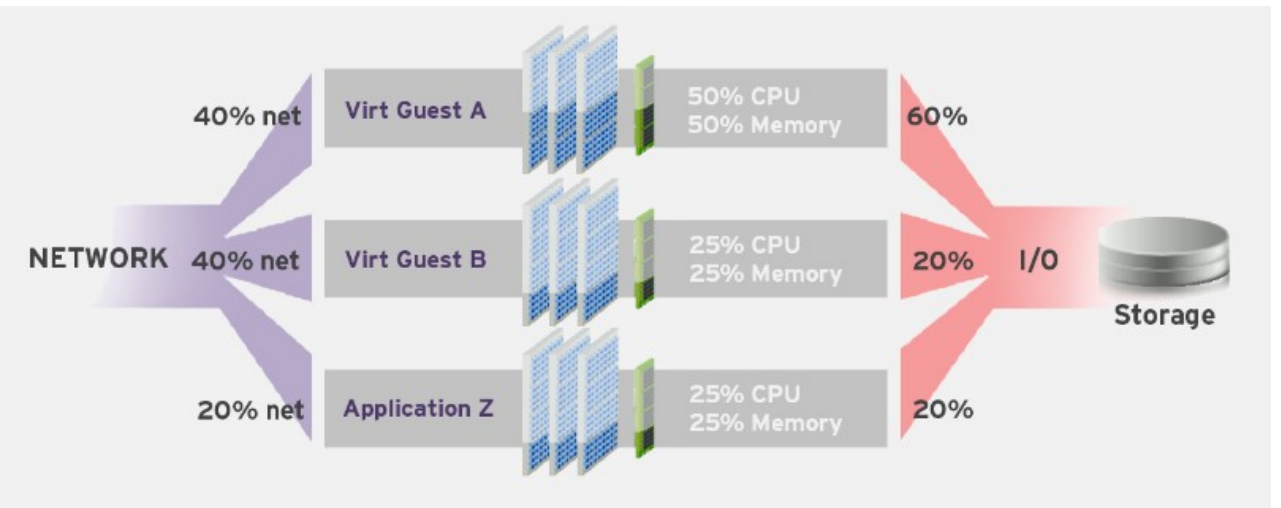
やってみよう

- root fs のデバイスを ROOT として。
  - udevadm info ROOT の出力をみる
    - 対応しそうな device unit を systemctl show して照らしあわせる (DEVPATH と Description 等)
    - TAGS= に systemd があることを確認する
- /lib/udev/rules.d/99-systemd.rules を眺める
  - printer サブシステムのデバイスがあると SYSTEMD_WANTS 経由で printer.target を要求
- USB メモリを挿入したり loopback device を作成して対応する device unit ができていることを確認する

slice, service, scope unit
cgroup 管理

cgroup って何？

- linux が持つプロセスをグループ化する仕組み control group の略
  - 主にリソース管理を行うために利用する
  - 全てのプロセスはいずれかのグループに所属し、子プロセスは明示的な移動がなければ親プロセスと同じグループに所属しつづける



cgroup の利用シーン例

- 特定サービス , VM, コンテナだけで特定 CPU を占有、他のサービスやユーザセッションでは利用できる CPU を制限
- 主にサービスを実施するマシンで
  - ユーザの対話セッションで利用できるリソースを制限
  - Ansible や Puppet が使うリソースを制限
- サービスに最低限必要なメモリ量を設定 (cgroup v2 が必要)
- コンテナ毎に利用できる IOPS を設定 (cgroup v2 が必要)

cgroup によるリソース管理の特徴

- cgroup はツリー状にグループを分類して、グループに対してリソースを割り当てる
- 各グループ毎にリソース制御の設定が可能
  - CPU 使用率の比
  - IO スループット
  - メモリ +Swap の割り当て量の上限、下限
  - アクセスできるデバイス
  - ネットワークのクラス付け など
- cgroup v1 と、 cgroup v2 が存在する。 2020 年時点では cgroup v2 への過渡期。
  - systemd は cgroup v1 および v2 の両方に対応しているが、 cgroup のバージョンにより使えるディレクティブが異なる。

systemd と cgroup の関係

- 各 unit には対応する cgroup がありリソース管理以外でも活用される
  - systemd は各 unit を初期化する時にグループの作成・設定をおこなう
  - リソース管理を行わない場合にも、各 unit から実行されるプログラムは対応する cgroup 内で実行される。そのためあるプロセスがどの unit に所属しているかが linux kernel により自動的に追跡される。
- systemd は cgroup を管理してリソース管理をおこなう
  - cgroup のパスは slice, scope, service, socket, mount, swap のいずれかの unit に対応づけられ、各 unit の設定としてリソース制御の指定をおこなう
  - systemd が直接管理する他に、部分的に管理を移譲する API がある

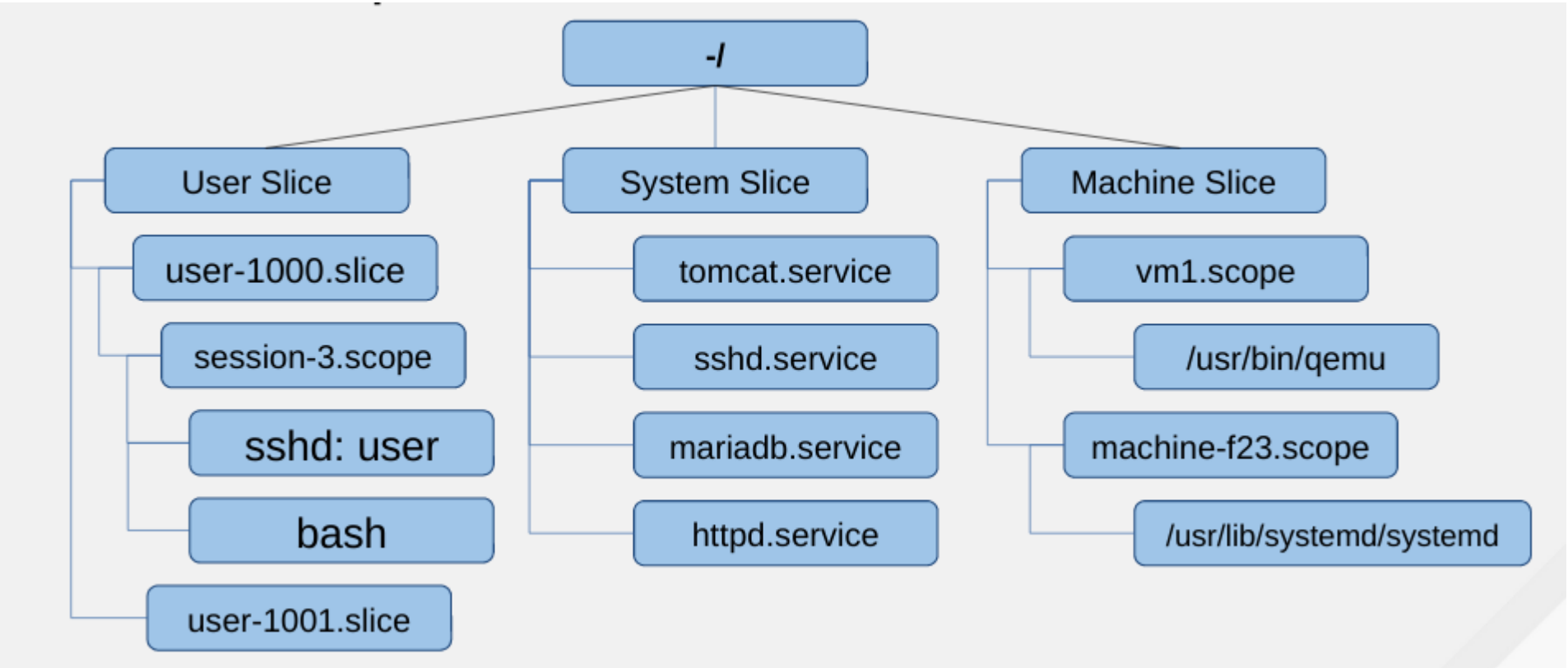
cgroup と関連してよく使う 3 種類の unit

- **slice**: cgroup のグループ分け用の unit
  - slice を単体で宣言する他 service の Slice= で指定して作成
- **service**: systemd が管理するプロセスおよびその子プロセス群
  - 各 service は自動的に対応する名前の cgroup の中に入れられる
- **scope**: systemd 以外のプロセスが fork&exec& 管理をおこなうものを API で登録する
  - systemd-run, gdm, gnome-launchd などは作成したプロセスを scope で作成した cgroup の中に入れる

デフォルトの 3 種類の slice

- User slice
  - ユーザセッションに対応するプロセスや service を格納する
  - 各ユーザ用の systemd (後述) を起動
  - 各ユーザの service を起動
- System slice
  - システムサービスを格納する
- Machine slice
  - 仮想マシン、コンテナ等を格納する

user slice, system slice, machine slice



前述の 3 つの slice の下にユーザ毎、サービス毎、 VM 毎などツリー状に分類される

systemd-cgls

cgroup の階層構造と所属プロセスを一覧表示する

```
-.slice
├─user.slice
│ └─user-11956.slice
│   ├─session-3.scope
│   │ ├─1909 gdm-session-worker [pam/gdm-password]
│   │ ├─4727 /usr/bin/gnome-keyring-daemon --daemonize --login
│   │ ├─7654 /usr/libexec/gdm-x-session --run-script /usr/bin/gnome-session
│   │ ├─7663 /usr/libexec/Xorg vt2 -displayfd 3 -auth /run/user/11956/gdm/Xauth>
│   │ ├─8020 /usr/libexec/gnome-session-binary
│   │ └─8225 /usr/bin/ssh-agent /bin/sh -c exec -l /bin/bash -c "/usr/bin/gnome>
│   └─user@11956.service
│     ├─gsd-xsettings.service
│     │ └─8764 /usr/libexec/gsd-xsettings
│     ├─gvfs-goa-volume-monitor.service
│     │ └─8657 /usr/libexec/gvfs-goa-volume-monitor
│     ├─gsd-power.service
│     │ └─8723 /usr/libexec/gsd-power
│     ├─xdg-permission-store.service
```

systemd-cgtop

cgroup 毎に CPU, メモリ ,I/O 使用量を表示する。ここで表示するには accounting(cgroup での資源利用追跡) が有効になっている必要がある。

```
Control Group                              Tasks   %CPU   Memory  Input/s Output/s
/                                            213      -    2.9G        -        -
/system.slice                                 57      -    2.7G        -        -
/system.slice/NetworkManager.service           3      -    7.5M        -        -
/system.slice/atd.service                      1      -  644.0K        -        -
/system.slice/auditd.service                   4      -    9.2M        -        -
/system.slice/boot.mount                       -      -   92.0K        -        -
/system.slice/chronyd.service                  1      -    3.1M        -        -
/system.slice/cockpit.socket                   -      -    2.2M        -        -
/system.slice/crond.service                    1      -    2.2G        -        -
/system.slice/dbus.service                     2      -   28.8M        -        -
/system.slice/dev-hugepages.mount              -      -  632.0K        -        -
/system.slic…-mapper-rhel\x2dswap.swap         -      -  456.0K        -        -
/system.slice/dev-mqueue.mount                 -      -   52.0K        -        -
```

ユーザのリソース制限

- 各ユーザは user-<PID>.slice により定義されるので、この unit に対して定義をおこなうことでユーザのリソースを制限できる
  - /etc/systemd/system/user-<*UID*>.slice.d/resources.conf
  - 全ユーザに対して同じ設定を指定したい場合は、 user-.slice がテンプレートなのでそこへ定義を記述することで全ユーザへ影響する
- ログイン時に pam_systemd により cgroup の作成・切り替えが行われる
  - su や sudo では cgroup は切り変わらないのでリソース管理上は同じユーザのまま
  - cgroup を変えたい場合は systemd-run を使うか ssh などでログインしなおす

設定例 : /etc/systemd/system/user-1000.slice.d/resources.conf

```
[Slice]
CPUQuota=120%
```

関連する man page

- systemd.resource-control (5): unit に対する cgroup でのリソース制御。 cgroup v1 、 v2 の違いは Unified and legacy control group hierarchies 節にまとまっている。
- RHEL7 ドキュメント「リソース管理ガイド」
  https://access.redhat.com/documentation/ja-jp/red_hat_enterprise_linux/7/html-single/resource_management_guide/index
- linux kernel ドキュメント「 Control Group v2 」
  https://www.kernel.org/doc/Documentation/cgroup-v2.txt

やってみよう

- systemd-run -p CPUQuota=10% -t yes > /dev/null として (--user では制限されないので注意)
  - top で CPU 消費量を見てリソース制限されていることを確認する
  - /proc/<PID>/cgroup を見て cgroup 名を確認
  - /system.slice/run-u492.service のような名前なので、 /sys/fs/cgroup/cpu,cpuacct/ 以下から対応するディレクトリをみつけ、 cpu.cfs_period_us, cpu.cfs_quota_us を比較する
- /etc/systemd/system/user-*<UID>*.slice.d/resources.conf に以下を記述する

```
[Slice]
CPUQuota=123%
```

- ログインしなおしてから systemd-cgtop で消費の様子をみる
- su, sudo 前後で /proc/self/cgroup を見て追跡されている様子を確認する

ユーザセッション用 systemd

ユーザセッションそのものの管理

- systemd-logind がユーザセッションを追跡する。 ConsoleKit の後継
  - どのユーザが、いつから、どの seat(通常ひと組のディスプレイ、キーボード、 USB ポート等) からアクセスしているか、関係するプロセスはどれかの追跡
  - 周辺装置の権限管理
  - idle 検出、サスペンド・レジューム、電源断
- loginctl コマンドで操作する
  - セッションのロック、強制終了、有効化、関連プロセス kill など

systemd の仕組みをユーザ用に使う

各ユーザで systemd を利用できると便利

- ユーザごとの service 管理
  - tmux 、 emacs server 、デスクトップ環境での各種サービス等
  - ハードウェア利用 (スマートカード、 Bluetooth アダプタ等) のためのサービス起動
  - システムのサービスと同じ仕組みで管理
  - 今まで混在していたり失われていたログを journald へ蓄積

ユーザ用 systemd

- 各ユーザの systemd インスタンス (service manager) は systemd --user として起動される。
  - /lib/systemd/user , ~/.local/share/systemd/user, /etc/systemd/user, ~/.config/systemd/user から unit file をロードする
  - 起動時は default.target を active にするよう動作する
- systemd のインスタンスと配下の service 群は「セッション毎」ではなく「ユーザ毎」に対応する。そのためログアウトのタイミングで終了するとは限らない。

ユーザ用 systemd からのリソース管理

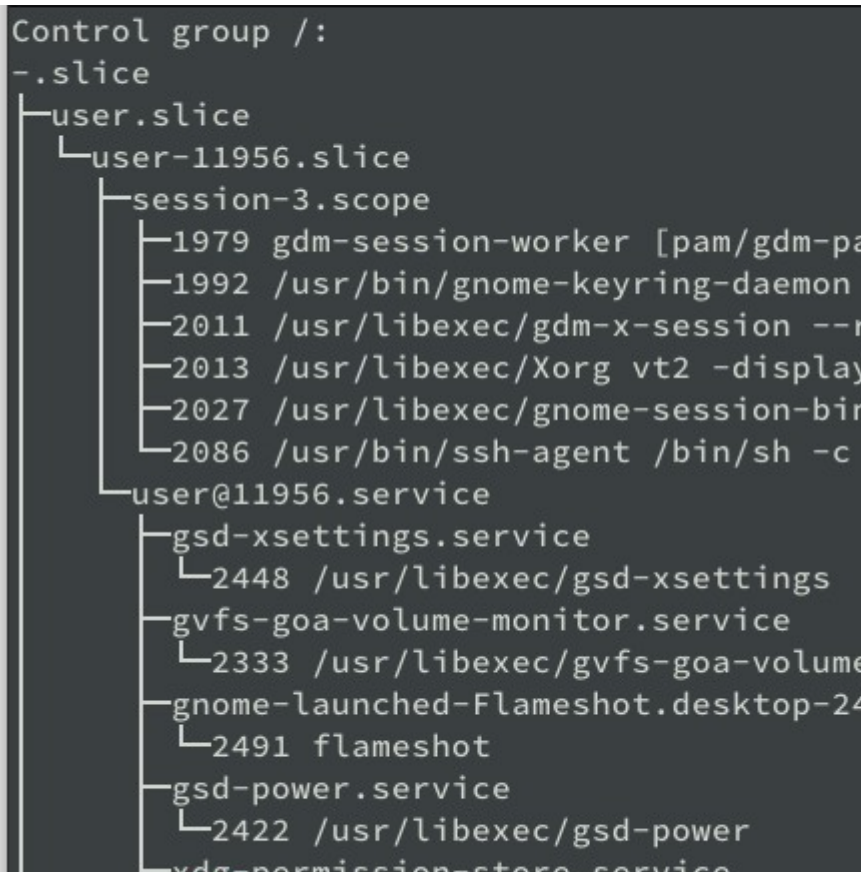
- デフォルトではユーザ用 systemd からの cgroup によるリソース管理は memory, pid しか利用できない。他のリソースを管理するにはシステムの systemd から管理を移譲する設定が必要。
  - 例 : /etc/systemd/system/user@.service.d/override.conf

```
[Service]
Delegate=yes
```

  - cat /sys/fs/cgroup/user.slice/user-11956.slice/cgroup.controllers のようにして cgroup に設定が反映されているかを確認

ログイン時の動作

- ログイン認証時にから pam_systemd を動作させる。
- pam_systemd はログインにあたりいくつかの作業をおこなう
  - ユーザ用の /run/user/<UID>/ を作成
  - セッション毎の scope を作成
  - 存在していなければユーザ毎の systemd インスタンスを起動する



ユーザプロセスの自動 kill

ユーザ用 systemd で管理される service および scope 内のプロセスは以下の条件で kill される

- セッション終了した場合の動作は logind.conf 内の KillUserProcess= 設定による :
  - **No(デフォルト)**: 特に何もしない
  - **Yes**: ユーザがログアウトした時にそのセッションに対応する scope 以下のプロセスを kill する。さらに特定ユーザのみ kill 対象、特定ユーザのみ kill 対象外の設定も可。
    - kiosk 端末などで便利だが tmux, nohup 等のログアウト後に実行を継続されてほしいソフトウェアにとっては想定と異なる動作になる
- ユーザ用 systemd が SIGTERM, SIGINT をうけると shutdown.target が有効化され (Conflict しているサービスが終了す) る。

loginctl enable-linger とは？

- 英語の linger は「居残る、長引く、なかなか消えない」というような意味
- ユーザは自分の全セッションが終了しても scope, service を自動で kill しないよう指定することができ、`loginctl enable-linger` がその指示を行うコマンド。
- 運用管理者は PolicyKit の設定で linger の指定を禁止することもできる。そのため実用上は以下の 3 種類から動作を選べる
  - enable-linger を禁止してセッション終了時に全プロセスを kill する (kiosk 端末など)
  - enable-linger したユーザのプロセスは kill しない、していないユーザのプロセスは kill する
  - セッション終了をきっかけとして kill はしない (デフォルト)

関連する man page

- systemd-logind(8), logind.conf(5), loginctl(1): セッション管理を行うサービス、設定ファイル、管理コマンド
- pam_systemd(8): ユーザログイン時の動作を行うための PAM モジュール
- systemd-user.conf(5): ユーザ用 systemd の設定は /etc/systemd/user.conf に分けて定義される。記述内容はシステム用のものと同じ。

やってみよう

- loginctl で現在のセッションを見たあと
  - loginctl show-session < セッション ID> で systemd-logind が追跡しているものを見る
  - loginctl seat-status <SEAT ID> でシートに対応づけられているデバイスを見る。これらのデバイスは対応するセッションのユーザからアクセスできるよう設定される。
- systemctl --user list-unit-files などを行いユーザセッション用の unit file を眺める

アドホックなコマンド実行の管理

systemd-run

- アドホックにプログラムを実行する際にも systemd の仕組みを使いたい
  - ulimit, cgroup, ロギングなどの環境準備を自作しなくても systemd に任せられる
  - シャットダウン時に”正しいお作法で”終了してくれる
- systemd-run で実行すると使い捨ての service または scope を作成する
  - service を activate する path,timer,socket も特定オプション作成
  - `systemd-run --uid=apache /usr/bin/foobar`

systemd-run 利用例

- systemd-run ls
  - 一時的な service を作成して systemd が実行。環境変数や標準出力が journal に記録されること、 CWD が / になることなども service と同様。
- systemd-run --uid=apache -t ls
  - uid を apache にし、標準入出力を端末へ接続して ls する。
- systemd-run --scope ls
  - 一時的な scope を作成して systemd-run がその中に入って ls を実行。 cgroup でリソース管理される以外は直接実行するのとほとんど変わらない。
  - scope の場合は systemd から実行されない点が大きく異なる

systemd-run の service と scope の違い

	service	scope
親プロセス	systemd	systemd-run を実行したプロセス (多くの場合 sh)
標準入出力	journal 。 -t オプションなどで変更できる。	親プロセスをひきつぐ
環境変数	systemd の service 用設定に従う	親プロセスをひきつぐ
systemd が制御する環境	cgroup, 環境変数、uid,gid などなど	cgroup だけ
実験用 shell 起動	systemd-run -S	systemd-run --scope bash

systemd-run を nohup のかわりに使う

	systemd-run	nohup
コマンド例	loginctl enable-linger systemd-run foobar	nohup foobar 2>&1 &
ログ出力先	journal オプションで任意ファイル	nohup.log
終了確認	systemd に unit 終了のログ	nohup.log を open しているプロセスを fuser で探す
シャットダウン時	SIGSTOP, SIGTERM -p ExecStop= 指定で任意コマンド実行	SIGSTOP, SIGTERM

taskset, ulimit 等のかわりに systemd-run を使う

systemd-run の -p オプションでリソース制御ができる。 -t オプションで端末との接続を維持する。

- `systemd-run -t -p BlockIOWeight=10  dd if=infile of=outfile count=10000 oflag=direct`
  - cgroup v1 で IO のスロットリングをしつつ実行する。 cgroup v2 では BlockIOWeight ではなく IOWeight を使う
- `systemd-run -t -p LimitCORE=0 ./mytest`
  - core 出力サイズを抑制して実行する
- `systemd-run -t -p CPUAffinity=0-1 ./mytest`
  - 実行可能 CPU を 0 と 1 だけに制約して実行する

関連する man page

- systemd-run(1): systemd-run の利用例など
- systemd.resource-control(5): cgroup に指定できる項目
- systemd.exec(5): プロセス実行で指定できる項目。 scope では指定できない。

やってみよう

- systemd-run env と systemd-run --user env で環境変数が異なる (後者はデスクトップ環境や X の設定なども含む) ことを確認する
- systemd-run -S で service のコンテキストで shell を実行する。 echo $PPID が 1 であること、 systemd-cgls で shell から実行されているプロセスも含めて service 内にあることを確認する
- systemd-run -S -p PrivateTmp=yes -p ProtectHome=yes のようなセキュリティ設定をおこなって /tmp や /home がどのように見えるか確認する

systemd-journald によるロギング

従来の syslog にあった課題

- syslogd が起動する前の boot 直後のログを統合できない
- 時刻が秒単位で粗いため複数ホストのログを集約すると前後関係がよくわからない。
- 時刻が localtime で夏時間がある地域では 1 時間隙間が進んだり戻ったりする
- 接続元、指定された優先度などのメタデータが失われる
- 構造化されていないため DB などに投入しにくい。 NUL などを含むデータをログに入れられないなど制限が厳しい。
- 保存されたログへのアクセス制御が粗い

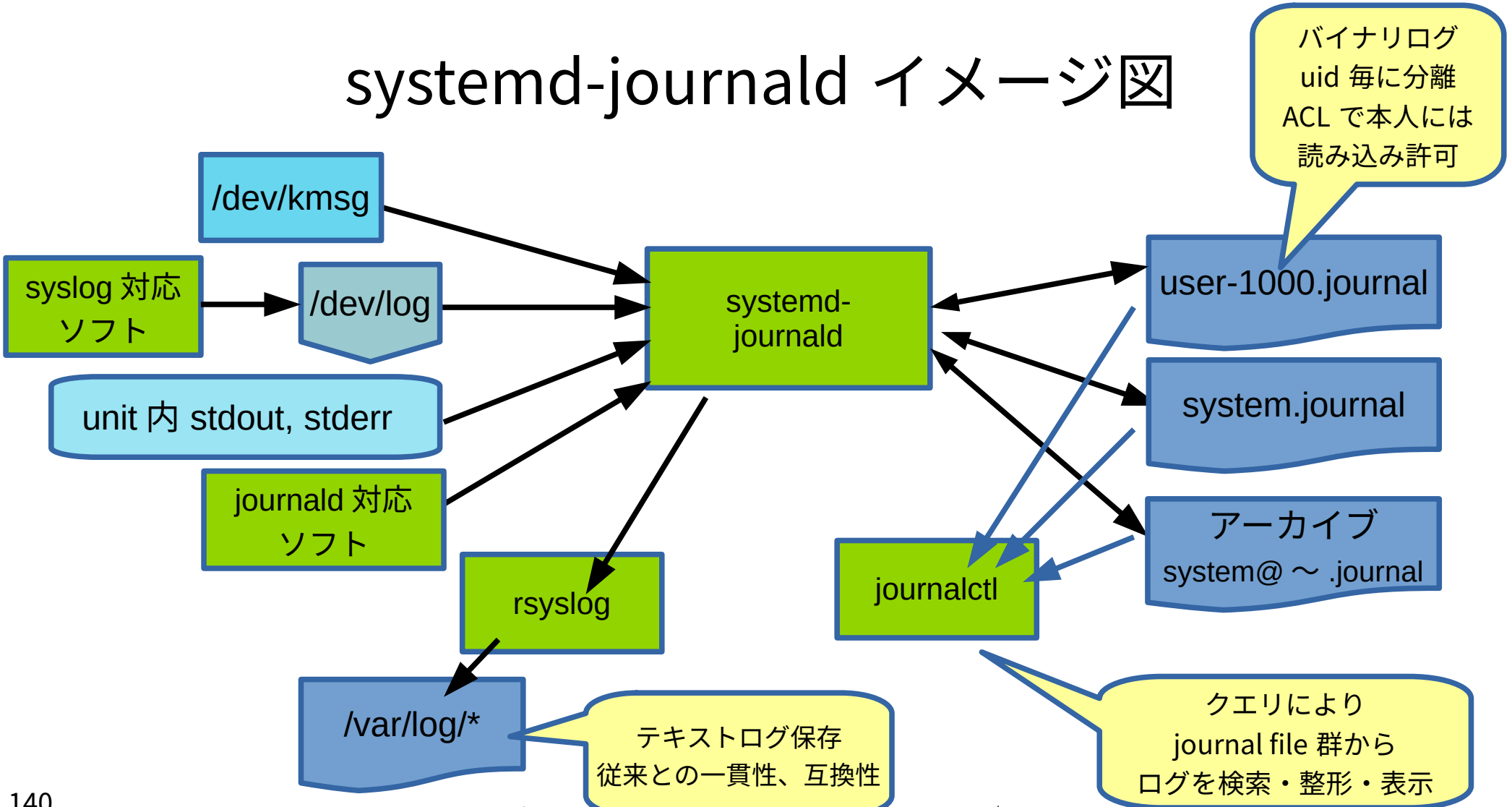
systemd とログを統合したい

- systemd はシステム起動のごく初期から起動される
  →同じ基盤でログの仕組みを作れば起動直後から記録できる
- unit 用に用意する環境の標準出力 (stdout, stderr) を /dev/null へ捨てるのではなくログへ保存して簡単にサービスを作りたい
- systemd が管理している環境についての ID 各種 (machine ID, boot ID, cgroup 名、 unit 名など) の情報をログに付与して、イメージを複製した複数 VM や再起動の前後、どの unit からのログかを識別したい

systemd-journald によるロギング

- 詳細なメタデータ付加
  - マイクロ秒 64bit でのタイムスタンプ , 起動からの時間 , ログの優先度 , systemd の unit, cgroup, SELinux コンテキスト , boot ID, machine ID 等
- 独自バイナリフォーマット
  - 圧縮・インデクシング・改竄検出対応
  - テキスト log が必要な場合 rsyslog と併用する (RHEL 7, 8 でのデフォルト)
- 自動的かつ透過的な rotate
- journalctl コマンドによるクエリ
- デフォルトでは journald のログは揮発性だが /var/log/journal を作ると永続化

systemd-journald イメージ図



journald で付加されるメタデータ例

カーソル位置	__CURSOR=s=70bed981e31a46e2a70c56776402e0eb;i=342;b=982826bc22454f079fd46ec94e2b
時刻 (usec)	__REALTIME_TIMESTAMP=1407830743019247
起動からの経過時間	__MONOTONIC_TIMESTAMP=1792842
BOOT ID	_BOOT_ID=982826bc22454f079fd46ec94e2b67fc
ログ優先度	PRIORITY=6
UID	_UID=0
GID	_GID=0
マシン ID	_MACHINE_ID=025b7ff2c8af43f78513996f06584c4c
ホスト名	_HOSTNAME=localhost.localdomain
syslog の ID と	SYSLOG_IDENTIFIER=systemd
ファシリティ	SYSLOG_FACILITY=3
ソースコード	CODE_FILE=src/core/unit.c
内の行番号	CODE_LINE=1115
内の関数名	CODE_FUNCTION=unit_status_log_starting_stopping_reloading
メッセージの ID	MESSAGE_ID=7d4958e842da4a758f6c1cdc7b36dcc5
接続方法	_TRANSPORT=journal
PID	_PID=1
コマンド名	_COMM=systemd
実行ファイル	_EXE=/usr/lib/systemd/systemd
CAPABILITY	_CAP_EFFECTIVE=1fffffffff
CGROUP	_SYSTEMD_CGROUP=/
コマンドライン	_CMDLINE=/usr/lib/systemd/systemd --switched-root --system --deserialize 20
SELinux コンテキスト	_SELINUX_CONTEXT=system_u:system_r:init_t:s0
systemd の unit 名	UNIT=rsyslog.service
メッセージ本文	MESSAGE=Starting System Logging Service...
出力時の時刻 (usec)	_SOURCE_REALTIME_TIMESTAMP=1407830743019153

journalctl によるクエリ例

journalctl コマンドのオプションで条件を指定。絞り込んで必要なログだけを抽出

- ログレベルの制限
  journalctl -p 4 ← warning(4) 以上の priority であること
- ユニットの制限
  journalctl -u NetworkManager.service ← 指定した unit からの出力であること
- 起動ごとの制限
  journalctl -b -1← 直近の boot より 1 つ前の起動以降のログ
- 時間の制限
  journalctl --since “2020-02-01” --until “2020-02-05” ← 時間ウィンドウの指定
- 任意のフィールド
  journalctl _COMM=dhclient ← < フィールド名 >=< 値 > で任意の値で絞り込み

journald のディレクトリ構造

- /run/log/journal (揮発) または /var/log/journal (永続) 以下に収集する
  - システムのログ ./<machine-id>/system.journal
  - 各ユーザのログ ./<machine-id>/user-*<uid>*.journal
    - ACL で各ユーザ本人へのアクセス許可を付与している
    - ログを uid で分離せずにまとめる設定も可能だが journalctl は透過的にまとめて検索するため (ユーザが自分のログを見られなくなる以外に) 特に使い勝手などの変化はない。
  - ローテートされたログ ./<machine-id>/system@*suffix*.journal
  - ローテートされたログ ./<machine-id>/user-*<uid>*@*suffix*.journal
  - ログの破損を発見すると、ログは *.journal~ という拡張子に変更され、新しいログへ書き出しを行う

journald の設定

/etc/systemd/journald.conf で設定する

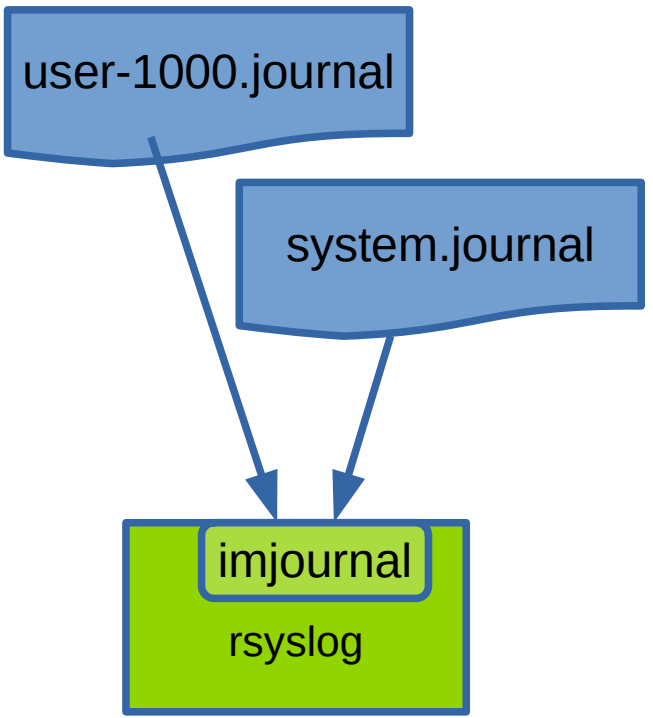
- 圧縮有無、 Seal 処理 (ハッシュを利用したログ改竄検出) 、ストレージへの同期間隔、ログ出力レート制限
- コンソールや syslog などへの forward の有無、ストレージへの保存や出力先毎の loglevel 指定
- ログローテート
  - サイズ毎、時間毎の rorate 処理
- アーカイブ削除
  - journal 全体のサイズ上限、ストレージの残容量下限による削除、指定時間以上経過で削除
  - 削除はローテートされたアーカイブファイル単位で行われる

ログ保存とローテートの設定

- journald のデフォルトではサイズベースでのローテートが指定される
  - ファイルシステムの 10% か 4GB の内小さい容量をログに使用
    - ログファイルはその 1/8 のサイズ (4GB の場合 512MB)
  - ファイルシステムの 15% か 4GB の内小さい空き容量を残す
- MaxFileSec= を指定してで毎日 / 毎月などのタイミングでもローテートを指定できる

rsyslog への forward

- rsyslogd の imjournal plugin が journald のログを監視して保存します
  - 逆に rsyslogd の出力先を journald にする omjournal plugin も提供されている
- RHEL のデフォルトでは journald は揮発性、rsyslogd で永続化
- 記録される情報としては journald が多いが、既存の syslog 対応ツールや外部との連携は rsyslog が便利なケースが多い



logger コマンドからのログ書き出し例

```
$ logger --journald <<end
FOO=bar
NONO=yes yes
end
$ journalctl -f -o json-pretty
(中略)
{
    "SYSLOG_IDENTIFIER" : "logger",
    "_TRANSPORT" : "journal",
    "NONO" : "yes yes",
    "_HOSTNAME" : "dragon",
    "_MACHINE_ID" : "03be3b9fa6a6fbc9e16a9cb9502f7e53",
    "_GID" : "1000",
    "__CURSOR" : "s=1b5712bd115e4a668a32e23ad77d617f;i=4adca;b=(略)",
    "_PID" : "304806",
    "_BOOT_ID" : "632ce2bdb26a46c9ae6df967003de064",
    "FOO" : "bar",
    "_UID" : "1000",
    "__REALTIME_TIMESTAMP" : "1582007587813923",
    "__MONOTONIC_TIMESTAMP" : "167902611176",
    "_SOURCE_REALTIME_TIMESTAMP" : "1582007587813856"
}
```

systemd unit と journal

- unit から何か実行するもの (service, socket, mount, swap) について stdout, stderr は journal に繋がるのがデフォルト (明示的に指定すれば syslog, kmsg, 端末その他に接続も可)
- unit の設定でログ出力について調整できる
  - LogLevelMax= ログ出力のレベルに上限を指定 (0: emerg から 7:debug まで。数字が大きいほどログが多くなる。)
  - LogRateLimit*= ログ出力頻度の制限
  - LogExtraFields= ログに追加のフィールドを付与 (複数インスタンスを識別するための情報など)
  - Syslog*= syslog のフィールドを指定

関連する man page

- journalctl (1): journal の検索と journald の管理操作
- journald.conf (5): journald の設定
- systemd.journal-fields(7): journald が利用するフィールドの説明

やってみよう

- journalctl -b -u NetworkManager -p 4 などの検索をいくつか試してみる
- journalctl -o verbose で表示されるフィールドの意味を man で調べる
- logger で journald へログを吐いて確認する
  (key=value 形式の複数行で記述する必要がある)

  ```
  $ logger --journald <<end
  FOO=bar
  NONO=yes yes
  end
  ```

sysvinit からの移行

sysvinit のスクリプトをどうする？

- 何もしなくてもかなり動きます
  - /etc/init.d/ 以下のスクリプトを systemd-sysv-generator で自動変換して unit file を自動生成する
  - 互換用の service スクリプトから start/stop/restart などの LSB に準拠したサブコマンドは実行できる
- サブコマンドを独自に拡張している場合は 対応作業が必要

自動生成される unit file の例

```
# Automatically generated by systemd-sysv-generator

[Unit]
SourcePath=/etc/init.d/sysv-init-script
Description=LSB: A sysV init script
After=remote-fs.target systemd-journald-dev-log.socket

[Service]
Type=forking
Restart=no
TimeoutSec=5min
IgnoreSIGPIPE=no
KillMode=process
GuessMainPID=no
RemainAfterExit=yes
ExecStart=/etc/init.d/sysv-init-script start
ExecStop=/etc/init.d/sysv-init-script stop
```

init スクリプト移行例 - iptables (1)
service コマンドの独自拡張

- iptables は /etc/init.d スクリプト内で独自拡張を行っている
  - service iptables save
- 以下のファイルで save コマンドを実装
  - /usr/libexec/iptables/iptables.init (もとの init スクリプト)
  - /usr/libexec/initscripts/legacy-actions/(ユニット名)/(サブコマンド名) に対応するスクリプトを配置
    - /usr/libexec/initscripts/legacy-actions/iptables/save
    - iptables.init スクリプトの save サブコマンドを呼ぶ
- これらを用意することで、独自拡張命令を以前と同様に実行できる

  ※service スクリプトからは呼びだせるが systemctl から呼びだす方法はない

init スクリプト移行例 - iptables (2)

```
[Unit]
Description=IPv4 firewall with iptables
After=syslog.target
ConditionPathExists=/etc/sysconfig/iptables

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/libexec/iptables/iptables.init start
ExecStop=/usr/libexec/iptables/iptables.init stop
Environment=BOOTUP=serial
Environment=CONSOLETYPE=serial
StandardOutput=syslog
StandardError=syslog
```

- ConditionalPathExists= でファイルが存在する場合のみ実行。
- iptables ではプロセスが常駐するわけではないので、 Type=oneshot と RemainAfterExit=yes を指定する。これで iptables.init が実行終了しても active 状態が維持され、必要に応じて終了処理が行われる。

sysvinit から systemd へ移行する際の注意

- サービスを起動するものは基本的に type=forking にする
  - oneshot だと二重起動の対応等をされない
  - エラーチェックしていないスクリプトなどではプロセスが異常終了しても発見されない
- 代表プロセスを発見する方法を指定する
  - PID ファイル
- 設定して一瞬 (～数十秒くらい) で終了するなら oneshot

関連する man page

- systemd-sysv-generator(8): init スクリプトを systemd の unit へ変換するプログラム
- daemon(7) Integration with systemd 節： 既存のデーモンを systemd で管理するための概要
- sysvinit と systemd の既知の非互換性
  https://www.freedesktop.org/wiki/Software/systemd/Incompatibilities/

systemd の動作を眺める

systemd の様子を眺めたい

systemd は中に様々な状態を持ち処理をしているので中の様子を眺めたい

- 人が眺めたい :
  - 思ったように unit が動かないなどのトラブルシュート
  - unit の定義を見ても動作がよくわからない時に理解するため
- プログラムから眺めたい :
  - systemd の流儀で作り直すのは大変すぎるので systemd の状態をとってきたい
  - systemd 環境と systemd 以外の環境のどちらでも動く必要がある

systemd-analyze

- systemd の分析、デバッグ用のツール
- unit file を書いたりトラブルシュート時に便利なサブコマンド
  - `systemd-analyze dump` → systemd の全 unit の現在の詳細情報を可読なテキスト形式でダンプする。作業前後での違いなどを保存する。
  - `systemd-analyze dot <unit 群 >` → graphviz で指定 unit 群の依存関係を図示する。
  - `systemd-analyze verify <unit file>` → lint 的に unit file をチェックする。ディレクトリや Exec* の対象存在有無など。
  - `systemd-analyze condition <condition の定義 >` → 各 condition のチェックと全体としての成否を表示する。
  - その他時刻や時間の表現がどう認識されるかテストする機能もあります。

systemd の log level を変える

systemd がどこと通信して何をしているか見たいときは debug レベルのログを見ます

- 起動時の問題であれば grub で kernel のオプションに `systemd.log_level=debug` のように指定する (起動時はログがとても多いので注意)
- /etc/systemd/system.conf に LogLevel=debug と記述して `systemctl daemon-reload` します。他の設定ファイルも全て読み直します。
- 実行中に loglevel だけを変更するには以下のように systemd-analyze コマンドで指定する
  - 現在の log level 確認 (デフォルトは info):
    `systemd-analyze log-level`
  - log level を debug にする :
    `systemd-analyze log-level debug`

unit の状態でシェルスクリプトの動作を変える

- unit の状態でシェルスクリプトなどの動作を変えたい場合は、systemctl のサブコマンド is-active, is-enabled, is-failed, is-system-running を使う
  - systemctl status や systemctl show だと sysvinit のスクリプトの状態が最新でない場合がある
- 出力はテキストと exit code 。 --quiet オプションをつけるとテキスト出力は抑制される。

余談 : systemd-sysv-install

- sysvinit 形式のスクリプトがある場合の is-enabled の確認は /lib/systemd/systemd-sysv-install で行われる。
- /lib/systemd/systemd-sysv-install は RHEL/Fedora では chkconfig の別名で enable/disable を sysvinit のディレクトリの状態から返す
- systemctl status や systemctl show では unit の状態を返すが systemd-sysv-install によるチェックを行わないので最新でない場合がある

関連する man page

- systemd-analyze(1): systemd-analyze の各サブコマンドの実行例、出力例あり
- systemctl(1): is-enabled などの節には出力と対応する exit code の表あり

やってみよう

- graphviz を入れて unit 間の依存関係画像を見てみる

```
$ systemd-analyze dot 'cockpit.*' | dot -Tsvg >cockpit.svg
$ eog cockpit.svg
```

- systemd-analyze calendar daily として、 timer unit などで ’daily’ と指定があると実際に何時何分を指しているのかを確認する
- systemd の log level を debug に変更し、 systemctl などのコマンドを実行した時にどのようなログが出るのか見る

周辺ツールなど

systemd-tmpfiles

- 一般に永続的でないディレクトリが多数ある /dev, /run, /tmp など
  - live CD のような環境では /var 以下も永続的であるとは限らない
- systemd-tmpfiles は起動時、または定期的なセットアップと掃除をこれらのディレクトリに対して行う
  - ファイルやディレクトリ、リンク、パイプ、デバイスファイルなどを**作成**
  - 特定ファイルを**特定のコンテンツで初期化**
  - 特定のディレクトリ以下のファイルを削除したり、ファイルそのものは残して **truncate** したり、 uid,gid, 権限 , ACL などを**変更**する
  - 最終アクセスから指定時間以上経過していたら**削除** など

systemd-tmpfiles

- systemd-tmpfiles --cat-config で全設定を出力
  - 設定は宣言的に行う。設定内容の読み方については tmpfiles.d(5) を参照
  - パッケージに含まれる設定を変更したい場合は /etc/tmpfiles.d 以下に /usr/lib/tmpfiles.d と同じ名前のファイルを作るとファイル単位で上書きされる
  - 作成 (または削除) されるディレクトリやファイルに親子関係があれば適切な順序で実行する
- 実行は systemd-tmpfiles-*.service, systemd-tmpfiles-*.timer から行われる。
  - デフォルトでは起動時に /dev/ とそれ以外に分けて setup を呼びだし、起動から 15 分後に 1 回目の clean 、その後 1 日 1 回 clean を繰り返す

hostnamectl

- ホスト名の設定
  - `hostnamectl set-hostname host.domain.example.org`
- ホスト名の確認

```
$ hostnamectl$ hostnamectl
   Static hostname: rhel8.example.com
         Icon name: computer-vm
           Chassis: vm
        Machine ID: 21975e50f73142d5b4a92eb05209e9fa
           Boot ID: 83c339091094442b8645fec5699c7e11
    Virtualization: kvm
  Operating System: Red Hat Enterprise Linux 8.1 (Ootpa)
       CPE OS Name: cpe:/o:redhat:enterprise_linux:8.1:GA
            Kernel: Linux 4.18.0-147.0.3.el8_1.x86_64
      Architecture: x86-64
```

machine id と boot id

- 各ホストや起動・再起動での一意性を確保したいので UUID のような ID があると便利
  - Machine ID: /etc/machine-id にある id 。ホスト毎に生成されランダム。 /etc/machine-id を削除して再起動すると自動で新しい乱数が生成される。
  - Boot ID: /proc/sys/kernel/random/boot_id にある id 。起動毎に生成されランダム。
- 利用例 : journald によるログには machine id, boot id が含まれている。

関連する man page

- systemd-tmpfiles(8), tmpfiles.d(5): systemd-tmpfiles と設定ディレクトリ、設定ファイルについて
- hostnamectl(1): ホスト名の設定、確認について
- random(4): linux kernel の random モジュールについて。boot_id もこのモジュールが生成している。

参考資料

man page

今まで参照した man page 群でわかるとおり systemd は多数の充実した man page が提供される

- systemd.index(7) - systemd の man page 一覧
- systemd.directives(7) - unit file の directive がどの man page に書かれているかの一覧。既存 unit file 読解時に便利

ホームページ、 blog

- https://systemd.io/ systemd project ホームページ
- https://www.freedesktop.org/wiki/Software/systemd/ 旧 systemd ホームページ
  - 特に運用管理者には The systemd for Administrators Blog Series を推奨します

スライド資料

- Demystifying systemd
  - 2018 年版 : RHEL7 ベース。これから使いはじめる人むけ

    https://www.redhat.com/files/summit/session-assets/2017/S103870-Demystifying-systemd.pdf
  - 2019 年版 : RHEL8 ベース。新機能に興味がある人むけ

    https://www.redhat.com/files/summit/session-assets/2019/T4D2A2.pdf