Les conteneurs sous Linux

Timothée Ravier
siosm@floss.social - tim.siosm.fr/cours - github.com/travier

5e année cycle ingénieur, filière STI
Option Sécurité des Systèmes Embarqués et du Cloud (2SEC)

2024 - 2025

Sommaire

Virtualisation ?

> On parle de virtualisation légère, virtualisation au niveau du système d'exploitation ou de conteneurs par opposition à la virtualisation lourde, complète, totale d'un système d'exploitation.

Cas d'usage des conteneurs

Objectifs :

- Regrouper logiquement les composants d'une application
- Isoler chaque applications vis à vis du système et des autres applications
- Maîtriser la consommation en ressources

Historique

- **1982** : chroot : exécuter un programme avec un `/` (root) distinct
- **2000** : FreeBSD Jails : ajoute des restrictions au niveau des intéractions avec le noyau, le réseau et virtualise le super utilisateur
- **2001** : Linux Vserver : concept de *context* et fonctionnalités similaire aux jails pour Linux. Patch noyau uniquement, support très limité
- **2004** : Solaris Zones : concept de *system call translation*
- **2005** : OpenVZ : patch noyau Linux uniquement, support limité. Certaines fonctionnalités ont été intégrées upstream (CRIU)

Conteneurs sous Linux

Attention ! Pas de concept de conteneur dans le noyau Linux !

Il faut combiner « manuellement » :

- l'isolation (modulable) avec les **namespaces**
- la gestion des ressources et de l'accès à certains périphériques avec les **cgroups**
- les restrictions de permissions avec les **capabilities**, les **filtres seccomp** et les **Linux Security Modules** (SELinux / AppArmor)

Isolation avec les namespaces

Isolation avec les namespaces (1)

Principe général :

- Crée un ou plusieurs espaces de nom (namespaces ou ns) pour séparer le contexte d'exécution d'un processus par rapport aux autres
- Isole les objets créés dans un namespace vis à vis des autres namespaces
- Lorsque tous les processus d'un namespace ont terminé leur exécution, les objets restant sont détruits ou renvoyés dans le namespace parent

Isolation avec les namespaces (2)

- `CLONE_NEWNET`, `CLONE_NEWNS`, `CLONE_NEWPID`, `CLONE_NEWUTS`, `CLONE_NEWIPC`, `CLONE_NEWUSER`, `CLONE_NEWCGROUP`, `CLONE_NEWTIME`
- Créer un processus fils dans des nouveaux namespaces :

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...);
```

- Déplacer le processus courant dans un nouveau namespace :

```
int unshare(int flags);
```

- Déplacer le processus courant dans un namespace existant :

```
int setns(int fd, int nstype);
```

- Voir namespaces(7), unshare(2), clone(2), setns(2)

Outils en ligne de commande

- `unshare` : lancer un programme en utilisant de nouveaux namespaces
- `nsenter` : lancer un programme avec les namespaces d'un programme existant
- Visualiser les namespaces associés à un programme :

```
$ ls -l /proc/1006/ns/*
... /proc/1006/ns/cgroup -> 'cgroup:[4026531835]'
... /proc/1006/ns/ipc -> 'ipc:[4026531839]'
... /proc/1006/ns/mnt -> 'mnt:[4026531840]'
... /proc/1006/ns/net -> 'net:[4026531993]'
... /proc/1006/ns/pid -> 'pid:[4026531836]'
... /proc/1006/ns/pid_for_children -> 'pid:[4026531836]'
... /proc/1006/ns/user -> 'user:[4026531837]'
... /proc/1006/ns/uts -> 'uts:[4026531838]'
```

CLONE_NEWNS

Crée un namespace pour les points de montage :

- Présent depuis le noyau 2.4.19 (2002)
- Permet au fils de monter/démonter des systèmes de fichier sans impacter le namespace parent

```
$ ls /mnt
a b
$ unshare --mount
[newns]# umount /mnt
[newns]# ls /mnt
[newns]# exit
$ ls /mnt
a b
```

CLONE_NEWUTS

Crée un namespace UTS :

- Présent depuis le noyau 2.6.19 (2006)
- Contient le domain name et le host name
- Permets au fils de changer ces informations sans impacter le namespace parent

```
$ hostname
foo.bar
$ unshare --uts
[newns]# hostname toto
[newns]# hostname
toto
[newns]# exit
$ hostname
foo.bar
```

CLONE_NEWIPC

Crée un namespace pour les IPC :

- Introduit dans le noyau 2.6.19, complété dans le 2.6.30 (2009)
- Vue isolée des IPC System V : seuls les processus de ce namespace peuvent accéder à ces objets
- Destruction des objets à la fin du namespace

```
$ ipcs --summary
...
segments allocated 4
...
$ unshare --ipc
[newns]# ipcs --summary
...
segments allocated 0
...
```

CLONE_NEWNET

Crée un namespace pour les interface réseaux :

- Introduit dans le noyau 2.6.24, complété dans le 2.6.29 (2009)
- Le fils dispose de sa propre stack réseau : interfaces, stacks IPv4/IPv6, tables de routage IP, règles de parfeu, sockets, etc.

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
...
2: enp0s31f6: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
...
$ unshare --net
[newns]# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

CLONE_NEWNET

- Une interface physique ne peut être que dans un seul namespace à la fois
- Rattachée au namespace initial à la destruction du namespace
- Utilisation de paires d'interfaces virtuelles (`veth`) pour lier les namespaces :

```
$ ip link add ... type veth
...
```

CLONE_NEWPID

Crée un namespace pour les PID :

- Présent depuis le noyau 2.6.24 (2008);
- Les PIDs recommencent à 1 dans ce namespace
- Les processus du namespace parent voient les processus du namespace fils avec des PIDs différents
- Les processus dans un namespace ne peuvent voir que ceux du même namespace et des namespaces créés par des processus de leur namespace;

CLONE_NEWPID

```
$ ps aux
root           1  0.0  0.0 172424 17216 ?        Ss   Jan29   0:03 /usr/lib/systemd/systemd ...
root           2  0.0  0.0      0     0 ?        S    Jan29   0:00 [kthreadd]
...
tim        85096  0.4  0.2 25759272 87136 ?      Sl   20:51   0:00 /usr/bin/google-chrome ...
tim        85127  0.0  0.0 235748  5044 pts/3    R+   20:51   0:00 ps aux
$ unshare --mount --pid --fork
[newns]# mount proc -t proc /proc/
[newns]# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 233184  5928 pts/3    S    22:52   0:00 -bash
root          30  0.0  0.0 234384  3652 pts/3    R+   22:52   0:00 ps aux
```

CLONE_NEWUSER

Crée un namespace pour les UID/GID :

- Introduit dans le noyau 2.6.24, finalisé dans le 3.8 (2013)
- Associe un ensemble d'UID/GID dans le nouveau namespace au couple UID/GID initial
- `root` dans un user namespace sans l'être en dehors
- Le noyau s'assure qu'il n'y as pas d'élévation de privilèges par rapport au couple UID/GID initial
- Nécessite la mise en place d'une correspondance entre les UIDs à l'intérieur et à l'extérieur du user namespace

CLONE_NEWCGROUP

Crée un namespace pour les hiérarchies cgroups :

- Introduit dans le noyau 4.6 (2016)
- Abstraction des hiérarchies cgroups pour les conteneurs

```
$ cat /proc/self/cgroup
0::/user.slice/user-1000.slice/user@1000.service/app.slice/app-....scope
$ sudo unshare --cgroup
[newns]# cat /proc/self/cgroup
0::/
```

Privilèges requis pour créer un namespace ?

- En général, privilèges équivalent à `root` nécessaires.
- Voir cours sur les *Capabilities*

Implémentations sous Linux

Création d'un conteneur sous Linux

Eléments nécessaires :

- gestion des ressources et de l'accès à certains périphériques avec les cgroups
- isolation (modulable) avec les namespaces
- restrictions de permissions avec les capabilities, les filtres seccomp et les Linux Security Modules (SELinux / AppArmor)
- image du conteneur (arborescence propre au conteneur : contenu du `/`)
- accès au réseau

Container runtime (`runc`, `crun`)

Programme responsable de la création d'un conteneur et de la configuration de l'environnement d'exécution :

- mise en place des cgroups et limites UNIX
- mise en place des namespaces
- configuration des capabilities, filtres seccomp, LSM

Container engine (`docker`, `podman`)

Programme responsable de la gestion du cycle de vie d'un conteneur :

- gestion des images de conteneurs (création, récupération à distance, envoi, instanciation, etc.)
- gestion des points de montage, volumes persistants, etc.
- mise en place des interfaces réseau (virtuelles)
- suivit des conteneurs en cours d'exécution.

Utilise un **container runtime** (`runc`, `crun`) pour lancer les conteneurs.

Historique des implémentations

- 2008 : LXC / LXD (et Incus depuis 2023)
- 2010 : systemd-nspawn
- 2013 : Docker (Moby engine)
- 2014 : rkt
- 2015 : runc, containerd, Clear Containers et Kata Containers
- 2016 : Minijail
- 2017 : railcar & CRI-O
- 2018 : Podman, Buildah, Skopeo, gVisor et Nabla containers
- 2019 : crun
- 2022 : youki

Implémentations les plus utilisées

- LXC et LXD / Incus
- Docker (Moby engine)
  - composé de `dockerd`, `containerd` et `runc`
  - seulement `containerd` et `runc` utilisés avec Kubernetes
- Podman, Buildah et Skopeo
  - `runc` ou `crun` en *conteneur runtime*
- CRI-O
  - utilisé uniquement avec Kubernetes
- Kata Containers et gVisor
  - Utilise la virtualisation avec KVM

LXC (runtime) et LXD / Incus (engine)

- Objectif : Système complet dans un conteneur
- Image : simple `tar.gz` d'un `/` (obtenu avec `debootstrap`, `dnf --install-root=...` ou équivalent)
- Création de conteneurs à partir de templates
- LXD : gestionnaire de conteneurs LXC
- Fonctionnement similaire à un système classique :
  - Plus proche de la virtualisation « classique »
  - Processus `init` (`systemd` ou autre) et services systèmes
- Incus : Fork communautaire suite à la reprise par Canonical

Docker (Moby engine) (1)

- Objectif : Une application par conteneur
- Dockerfile : « Script » pour créer une image de conteneur

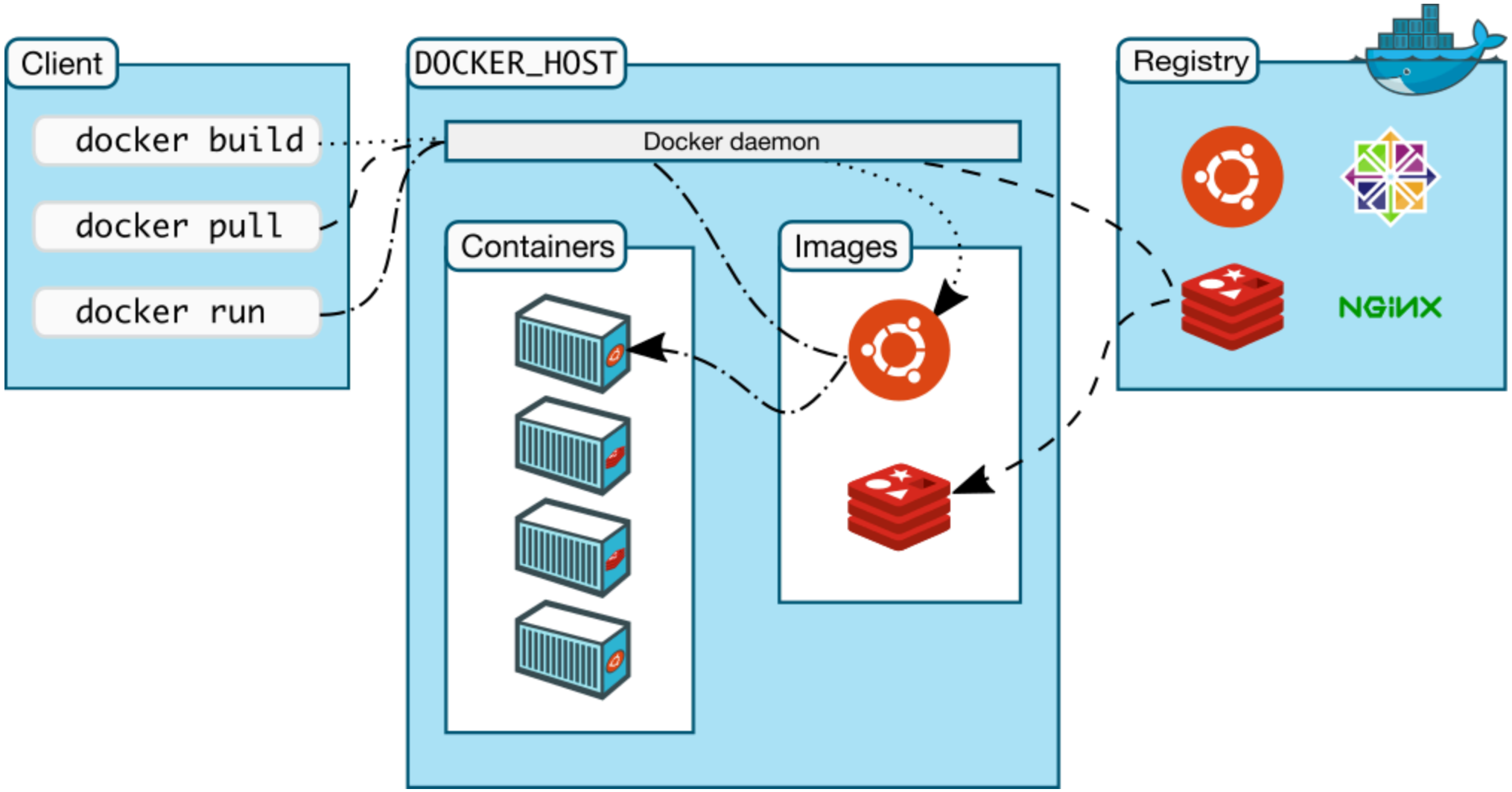
```
FROM ubuntu
RUN apt-get install nginx
CMD ['nginx']
```

- DockerHub : *Container registry*
  - Hébergement d'images de conteneurs
  - Registre par défaut avec Docker
- Docker Compose :
  - Gérer plusieurs conteneurs et leurs réseaux, volumes, etc.
  - Configuration déclarative

Docker (Moby engine) (2)

- Avantages :
  - Simplificité (récupération des images, utilisation)
  - Consommation en ressources (mémoire et disque) réduite
  - Gestion d'une application et non plus d'un système
- Désavantages :
  - Chaque conteneur doit être maintenu à jour
  - Les applications classiques ne sont pas prévues pour fonctionner en tant que PID 1 dans un conteneur : processus zombie, etc.
  - Aucun service support disponible dans le conteneur

Docker : architecture



https://docs.docker.com/engine/docker-overview

Podman, Buildah, Skopeo

- Podman : Ré-écriture de Docker sans utiliser de démon système
- Buildah : Construire des images de conteneur
  - à partir d'un Dockerfile
  - à l'aide de commandes arbitraires
- Skopeo : Manipulation d'images de conteneur
  - téléchargement, envoie, metadata, etc.

Podman, Buildah, Skopeo vs Docker ?

- Avantages :
  - Pas de démon central
  - Bonne intégration avec systemd : Quadlets
    - Alternative à Docker Compose
  - Fonctionnement et ligne de commande équivalente à Docker
  - Séparation des tâches $\Rightarrow$ meilleure sécurité
  - Rootless : conteneurs non privilégiés avec utilisateur « `root` »
- Désavantages :
  - Légères différences de fonctionnement par rapport à Docker

Standards et compatibilité

Standards et compatibilité

- La prolifération des outils de gestion de conteneurs a poussé à la standardisation
- Création de l'Open Container Initiative
- Plusieurs spécifications OCI
  - OCI Image Format Specification (+ Docker Format v2)
  - OCI Runtime Specification (`runc`, `crun`)
  - OCI Distribution Specification (registres)
- Making Sense of Container Standards (PodCTL - Enterprise Kubernetes)
- Making Sense of Container Standards and Foundations: OCI, CNCF, appc and rkt

Gestion statique des ressources

L'overcommit

- **Concept essentiel** permettant la co-location des VMs et conteneurs sur un système
- Chaque processus a l'impression d'être le seul à utiliser le matériel
- Chaque processus « dispose » de plus de ressources que ce qui est réellement disponible
- Exemple : l'espace d'adressage virtuel :
  - $2^{32} \approx$ 4Go en 32bits, pour l'instant 2^{48} en 64 bits
  - `/proc/sys/vm/overcommit_memory` & `/proc/sys/vm/overcommit_ratio`
- Linux kernel : Overcommit Accounting
- Linux kernel : Memory Layout on AArch64 Linux

rlimits : resource limits

- Limite les ressources disponibles pour l'ensemble des processus d'un utilisateur ou d'un groupe
- Affichage / manipulation des limites avec `ulimit`
- Deux types de limites : `hard` et `soft`
- Limite `soft` :
  - limite appliquée par le noyau
  - définissable entre 0 et la limite `hard`
- `CAP_SYS_RESOURCE` pour augmenter les limites `hard`
- Réduction des limites `hard` irréversibles pour `!root`
- Limites initiales appliquées lors du login par PAM, configurées dans `/etc/security/limits.conf`

rlimits : resource limits

cpu time (seconds)	unlimited	locked-in-memory size (kbytes)	8192
file size (blocks)	unlimited	address space (kbytes)	unlimited
data seg size (kbytes)	unlimited	file locks	unlimited
stack size (kbytes)	8192	pending signals	127183
core file size (blocks)	unlimited	bytes in POSIX msg queues	819200
resident set size (kbytes)	unlimited	max nice	0
processes	127183	max rt priority	0
file descriptors	1024	rt cpu time (microseconds)	unlimited

Quota disque

- Limite les ressources disque par `user` et par `group`
- Limite `soft` & `hard` par système de fichier
- La limite `hard` ne peut pas être dépassée
- La limite `soft` peut être dépassée pour un temps donné
- Une fois ce temps dépassée, la limite `soft` devient `hard`
- L'utilisateur doit libérer de la place pour redescendre au dessous de la limite `soft` avant de pouvoir écrire à nouveau des données
- Quota disque par « projets » avec XFS : voir xfs_quota(8)

Espace disque réservé pour les processus privilégiés

- Possibilité de réserver de l'espace sur un système de fichier
- Permet aux processus privilégiés de continuer à fonctionner lorsque le disque est « plein »
- `$ tune2fs -m reserved-blocks-percentage`
- Valeur par défaut : 5 % de la taille du système de fichier

Priorités, IO, et CPU

- nice level : Défini la priorité d'accès aux ressources CPU pour un processus
- ionice : Défini la priorité d'accès aux périphériques pour un processus
- cpu affinity : Oblige un processus à utiliser des cœurs définis

Gestion dynamique : cgroups

Limites du modèle statique

- Flexibilité très limitée
- Gestion manuelle des changements de limites :
  - Pas de répartition automatique (entre utilisateurs, services, applications)
- Peu adapté au modèle de co-location d'un grand nombre de machines virtuelles ou de conteneurs :
  - Besoins variables
  - Un seul utilisateur
- Peu adapté à la gestion d'un service constitué de plusieurs processus

cgroups

- Introduit dans le noyau 2.6.24
- Permets de regrouper des processus en leur associant un label
- Organisation hiérarchique
- Les processus fils héritent du cgroup du père
- Représenté par un système de fichier virtuel : `/sys/fs/cgroup/`

Exemple : libvirt : Control Groups Resource Management

cgroups : `controllers`

- Configuration d'un ou plusieurs `controllers` pour chaque groupe de tâches
- Tous les processus d'un groupe se voient imposer les mêmes contraintes
- cgroups(7) : liste complète ainsi que les versions du noyau à partir desquelles chaque `controller` est disponible
- Liste des `controllers` disponibles sur un système :

```
$ cat /proc/cgroups
#subsys_name    hierarchy       num_cgroups     enabled
cpuset  0       358     1
cpu     0       358     1
cpuacct 0       358     1
blkio   0       358     1
memory  0       358     1
devices 0       358     1
...
```

cgroups : `controllers` disponibles

- cpu (& cpuacct v1) : mesure et limite l'utilisation CPU
- cpuset : répartition sur les CPUs / nœuds NUMA
- memory : mesure et limite l'utilisation de la mémoire
- blkio/io : mesure et limite l'utilisation des périphériques de type block
- pids : limite le nombre de PID disponibles
- freezer : pause l'exécution des processus du groupe
- perf_event : groupes spécifiques au sous-système *perf*
- rdma : limite les accès RDMA et InfiniBand
- hugetlb : limite l'utilisation des Huge Page
- devices : restreint l'accès aux devices disponibles
- net_cls & net_prio (v1) : classe et définit des priorités pour les paquets réseaux

cgroups v1 & v2

Deux versions des cgroups :

- cgroups v1 :
  - chaque `controller` a sa propre hiérarchie
  - pose des problèmes de gestion, performance, complexité
  - diverses limites liées au design
- cgroups v2 :
  - une seule hiérarchie commune à tous les `controllers`
  - configuration différente pour certains `controllers`
  - disponibilité des `controllers` v2 équivalente à v1 à partir du noyau 5.6.

Linux kernel : cgroups v2

cgroups v1 ou v2 ?

- Utiliser la version 2 (sauf bonne raison, vieux systèmes, etc.)
- Version 2 désormais supportée par tous les gestionnaires de conteneurs :
  - podman 1.7.0 (Janvier 2020)
  - Docker 20.10 (Décembre 2020)
- Version 2 disponible par défaut :
  - Fedora 31 (2019), CentOS 9 Stream & RHEL 9 (2022)
  - Debian 11 (2021), Ubuntu 21.10 (2021)
- Supporté par Kubernetes depuis la version 1.25 (2022)
  - Kubernetes: About cgroups v2
  - Openshift 4.13 (May 2023)

Suite

systemd