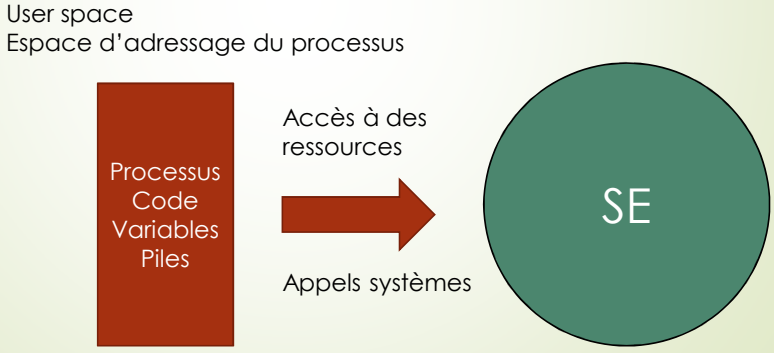
Conteneurs Linux: l'isolation et les namespaces

Anthony BUSSON

Exécution d'un processus

- Un processus s'exécute au sein de son espace d'adressage (en RAM).
- Il n'a pas le droit de lire ou d'écrire en dehors, il n'a pas directement accès aux ressources (fichiers, réseau, etc.) : en cela il est par nature isolé.
- L'interaction avec les ressources ou les autres processus se fait obligatoirement via des appels systèmes.
- Si il existe des services d'isolation, c'est forcément le système qui les implémentent.



Conteneurs: les outils linux

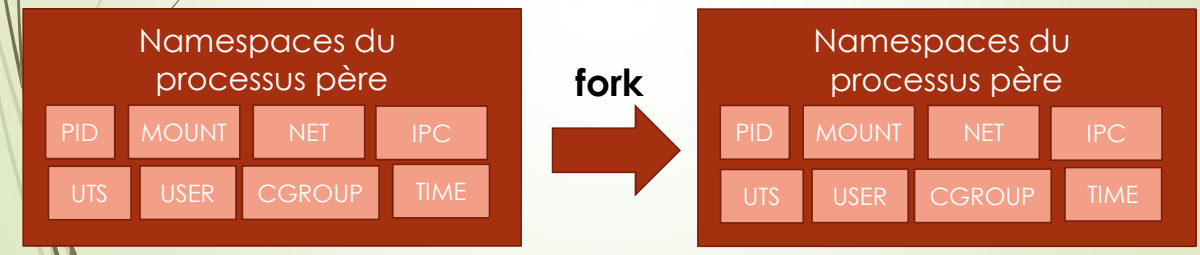
- Nous nous intéressons à l'isolation sous Linux (en cours sous windows)
- Les conteneurs sont basés sur des fonctionnalités du noyau Linux
  - namespace
  - cgroups
  - sécurité

namespace

- Depuis 2002
  - Linux 2.4.19 (dernière version 5.15.5)
- 8 différents namespace depuis la version 5.6 du noyau Linux
- Chaque type de namespace définit ce que le processus ou groupe de processus peut accéder et modifie le comportement des appels systèmes associés
  - mount: montage des périphériques / de systèmes de fichiers
  - process ID (pid)
  - network: interface, table de routage, ports, iptable, etc.
  - ipc (inter processes communication)
  - UTS (Unix Time Sharing): host and domain name
  - user ID: mapping des ID
  - time namespace: le temps peut être différent d'un namespace à l'autre (depuis 2020)
  - Control group namespace

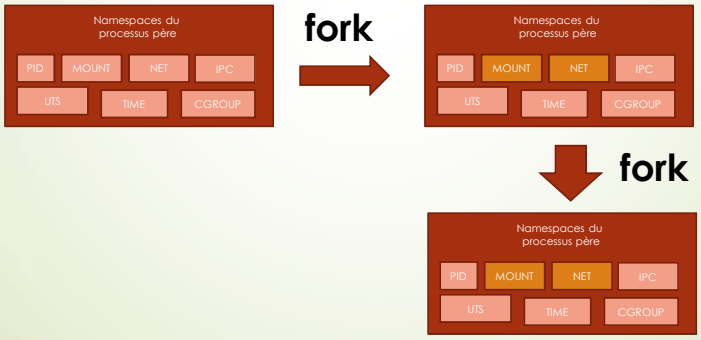
Principe

- Chaque processus appartient à chacun des namespace
- Par défaut, un processus hérite des namespace de son père
- Un processus peut créer son propre namespace
  - Par défaut, tous ses fils appartiendront au même namespace (sauf nouvel appel à unshare)



Principe

- Chaque processus appartient à chacun des namespace
- Par défaut, un processus hérite des namespace de son père
- Un processus peut créer son propre namespace
  - Par défaut, tous ses fils appartiendront au même namespace (sauf nouvel appel à unshare)

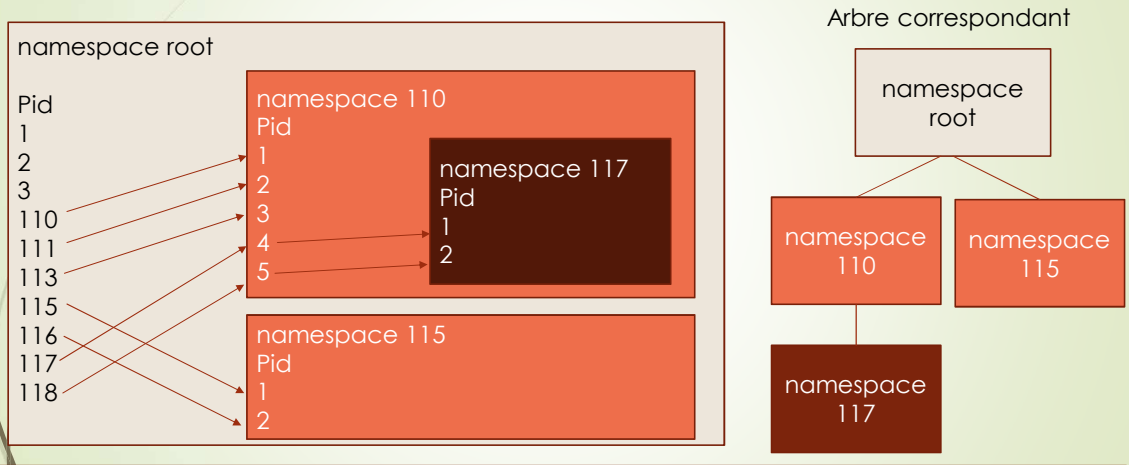


Namespace pid

- Isolation des processus
  - Lorsqu'un processus est assigné a un nouveau namespace « pid »
    - Il récupère le pid 1 dans son propre namespace
    - Ses fils auront les pid 2, 3, etc.
  - Les processus ne peuvent interagir qu'avec les processus de leur propre namespace ou des namespace enfants
    - Interagir = utiliser des appels systèmes avec eux (signaux, etc.)

Namespace pid: détail (1)

- Les namespace sont emboîtés (nested)



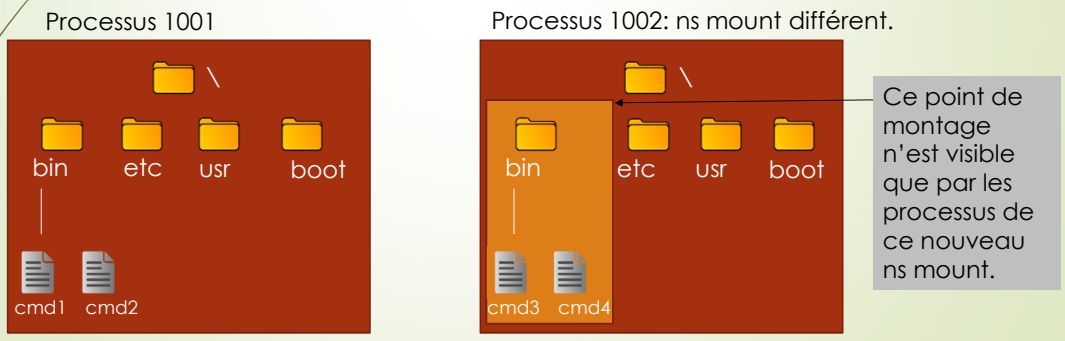
Le processus de pid 117 a 3 PID: un dans son propre namespace (1 ici), et un dans chacun des namespaces de niveau supérieur (4 et 117).

Namespace pid: détail (2)

- Les processus d'un même namespace peuvent être collectivement arrêté/suspendu ou repris.
- Le premier processus du namespace (pid 1) devient le « init » de ce namespace:
  - Devient le père de tous les processus orphelins
  - Une terminaison de ce processus génère un SIGKILL de tous les processus de ce namespace
- Isolation:
  - Ci-dessous Intéragir=effectuer des appels systèmes prenant le pid du processus comme argument (kill, waitpid, setpriority par exemple)
  - Les processus peuvent intéragir avec les processus de leur propre namespace et les processus des namespace enfants.
  - Les processus des autres namespace sont invisibles.
  - Les PID utilisés dans ces appels systèmes sont ceux du namespace du processus appelant.
- Signaux toujours (exception du processus de pid 1):
  - Signal provenant d'un processus du même namespace: seuls les signaux pour lequel le processus init a un handler peut être envoyé au processus init.
  - Signal transmis d'un processus d'un namespace parent: seuls les signaux pour lequel le processus init a un handler peut être envoyé (à init) sauf pour les signaux SIGSTOP et SIGKILL.

Namespace mount

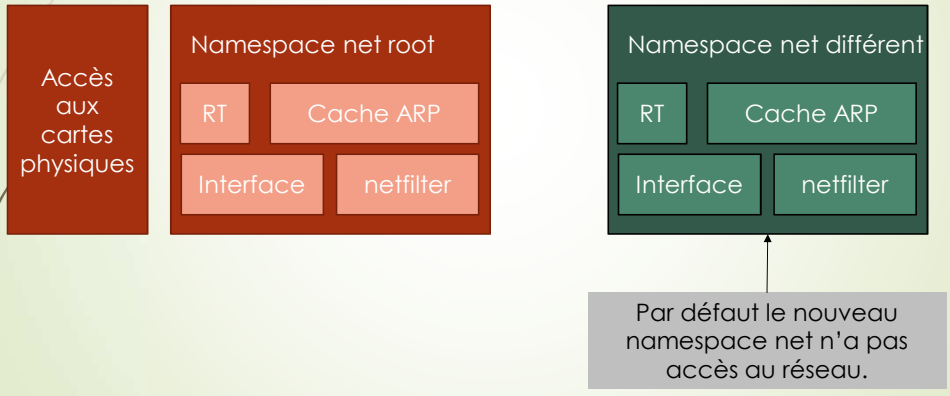
- Isole les points de montages pour chacun des « mount » namespace
- Chaque namespace « mount » a sa liste de points de montage
- Intérêt: monter des parties du système qui seront propre au processus de ce namespace (et pas visible par les autres processus).
  - On peut donc avoir des dossiers qui sont les mêmes que ceux du SE (namespace natif) et d'autres qui sont locales au namespace (suivant ce qui a été monté).



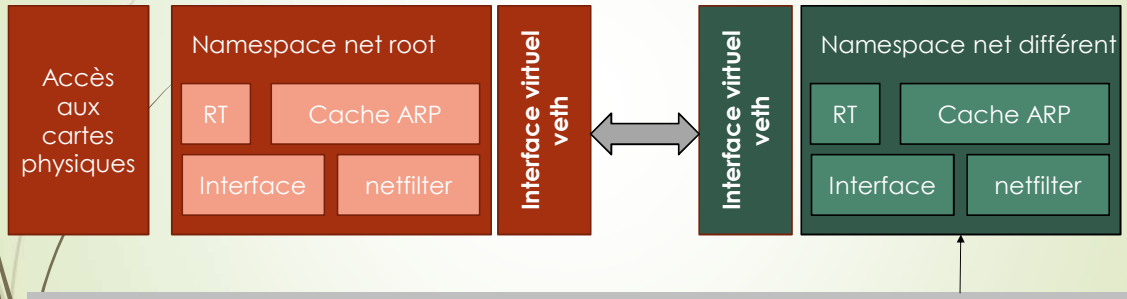
Namespace net

- Permet d'isoler les aspects réseau des processus.
- Pour chaque net namespace, les éléments suivants lui sont propres:
  - interface
  - Une table de routage, une table arp
  - Régles de filtrage (iptable)
  - Etc.
- Les commandes doivent s'appliquer au bon namespace

Namespace net

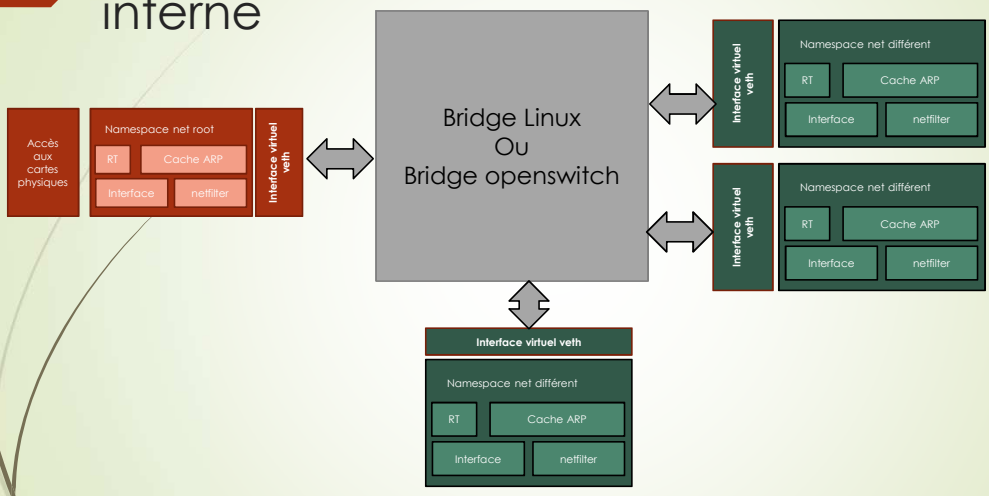


Namespace net: se connecter au namespace root



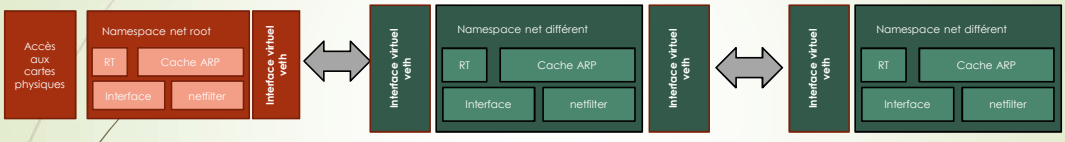
Par défaut le nouveau namespace net n'a pas accès au réseau.
Des interfaces virtuelles (dans chaque namespace) permettent d'interconnecter les namespace net entre eux et potentiellement d'accéder aux interfaces physiques via le namespace root.

Namespace net: utilisation d'un bridge interne



Possibilité de créer un ou des « bridges » internes au système. Ils seront utilisés pour interconnecter les namespace net.

Namespace net: topologie arbitraire



Il est possible de créer des topologies arbitraires avec un jeu d'interfaces virtuelles, éventuellement des bridges, et des routes adaptées.
Les namespaces peuvent alors avoir des rôles différents, de filtrages par exemple.

Les commandes

- lsns: affiche les ns d'un processus
- unshare:
  - associe un namespace à un processus existant
  - créer un processus avec un ou des namespaces différents
- nsenter:
  - permet de taper des commandes associés à certains namespace
    - Changer l'utilisateur
    - Monter des systèmes de fichiers ou périphériques
    - Changer le hotsname
    - Etc.

cgroups (control groups)

- Depuis 2008 (Linux version 2.6.24)
- Bien distinguer les cgroups (ci-dessous) et le namespace cgroup (non vu dans ce cours)
- Limite l'usage des ressources pour un ensemble de processus
  - Priorisation: limite l'usage CPU
  - Mémoire: alloue un volume maximum de mémoire au groupe
  - Comptabilité: compte le temps CPU
  - Isolation: un groupe de processus sera associé à des namespace (déjà fait).
- Actuellement: cgroup version 2

cgroups: fonctionnement

- Les cgroups sont créés/supprimés au travers de dossiers dans le répertoire suivant (pour le cpu ici / cela peut être memory aussi):

  /sys/fs/cgroup/cpu/
- Le cgroup initial incluant tous les processus est le cgroup « / » se trouvant à la racine de /sys/fs/cgroup/cpu
- La création d'un cgroup peut se faire avec mkdir:

  mkdir /sys/fs/cgroup/cpu/**myCgroup**
- Pour ajouter un processus à ce cgroup, il faut ajouter son pid dans le fichier

  /sys/fs/cgroup/cpu/myCgroup/cgroup.procs

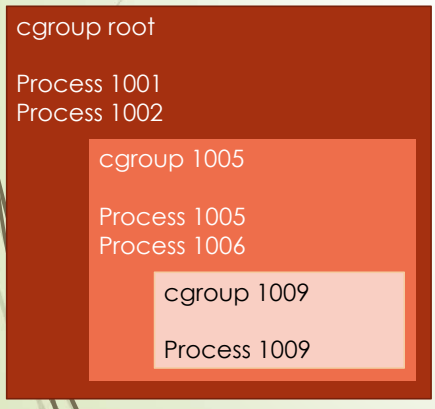
Les fils appartiennent aux cgroups de leur père.

cgroups: création des limites

- CPU:
  - On alloue des ressources sur une base globale de 1024
  - Par exemple: 512 limitera à 50% du CPU
  - Alloué dans le fichier /sys/fs/cgroup/cpu/myCgroup/cpu.shares
- Mémoire:
  - On alloue en nombre d'octets /sys/fs/cgroup/memory/myCgroup/memory.limit_in_bytes

cgroup hierarchy

- Cgroup est un moyen de limiter l'accès aux ressources CPU et mémoires
- Il est possible de créer une hierarchie entre les cgroups.



Hierarchie de processus:

- L'ensemble des proc se partagent les ressouurces du cgroup root.
- Les processus du cgroup 1005 se partagent des ressources plus contraignantes que le cgroup root, etc.

Sécurité: contrôler l'accès aux ressources (appels systèmes)

- Il existe 3 moyens de contrôler les appels systèmes qu'un processus a le droit d'appeler.

Namespace user Root mapping	Capabilities	seccomp
Root au sein de son Namespace.	Autoriser/interdire chaque tâche système.	Autoriser/interdire les tâches systèmes avec la granularité des appels systèmes.

Sécurité: capabilities

- Les capabilities sont associés à un processus donné.
- Ils regroupent les fonctionnalités des appels systèmes.
- Permet de limiter les droits root (pour certains processus)
- Permet d'augmenter les droits de certains processus (non root).
- Les capabilities sont modifiables mais héritées du père.

Liste de toutes les « capabilities »

CAP_SYS_NICE

CAP_SYS_ROOT

CAP_SYS_TIME

…

Processus en cours d'exécution

CAP_SYS_NICE

CAP_SYS_ROOT

CAP_SYS_TIME

…

Capabilities autorisé

Capabilities Non autorisé

Sécurité

- L'appel système seccomp(2):
  - Filtre les appels systèmes qui peuvent être exécutés par un processus
  - Liste blanche
  - Liste noir
  - Une liste préétablie: read, write, exit, etc.

Les fils récupèrent les mêmes filtres que leur père.

Conclusion

Un processus a initié une isolation:

- Montages (mount) isolés
- Isolation des processus fils
- Isolation du réseau
- Hostname propre
- Privilèges limités au namepaces

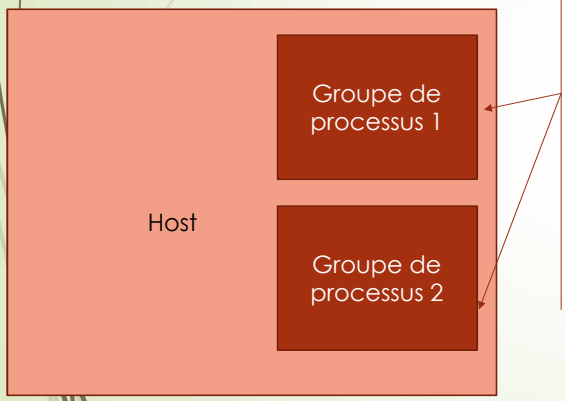
namespaces

- Allocation limitée de CPU et mémoire

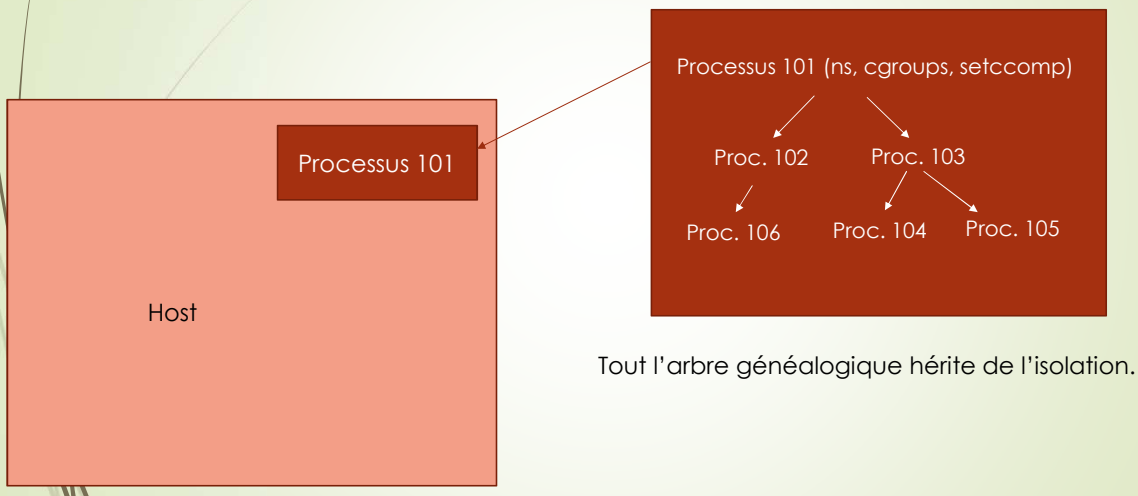
cgroups

- Limitation des appels systèmes possibles

Seccomp capabilities



Conclusion



Tout l'arbre généalogique hérite de l'isolation.

Docker

LP devops

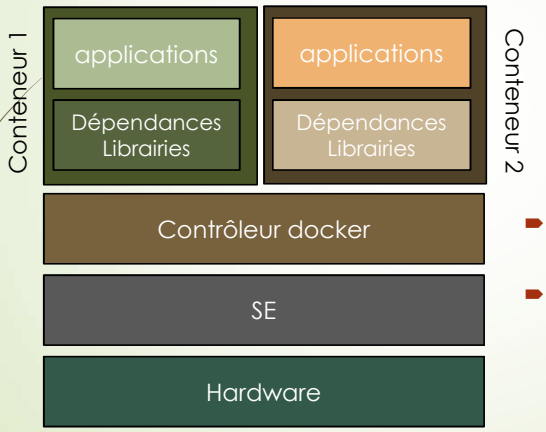
Exécution d'un conteneur

- Un conteneur docker consiste en un processus qui est isolé:
  - namespace
  - cgroups
  - capabilities
  - chroot
- Lancement d'un ou de plusieurs processus
  - La durée de vie du conteneur est celle de ces processus.
  - Le processus init (le premier processus) doit survivre aux autres

Motivation

- Isoler les processus/applications en cours de tests
- Déployer rapidement une application avec tout son environnement
- Isoler des applications qui s'appuient sur des librairies ou binaires en conflits
- S'abstraire des mises à jour systèmes ou logiciels
- Tester un ou plusieurs logiciels dans des environnements différents
- Reproduire facilement un environnement de tests
- Déployer des applications non dépendantes de l'environnement des hosts (sauf le noyau et docker)
- Déployer simplement des applications qui requiert des configurations.

Architecture Docker



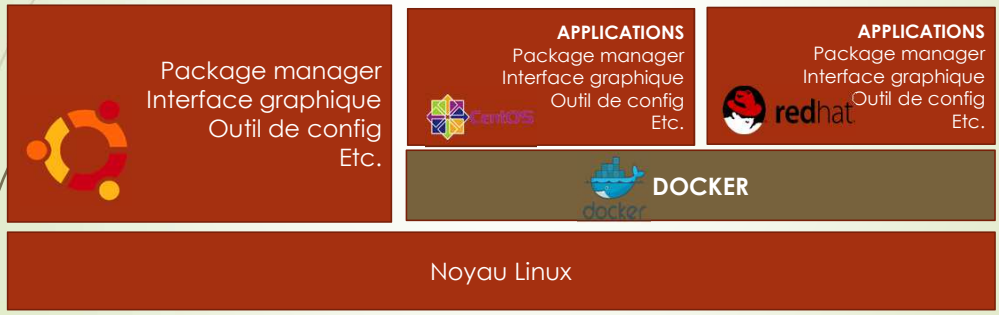
- Toutes les applications peuvent avoir un environnement propre.
- L'intérêt est qu'il y a un seul SE.

Jusqu'où descend on?

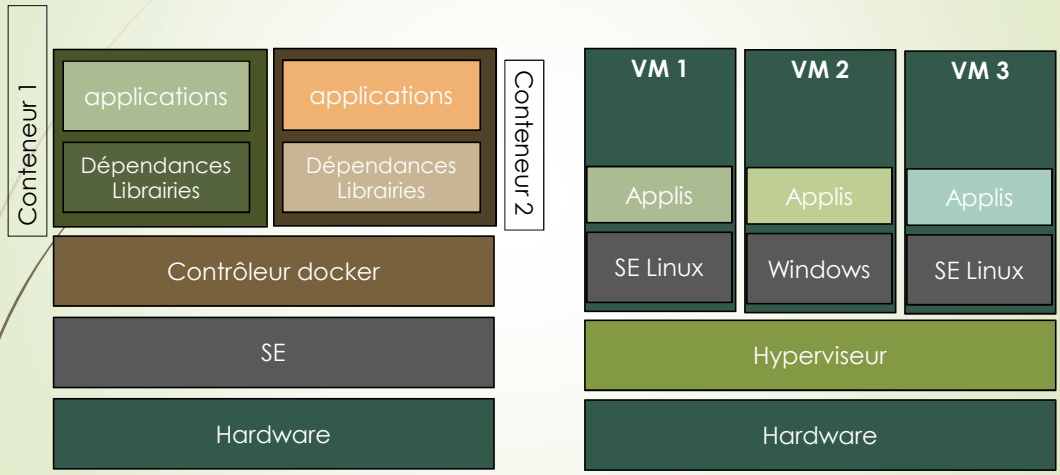
- Préambule:
  - Le noyau Linux est commun à toutes les distributions.
  - C'est les interfaces et les outils de gestion qui diffèrent d'une distribution à l'autre.

Jusqu'où descend on?

- Le noyau est commun à tous les conteneurs
- Docker est installé sur/via une distribution donnée (celle de l'host)
- Les conteneurs peuvent contenir une distribution différente



Différence VM et conteneur



Docker: fonctionnalités

- Offre une interface et une gestion conviviale des conteneurs
- Offre aussi des dépôts publiques de conteneurs pour les applications courante
  - Base de données
  - Serveurs web
  - Etc.
- Possibilité de créer ses propres conteneurs et de les partager.
- Ces conteneurs sont stockés sous formes d'images.

- Deux versions de docker: community et enterprise.

Sécurité / privilèges

- Les conteneurs peuvent être lancés avec n'importe quel utilisateur propriétaire:
  - Option --user pour un utilisateur donné
  - Un utilisateur par défaut (spécifié dans l'image)
- L'utilisateur root (souvent utilisateur par défaut) a des « capabilities » limitées
- Possibilité de lancer un conteneur avec les droits root complet (option --priviliged)
  - déconseillé
- UID mapping: optionnel (voir namespace user)

Les commandes

- Voir sur le site docker: command-line reference
  - https://docs.docker.com/engine/reference/commandline/docker/
- Plusieurs catégories de commandes :
  - docker run …: lance un conteneur
  - docker image …: gère les images
  - docker exec …: exécute une commande/processus dans un conteneur (en cours d'exécution)
  - docker inspect …: permet de récupérer des informations sur une image, conteneur, volume, etc.
  - docker pull …: récupère une image
  - docker ps …: liste les images en cours d'exécution ou exécuté, etc.
  - docker build…: créer une nouvelle image
  - …

Normalisation

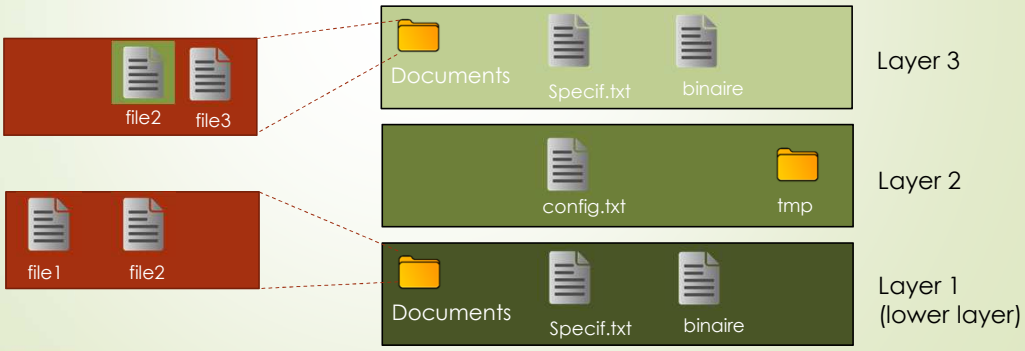
- Normalisation des conteneurs et de leurs fonctionnement
  - OCI: Open Contener Initiative
- Normalisation pour le lancement des conteneurs
  - Runtime: runc sous docker (qui respecte la norme)
- Normalisation du format des images

- Autres gestionnaire de conteneurs:
  - Containerd
  - CRI-O
  - Kata containers

Docker: les images

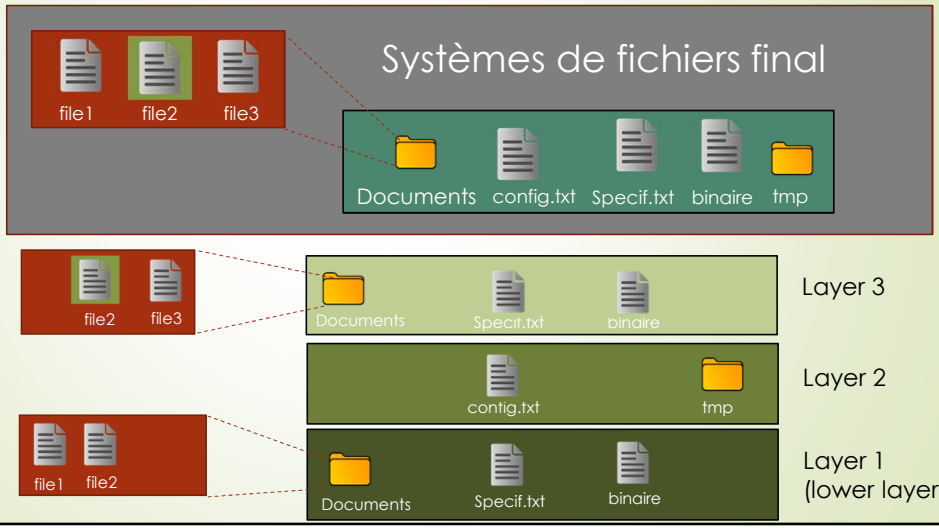
Structure des images: le système de fichiers

- Utilisation du système de fichiers overlay2
  - Anciennement overlay et aufs
  - Ils appartiennent à la famille des Union file system
- Principe:
  - Fonctionnement en couche
  - Possibilité de fusionner des dossiers



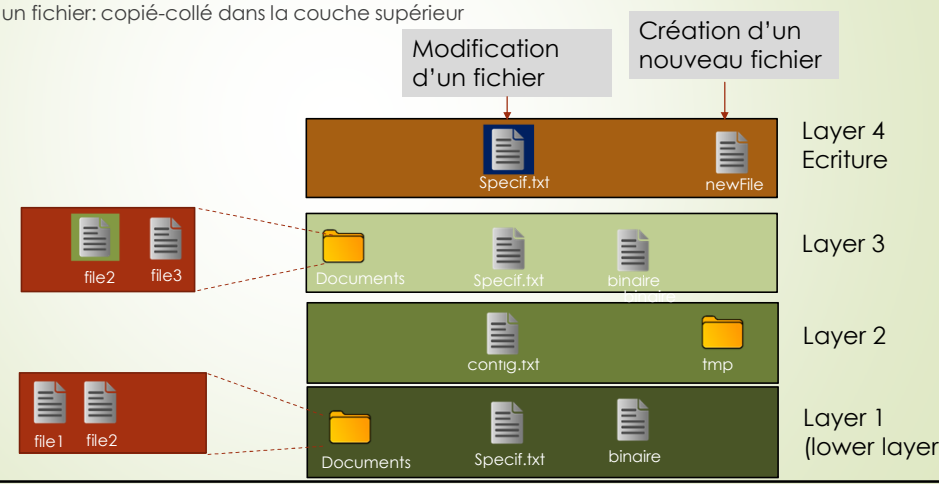
Le système de fichiers overlay2

- **Les couches sont fusionnées:**
  - **En cas de conflits les fichiers de niveaux supérieurs écrasent les fichiers de niveaux inférieurs.**

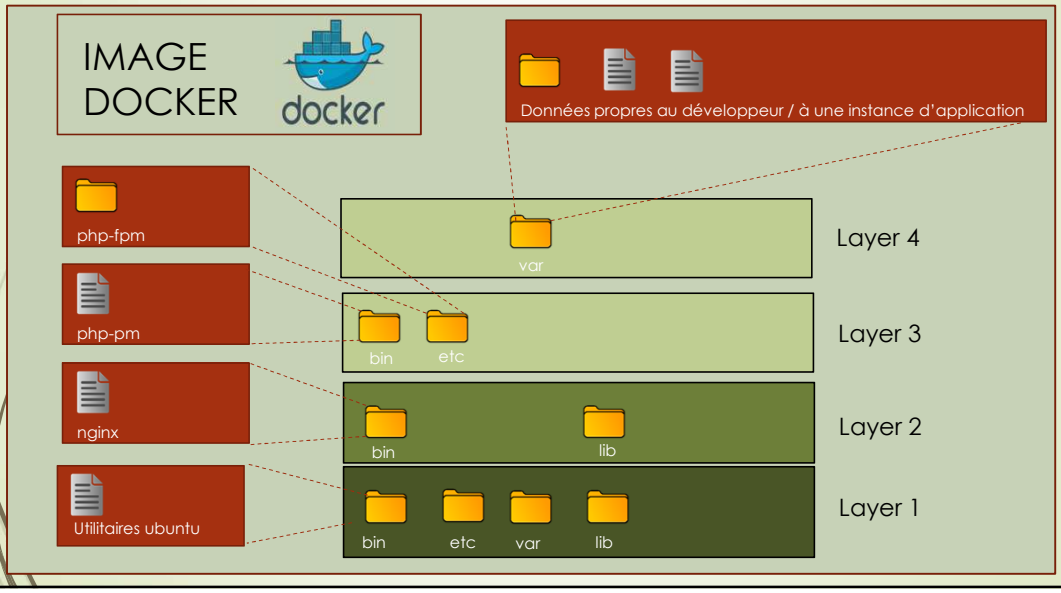


Overlay2: écriture

- Les couches inférieures sont en lecture uniquement
- Les écritures se font dans la couche de plus haut niveau
- Création de fichiers
- Modification d'un fichier: copié-collé dans la couche supérieur

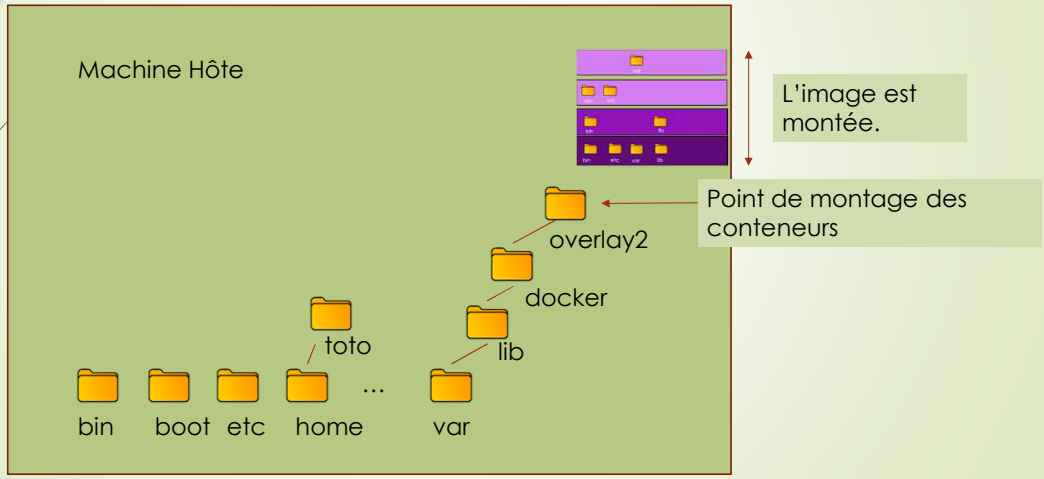


Overlay2 dans docker



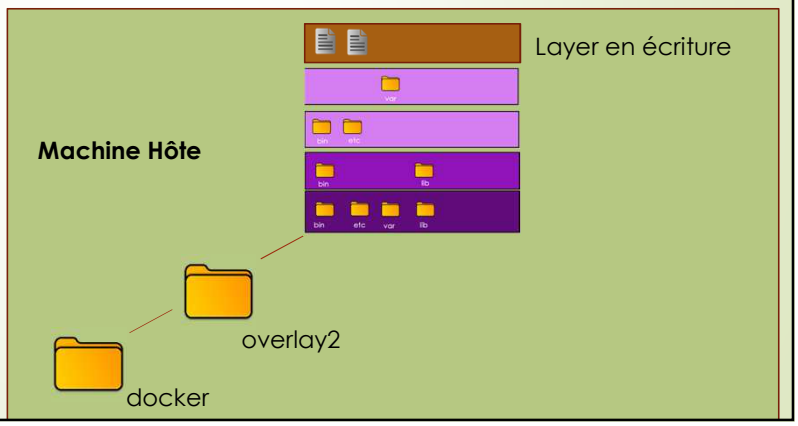
Conteneur en cours d'exécution

- Les images sont montés dans le dossier /var/lib/docker/overlay2
- Elles sont montées avec un système de fichiers overlay2



Conteneur en cours d'exécution

- Une couche en écriture est créer lors du montage qui va héberger les changement par rapport à l'image.
- L'image n'est pas sauvegardée/modifiée à la fin de la vie du conteneur.
- Les modifications ne sont pas pris en compte lors du `umount`.

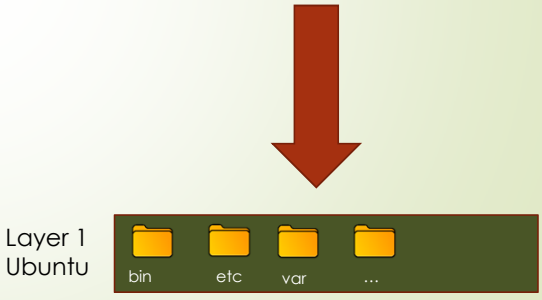


Le dockerfile: FROM

Dockerfile

```
FROM Ubuntu
```

Image de base
Le conteneur va être chrooté
Nécessité d'installer l'ensemble des binaires, configuration, et librairies nécessaire à l'exécution des programmes.



Le dockerfile: RUN

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
```

Commande qui va être réalisée lors de la création de l'image. Installe un ensemble de dossiers et de fichiers supplémentaires dans une nouvelle couche.

Le dockerfile: RUN

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
RUN apt-get install software2
```

Chaque instruction RUN ajoute une couche au système de fichiers overlay.

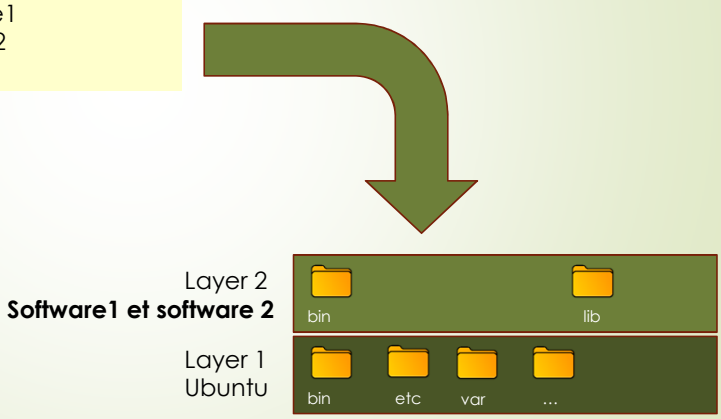
Le dockerfile: RUN

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
&& apt-get install software2
```

Compromis à trouver sur le nombre de couches.



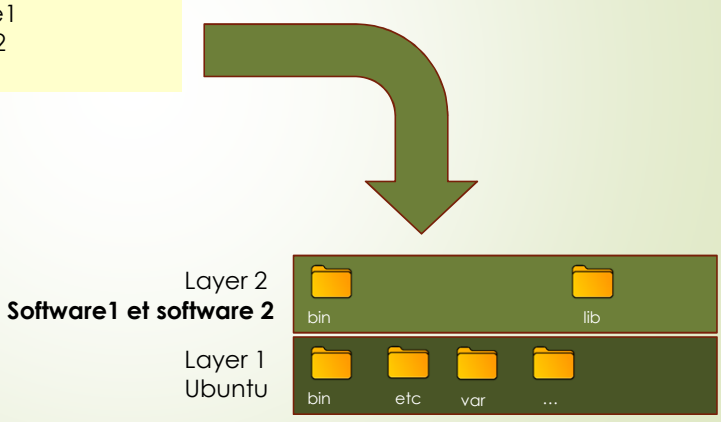
Le dockerfile: RUN

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
&& apt-get install software2
```

Compromis à trouver sur le nombre de couches.



Le dockerfile: COPY et ADD

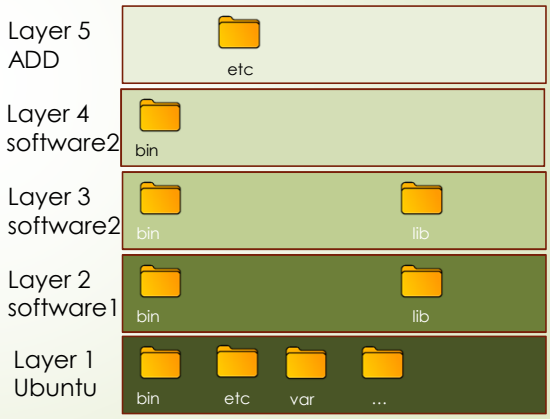
Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
RUN apt-get install software2
COPY  /bin/myCommand  /bin
ADD   http://127.0.0.1/index.html /etc
```

Chaque instruction COPY et ADD ajoute une couche au système de fichiers overlay.

COPY et ADD copie/colle des fichiers ou des dossiers. ADD permet en plus de spécifier des url ou des fichiers .tar comme source.



Le dockerfile: WORKDIR / EXPOSE

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
RUN apt-get install software2
COPY  /bin/myCommand  /bin
ADD   http://127.0.0.1/index.html /etc

WORKDIR dossier
COPY file1  /bin

EXPOSE   80
VOLUME  /var/web
```

Change de répertoire. Utile pour l'instruction qui suit.
Ici, on entre dans dossier puis on copie le fichier file1.

Indique le port à l'écoute.

Partage un dossier avec le système de fichiers hôte.

Le dockerfile: CMD

Dockerfile

```
FROM Ubuntu

RUN apt-get install software1
RUN apt-get install software2
COPY  /bin/myCommand  /bin
ADD   http://127.0.0.1/index.html /etc

WORKDIR dossier
COPY file1  /bin

EXPOSE    80
VOLUME  /var/web


CMD  /bin/software1 arg
```

Les deux seules champs/instructions obligatoires sont:
FROM

CMD ou **ENTRYPOINT**

Commande lancé au démarrage du conteneur.
Ou sous forme de liste:
CMD [« /bin/software1 », « arg1 »]

CMD et ENTRY POINT

```
FROM bash

…

#Utilisation de ENTRYPOINT et CMD


CMD  ls
```

```
bash#docker run myLs
-rw-rw-r-- 1 busson busson 53 déc.   2 08:55 fichier1.txt
-rw-rw-r-- 1 busson busson 53 déc.   2 08:55 fichier2.txt
…
```

Sans argument docker lance CMD:

ls

CMD et ENTRY POINT

FROM bash

…

#Utilisation de ENTRYPOINT et CMD

CMD ls

```
bash#docker run myLs echo toto
toto
```

Les arguments de docker run remplace CMD:

echo toto

CMD et ENTRY POINT

FROM bash

…

#Utilisation de ENTRYPOINT et CMD

ENTRYPOINT [« ls », »-l »]
CMD [« fichier1.txt »]

Conseil: utilisé des listes.

```
bash#docker run myLs
-rw-rw-r-- 1 busson busson 53 déc.  2 08:55 fichier1.txt
```

Sans argument docker lance ENTRYPOINT+CMD:

ls –l fichier1.txt

CMD et ENTRY POINT

FROM bash

…

#Utilisation de ENTRYPOINT et CMD

ENTRYPOINT [« ls », »-l »]
CMD [« fichier1.txt »]

```
bash#docker run myLs fichier2.txt
-rw-rw-r-- 1 busson busson 53 déc.  2 08:55 fichier2.txt
```

Les arguments remplacent CMD et docker lance ENTRYPOINT+CMD:

ls –l fichier2.txt

CMD et ENTRY POINT

FROM bash

…

#Utilisation de ENTRYPOINT et CMD

ENTRYPOINT [« ls », »-l »]
CMD [« fichier1.txt »]

```
bash#docker run --entrypoint=« date » myLs
mer. 02 déc. 2020 09:18:34 PST
```

Docker lance ENTRYPOINT, CMD est vide:

date

CMD et ENTRY POINT

```
FROM bash

…

#Utilisation de ENTRYPOINT et CMD



ENTRYPOINT [« ls », »-l »]
CMD  [« fichier1.txt »]
```

```
bash#docker run --entrypoint=« echo » myLs toto titi
toto titi
```

Docker remplace ENTRYPOINT par « echo » puis CMD par « toto titi » et lance ENTRYPOINT+CMD:

echo toto titi

Conteneur light

- Possibilité de créer une image partant de rien
- Permet d'avoir une image/conteneur léger
- Nécessité de copier-coller les binaires et librairies nécessaires dans /lib, /bin, etc.

```
FROM scratch

ADD ….
COPY ….

CMD  myBin
```

Divers Les autres conteneurs / divers

- Composer des conteneurs
  - Docker compose
- Orchestrateur
  - Docker-Swarm
  - Kubernets
  - Openshift