Docker의 기초 컨셉

Container가 사용하는 기반 기술: namespace와 cgroup

ESukmean(이석민) · 2025-02

esukmean@esukmean.com
https://blog.esukmean.com

목차

목차 1

Container 기술 2

격리를 위해 사용되는 namespace 2

리소스 제한을 위해 사용되는 Cgroup 4

Container 이미지 4

우려되는 부분 5

Java 같이 이식성이 높은가? 6

Container 기술

가상 머신은 하이퍼바이저(Hypervisor)를 통해 하나의 물리적 서버에서 여러 운영 체제를 실행할 수 있게 하는 기술이다. 예를 들어, 물리 서버에 Windows Server가 설치되어 있더라도 VMware나 VirtualBox를 통해 별도의 Linux나 다른 OS를 설치하고 실행할 수 있다.

반면 도커는 호스트 운영 체제 위에서 격리된 실행 환경인 컨테이너(Container)를 제공한다. 컨테이너는 호스트 운영 체제를 공유하지만, 각 컨테이너는 독립적인 프로세스로 작동하며, 필요한 라이브러리와 설정을 자체적으로 포함한다.

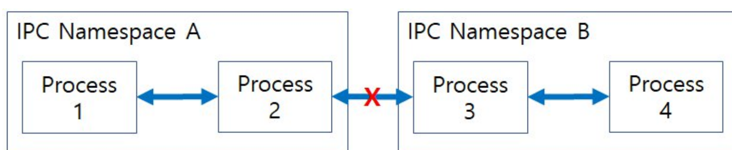
컨테이너는 호스트 OS의 커널을 공유한다.

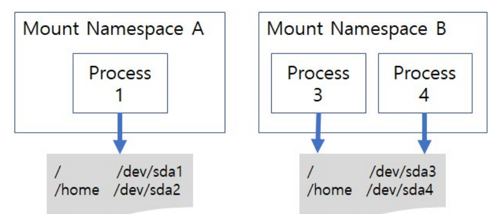
쉽게 생각하자면, 옵션을 통해 네트워크 및 파일시스템을 격리한 상태로 프로그램을 실행 하는 방식으로 이해할 수 있다.

비교 항목	VM	도커(컨테이너)
가상화 방식	하드웨어 수준의 가상화	운영체제 수준의 가상화
운영 체제	각 VM에 별도의 운영체제 설치	호스트 운영 체제를 공유
격리 수준	운영 체제 자체가 격리됨	프로세스 수준으로 격리됨 (cgroup, chroot 개념)
리소스 소비	가상 OS를 돌리는 오버헤드 발생	일반 OS에서 프로그램을 돌리는 것과 동일
시작 속도	부팅 시간 발생 (무거움)	매우 빠른 실행
앱 배포	OS 이미지 단위로 배포	도커 이미지로 배포
복구 및 백업	OS 전체 백업 필요	컨테이너 이미지만 백업
보안	OS가 다른 수준으로 격리	동일한 커널 내에서 설정으로 격리

격리를 위해 사용되는 namespace

"컨테이너"라는 단어에서 환경이 독립되어 있음을 엿 볼 수 있다.
독립된 환경은 커널의 namespace 기능을 통해서 보장된다.
namespace를 사용하면 같은 OS 속에서 다른 Mount정보, 네트워크 정보를 가지게 하거나, 서로간 통신이 안되게 할 수 있다.





각 namespace 끼리는 프로세스, 파일 시스템, 심지어는 사용자 정보(uid, gid)가 보이지 않는다. 이 기능덕에 프로세스는 독립된 환경에서 동작하며, 다른 프로세스에 간섭을 하거나 당할 수 없다.

NAMESPACE 항목	격리 대상	설명
PID	프로세스 ID	각 namespace에서는 독립된 프로세스 트리를 가짐. 다른 namespace의 프로세스 정보를 볼 수 없음
NET	네트워크 인터페이스	namespace별로 독립된 네트워크 인터페이스를 가짐 (IP 주소 대역, 가상 인터페이스등을 부여할 수 있음)
MNT	파일 시스템 마운트	namespace 별로 독립된 파일시스템 구조를 가질 수 있음.
UTS	호스트 이름	각 namespace마다 독립된 hostname을 가질 수 있음
IPC	프로세스간 통신	본인의 namespace 내에서만 IPC 작업이 가능함
USER	사용자 및 권한	namespace 내에서 독립적으로 사용자 권한을 가질 수 있음.
TIME	시간 정보	Namespace 내에서 독립적인 시간 정보(offset)을 가질 수 있음. (Namespace내에서 별도로 NTP 작업 가능)

Linux의 대표적인 namespace 목록 (현 시점에 13개가 있다)

Docker, Kubernetis, Containerd 모두 Namespace를 활용하여 논리적 환경 독립을 제공해 주는 프로그램일 뿐이다. namespace로 독립된 환경속에 네트워크를 붙여주고 마운트 해주는 기능을 해 줄 뿐이다. 다만에, 각 프로그램의 지원 기능에 따라서 더 복잡하고 덜 복잡하다는 차이가 있다.

```
esukmean@esukmean-mint:~$ lsns
        NS TYPE   NPROCS   PID USER     COMMAND
4026531834 time      108  3103 esukmean /usr/bin/pipewire
4026531835 cgroup    108  3103 esukmean /usr/bin/pipewire
4026531836 pid       109  3103 esukmean /usr/bin/pipewire
4026531837 user       91  3103 esukmean /usr/bin/pipewire
4026531838 uts       108  3103 esukmean /usr/bin/pipewire
4026531839 ipc        91  3103 esukmean /usr/bin/pipewire
4026531840 net        91  3103 esukmean /usr/bin/pipewire
4026531841 mnt       108  3103 esukmean /usr/bin/pipewire
4026532821 user        1 43020 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026532822 ipc         1 43020 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026532823 user        1 43618 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026532824 ipc         1 43618 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026532825 user        1 43044 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026532826 ipc         1 43044 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026532827 net         1 43044 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026532966 user        1 43054 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026532967 net         1 43054 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026533022 user        1 43109 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026533023 ipc         1 43109 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026533024 net         1 43109 esukmean /usr/lib/firefox/firefox-bin -contentproc -isForBrowser -prefsHandle 0 -
4026533079 user        1 43172 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026533080 ipc         1 43172 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
4026533081 net         1 43172 esukmean /usr/lib/firefox/firefox-bin -contentproc -parentBuildID 20250130195129
```

필자의 노트북에서 lsns를 실행한 모습이다.
Firefox가 namespace를 활용해 시스템을 보호하기 위해 프로세스 격리를 하고있다.

리소스 제한을 위해 사용되는 Cgroup

namespace를 활용하면 동일한 커널을 사용하면서도 독립된 환경을 만들 수 있다는 것을 알게 되었다. 그러나 Container를 돌리기 위해서는 namespace뿐 아니라 Cgroup도 있다.

Namespace속에서 실행되고 있는 특정 프로세스가 CPU를 100%를 점유하거나, 메모리를 과다하게 차지하고 있을 수 있다. 이 경우, 다른 프로세스에 문제를 야기할 수 있다.

이것을 방지하기 위해서 CGroup이라는것이 추가적으로 개입한다. cgroup은 특정 프로세스 트리의 최대 CPU, Memory, DISK I/O, 네트워크 사용량 등등을 제한할 수 있다.

각 프로세스가 사용 가능한 최대 리소스량을 제한하여 특정 프로그램이 전체 시스템에 영향을 미치는 것을 방지할 수 있다.

Docker를 비롯한 컨테이너 시스템에도 Cgroup이 적용되어 있다. Cgroup을 설정하여 각 컨테이너의 자원 사용을 제한 할 수 있다.

cgroup을 관리하는 주체는 하나만 있는것이 권장된다. 시스템이 cgroup을 통해 시스템의 주요 구성요소에 OOM(Out-of-Memory) 또는 CPU 기아 문제가 발생하지 않도록 관리하기 때문이다.

최근의 리눅스 OS에서는 systemd가 cgroup을 관리한다. 설정만 한다면 특정 서비스의 CPU의 점유율, 메모리 최대 크기 등등을 제한할 수도 있다.

최대 메모리 크기는 절대값이지만, CPU 사용량은 상대값이다. 그러므로, CPU 종류가 다르다면 사용가능한 CPU 능력이 달라질 수도 있다.

Cgroup 관리 주체가 두개일 경우를 생각해보자.

시스템 구성요소인 프로세스 A, 일반 프로그램인 B,C,D가 있다. 4GB가 설치된 시스템에서 안전을 위해 1GB를 A에 주고, B, C, D 에는 총 3GB 가 넘지 않도록 제한했다.
이때, 또 다른 Cgroup 관리 주체가 D 프로세스에 3GB를 별도로 제한하면 이론적으로 6GB가 할당 될 수 있다.

이 경우 시스템 구성요소인 A에 메모리 부족으로 안정성에 문제가 발생할 수 있다.

Container 이미지

위에서 컨테이너는 namespace와 cgroup을 통해서 동작하며, Docker, Kubernetes등은 이것들을 제어해 주는 프로그램임을 설명했다.

앞서 namespace에는 파일 시스템 격리가 가능하다고 했다. 이것을 활용하면 루트 디렉터리까지 제어할 수 있다. 파일 시스템 격리로 호스트 OS의 커널을 공유하면서 컨테이너 내에서는 다른 배포판을 동작할 수도 있다.

즉, 레드햇 OS에서 Ubuntu 이미지, 또는 우분투 OS에서 Redhat 이미지를 돌릴 수도 있다. 이것으로 바깥의 호스트 OS가 달라져도 컨테이너 안쪽에서는 동일한 동작 환경을 보장받을 수 있다.

커널이 달라져도 syscall, interupt 0x80, sysenter 등의 커널 함수 호출 방법은 변하지 않는다.
그러므로 커널이 바뀌거나, 심지어도 배포 이미지가 달라져도 문제 없이 동작한다.

상상이 잘 되지 않는다면 커널 업데이트를 해도 왠만한 프로그램은 문제없이 동작 함을 생각해 보자.

그러나 이 점을 다시 생각해 보면, 각각의 컨테이너는 각자의 Root 디렉토리를 가져야 한다는 것이 꺼림직하다. 파일 시스템이 격리됐기 때문에, Root 디렉토리 및 프로그램 실행에 필요한 모든 라이브러리·파일을 알아서 가지고 있어야 한다.

이 파일들을 하나로 뭉친것이 “컨테이너 이미지”가 된다. Docker 등의 컨테이너 프로그램들은 이미지를 namespace 내에 마운트한다. 그리고 마운트된 파일 내에서 필요한 프로그램을 실행한다.

우려되는 부분

컨테이너를 돌리기 위해서는 컨테이너 이미지가 필요하다. 그리고 해당 이미지에는 OS가 도는데 필요한 모든 파일들이 들어가 있어야 한다. 그러면 자연스레 걱정이 생긴다

- 각 컨테이너 이미지에 OS 파일이 들어가면 필요한 용량이 너무 커지는것이 아닌가?

이것은 두가지 방법으로 해결되어 있다

1. OS에 꼭 필요한 파일만을 담으면 용량이 그리 크지 않다.
2. OverlayFS를 활용해 이미지간 공통 부분을 하나로 저장한다. (공통 부분을 중복 저장하지 않는다)

우선 첫번째 부분을 보자. Ubuntu를 서버용으로 설치하면 수백MB의 디스크 용량이 사용된다. 이것은 시스템 운영에 필요한 모든 프로그램이 탑재되어 있기 때문이다. 하지만 Docker에는 모든 것을 담을 필요는 없다. 시스템 동작에 반드시 필요한 것만 넣고, 그 외는 apt install로 직접 설치하게 유도하면 된다.

이렇게 만들어진 이미지가 ubuntu-slim이다. 우분투 22.04 버전을 기준으로 77MB 용량만을 차지한다.

두번째 방법은 OverlayFS를 이용하는 방법이다. OverlayFS는 여러개의 파일 시스템(파티션, 디렉토리)을 뭉쳐서 하나로 볼 수 있게 하는 파일 시스템이다.

컨테이너 이미지는 실제로는 여러개의 이미지가 겹쳐져 있는 구조이다. 기본적으로 OS 이미지가 아래 깔려있다. 그리고 그 위에 Java, Python, PHP 와 같은 필수 프로그램 이미지가 있을 것이다. 맨 위에 .jar, .py, .php 파일들이 있다.

이때, 공통적으로 사용된 이미지는 중복 필요없이 “하나”만 존재하면 된다. OS 이미지 하나를 여러개가 공유할 수 있는 것이다. 수동으로 빌드한 이미지가 또한 여러 컨테이너에서 공유할 수 있다.

공유하는 부분이 많아진다면 결국 “완전히 다른” 부분만이 용량을 차지하게 된다. 최종적으로는 .jar 파일, .py, .php 파일 등등만을 가지면 된다. (공유하는 부분은 하나만 존재하면 되기 때문에)

이미지 하나는 어딘가 저장되어어야 한다. 2개째 부터 중복되는 이미지에 한해서만 용량 절감의 효과가 있다.

다만, 공유의 기준은 “이미지 단위”이다. 그러므로, 같은 이미지를 많이 사용할 수록 도움이 된다. 전략적으로 사용하는 이미지를 제한하여 총 이미지 용량을 줄일 수 있다.

Java 같이 이식성이 높은가?

Container는 최소한의 리소스만으로도 격리된 환경을 구성할 수 있다는데 장점이 있다. 격리된 환경을 통해 어디서나 동작한다는 이식성(portability)를 확보할 수 있다.

그러나, 기본적으로 호스트 OS의 커널을 사용하는 점과 명령어셋 변환은 없다는 점을 간과해서는 안된다. Docker 이미지를 생성해도 CPU 종류가 달라지면 실행이 불가하다.

실제로 Arm64용으로 빌드된 이미지를 x86에, 또는 x86용 이미지를 Arm64에 쓰려고 하면 exec format error 오류를 보게 된다.

동일한 CPU 명령어셋을 사용하더라도 시스템콜의 ID가 다르다면 작동되지 않는다. 이것은 어쨌든 커널 자체는 호스트OS의 것을 사용하기 때문이다. 그러므로, 아예 다른 커널을 사용한다면 작동이 안된다.

동일한 종류의 커널 내에서는 하위 호환성 지원 덕분에 문제 없이 동작한다.
(예: Linux 4.6 - 5.6)

그러므로 Java와 같이 모든 시스템에서 동작한다고 생각하면 안된다. 대신, 같은 종류의 커널에 같은 CPU 종류를 사용하는 시스템 사이에서 이식성이 생긴다.

그럼에도 Java 대신 Conatiner를 쓰는 이유는 다음과 같다:

- 각 컨테이너 사이에 별개의 파일구조 분리가 가능하다.
  (동일한 OS에서 여러개의 MySQL를 동시에 돌릴 수 있다)
- 네트워크 분리 덕에 동일 포트를 쓰는 프로그램을 같이 실행가능하다.
- 네이티브 앱을 돌리므로 성능저하가 없다.
- 커널 및 CPU 종류만 일치시키면 OS를 넘기는 수준의 이식성이 보장된다.
  (물리서버 장애시에도 빠르게 다른 서버에서 서비스를 재개할 수 있다.)