표준 리눅스 101

소개

랩

교육목적

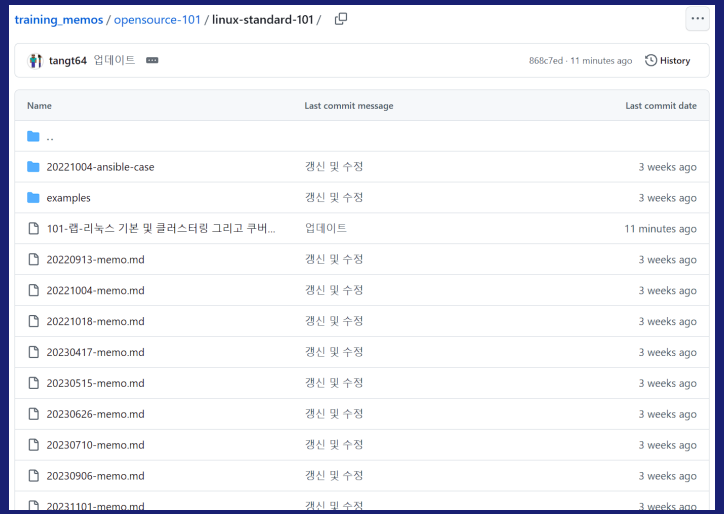
- 리눅스 시스템에 대한 소개 및 기술 동향 파악
- 랩 기반으로 표준 리눅스 시스템을 직접 구성
- 새로운 서비스 기반으로 간단한 리눅스 시스템 구성
- 자동화를 통한 리눅스 시스템 표준화 구성 및 배포 방법

교육자료 위치

아래 위치에서 마크다운 및 PDF확인이 가능합니다.

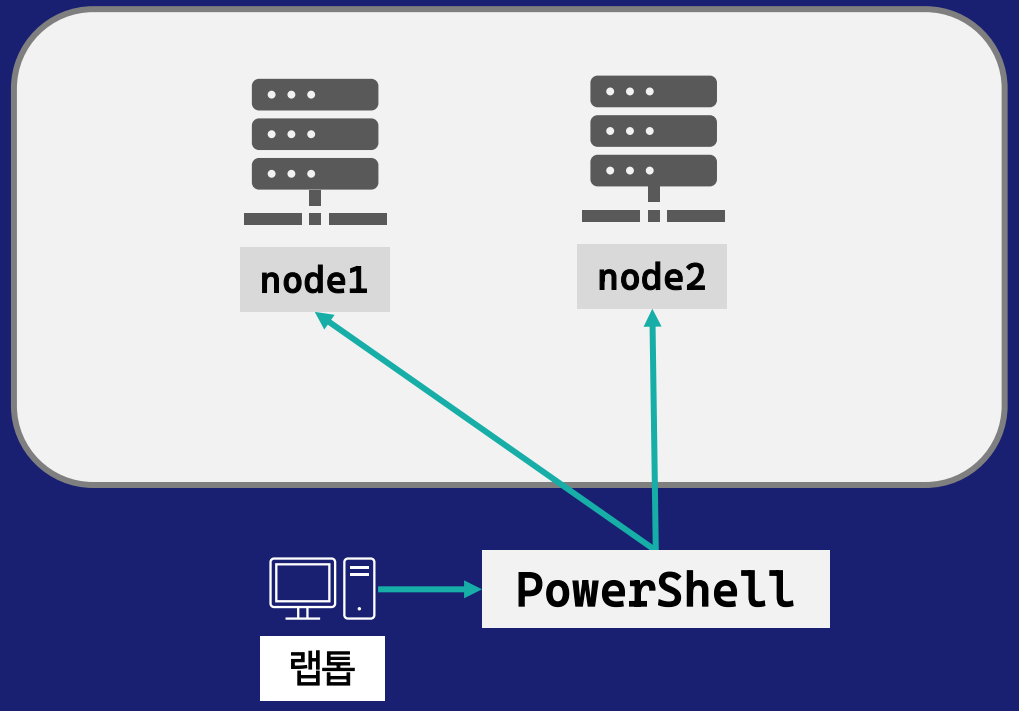
https://github.com/tangt64/training_memos/blob/main/opensource-101/linux-standard-101

마크다운은 오늘 날짜로 접근 하시면 됩니다. PDF파일은 종종 변경 됩니다.



기본과정 랩

총 2대의 가상머신을 이용하여 랩을 수행한다. 리눅스 배포판은 Redhat, SuSE 계열 및 데비안 계열 기반으로 사용이 가능하다. 랩 진행은 보통 로키 리눅스 기반으로 진행한다.



목차

1. 오픈소스 리눅스 어드민 기본
2. 오픈소스
3. 변경 및 추가된 기본 명령어
4. 리눅스 커널/쉘
5. systemd
6. systemd 서브 시스템

목차

7. 시스템
8. 디스크 구성
    - vdo
    - lvm2
    - stratis
9. 네트워크 설명
10. 패키지 관리 방법
11. 가상머신 및 컨테이너 구현 도구
12. 자동화

랩

가상머신 두 대 기반으로 진행을 권장 합니다. 한 대만 사용하셔도 크게 문제가 없습니다. 하이퍼브이 사용이 어려운 경우, 버추얼 박스와 같은 다른 도구로 사용하셔 됩니다.

리눅스 배포판

- 가급적이면 **CentOS-9-Steam, Rocky 9**으로 권장 합니다.
- 우분투 및 수세 리눅스를 사용 하셔도 상관 없습니다. 다만 몇몇 명령어 및 패키지는 호환이 되지 않을 수 있습니다.
- **가상머신은 한대만 사용하여도 괜찮습니다.**

가상 머신 사양

vCPU: 2개

vMEM: 4096

vDISK:

```
NODE1{O/S(30GiB), DATA(100GiB)}
NODE2{O/S(30GiB), DATA(100GiB)}
```

오픈소스

리눅스와 오픈소스 관계

약간의 동향 이야기

오픈소스

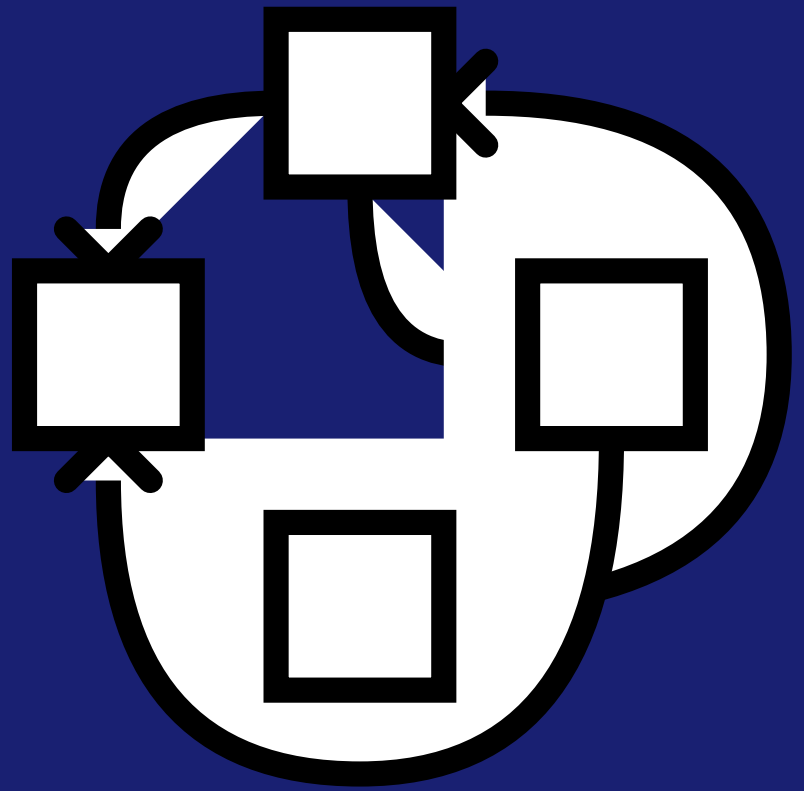
오픈소스는 리차드 스톨만이 GNU도구 및 라이선스를 선언하였음. 리눅스 토발즈는 GNU도구 중 리눅스 커널(Linux Kernel)를 만들었다. 리차드 스톨만은 Hurd라는 커널을 만들기도 하였음.

선진적인 디자인의 **마이크로 커널 디자인(Micro Kernel)**를 사용하여서, 당시 성능으로는 구현 및 사용하기가 어려운 부분이 많았음.

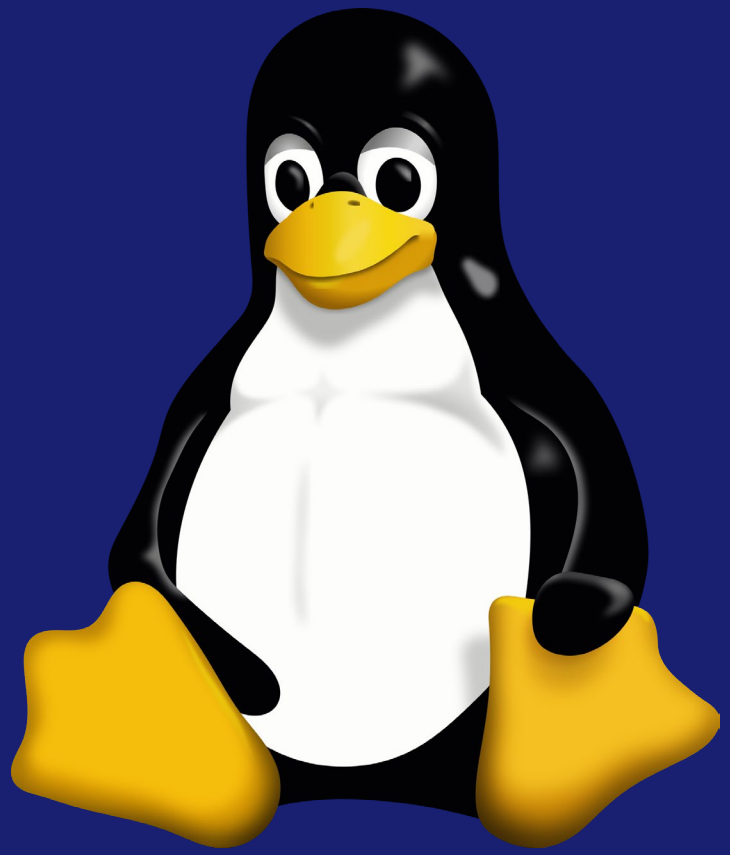
- GNU/Linux Kernel
- GNU/Hurd Kernel

현재 허드 커널을 사용하는 배포판은 데비안 리눅스 배포판 밖에 없다.

허드커널



리눅스 커널



GNU/LINUX 라이선스

리눅스의 대다수 소프트웨어는 GNU기반의 소프트웨어를 사용하고 있다.

GNU와 Unix소프트웨어와 큰 차이점은 없지만, 라이선스 및 커뮤니티 기반의 생태계를 좀 더 강하게 가지고 있다. 오픈소스에서 많이 비교되는 대상인 BSD소프트웨어와 제일 큰 차이점은 BSD는 유닉스 코드 기반으로 구성이 되어 있다.

리눅스는 리눅스 토발즈가 만든 리눅스 커널을 지칭을 하며, 리눅스 커널은 GNU소프트웨어와 같이 사용하기 때문에 보통 GNU/Linux라고 부른다.

리눅스 커널 뒷 이야기

리눅스 커널은 현재 안드로이드 및 IoT그리고 엔터프라이즈 시스템에 많이 사용하고 있다.

한때, 리눅스 커널은 애플사의 대표인 스티브 잡스를 통해서, Mac OS의 커널로 사용이 될 뻔 하였으나, 애플에서 비공개라는 조건으로 토발즈에게 요구(?) 하였고, 이에 대해서 **리누즈 토발즈**는 거절 하였다. (Just Fun에 해당 내용이 나와 있음)

OS X가 사용하는 커널은 4.4BSD Lite-2기반으로 구성이 되어 있다. 이 프로젝트에서 파생된 BSD, MIT라이선스 기반으로 작성된 프로그램 다윈(Darwin)으로 공개가 되고 있다. 현재 BSD에서 사용하는 커널은 XNU라고 부르고 있으며, 마이크로 커널 기반의 하이브리드 커널 구조를 가지고 있다.

루머링크: 쿼라링크

오픈소스 라이선스

현재 리눅스는 다음과 아래와 같이 소프트웨어 구성이 되어 있다. 배포판를 만드는 회사는 혹은 단체마다 허용하는 범위가 많이 다르지만, 대다수 상용 리눅스 배포판은 라이선스에 매우 민감하다. 특히, 레드햇 및 수세 리눅스는 매우 민감하게 라이선스 범주를 다룬다.

BSD Software

버클리 소프트웨어 라이선스. 대다수 유닉스 소프트웨어는 BSD라이선스 기반으로 작성이 되어 있으며, GNU소프트웨어의 대다수는 라이선스 및 이상으로 인하여 다시 GNU라이선스 기반으로 재작성이 되었다.

오픈소스 라이선스

GNU Software(Version 2 and 3)

GNU소프트웨어는 상용 소프트웨어 및 기존 유닉스 라이선스의 단점, 예를 들어 기여코드도 상용 라이선스에 내포가 되는 불합리 및 불공정한 부분을 없애기 위해서 만든 라이선스. 현재 리눅스의 대다수 라이선스 GNU기반으로 구성이 되어 있다.

MIT Software

MIT라이선스는 MIT대학교에서 만든 라이선스이다. MIT에서 배포되는 소프트웨어 및 기술은 MIT라이선스로 구성이 되어 있으며, GNU라이선스와 호환은 된다. MIT라이선스는 BSD라이선스와 매우 흡사한 구조를 가지고 있으며 둘 다 공통점은 소스코드 오픈에 대한 강제성이 없다.

LGPL Software

최대한 압축해서 이야기 하면 " GPL라이선스 라이브러리에 상용 애플리케이션 링크 허용"정도이다. 본래 GPL라이선스는 매우 엄격하기 때문에 라이선스에 따르지 않는 바이너리에 대해서 링크 및 허용하지 않는다. 하지만 LGPL기반의 라이선스 제품에는 상용 제품에 대해서도 링크를 허용한다. 이전에는 GPL3에서는 LGPL에 대해서 허용하지 않았는데, 현재는 GPL3에서도 허용한다.

최근 레드햇 라이선스 이슈

현재 레드햇은 다음과 같은 라이선스 정책을 오픈소스 커뮤니티에 알렸음.

- SRPM에 대한 재배포는 레드햇 서비스 라이선스에 위반
- 개인 사용 및 소스코드에 대해서는 문제가 없으나(GPL 2/3), 레드햇에서 제공한 패키지 기반으로 리-빌드 하는 경우, 앞으로 레드햇 서비스 사용 금지
- Rocky, Alma, Oracle 리눅스 같은 레드햇 클론 버전은 직접적으로 SRPM복제가 불가능

Check: Unauthorized Use of Subscription Services section

https://www.redhat.com/licenses/Appendix_1_Global_English_20230309.pdf

Partner will not use Red Hat Products or Services to create an offering competitive with Red Hat, directly or indirectly,

https://www.redhat.com/licenses/Partner_Agreement_Webversion_North_America_English_20220308.pdf

최근 레드햇 라이선스 이슈 결론?

1. 앞으로 100% RHEL클론 버전은 어렵다.
2. 대다수 레드햇 계열은 **ABI/kABI호환성** 위주로 개발 진행이 될 듯 하다.
3. **수세/데비안/알마/우분투 리눅스** 배포판으로 다양하게 사용할 예정.

재설치 없는 배포판(커뮤니티 → 상용)

- **레드햇 센트 스트림** → 레드햇 엔터프라이즈(제한적으로)
- **우분투** → 우분투 프로페셔널
- **오픈수세** → 수세 엔터프라이즈
- 알마 리눅스
- 데비안 리눅스

정리

1. RHEL경우에는 SRPM를 개인 혹은 내부적으로 사용하는 용도 이외에는, 배포 용도로 사용이 불가능한다.
2. 레드햇은 라이선스 정책이 다른 배포판보다 보수적인 편이다.
3. 데비안 계열(우분투 포함)은 다양한 저장소를 제공하기 때문에, 애플케이션 확장이 편하지만, 라이선스 및 패키지 품질에 레드햇보다 느슨하다.

에디터

나노/빔

에디터

'VI/VIM'사용이 어려운 경우 "nano"에디터를 사용한다. **나노(nano)**에디터는 현재 대다수 배포판에서 표준 에디터로 사용하고 있다. systemd에서는 기본 에디터로 사용하고 있으며, 대다수 애플리케이션도 현재는 "vi/vim"보다 "나노"에디터 사용을 권장한다.

```
# dnf install nano -y
# export EDITOR=nano
# echo $EDITOR
```

나노 에디터 사용

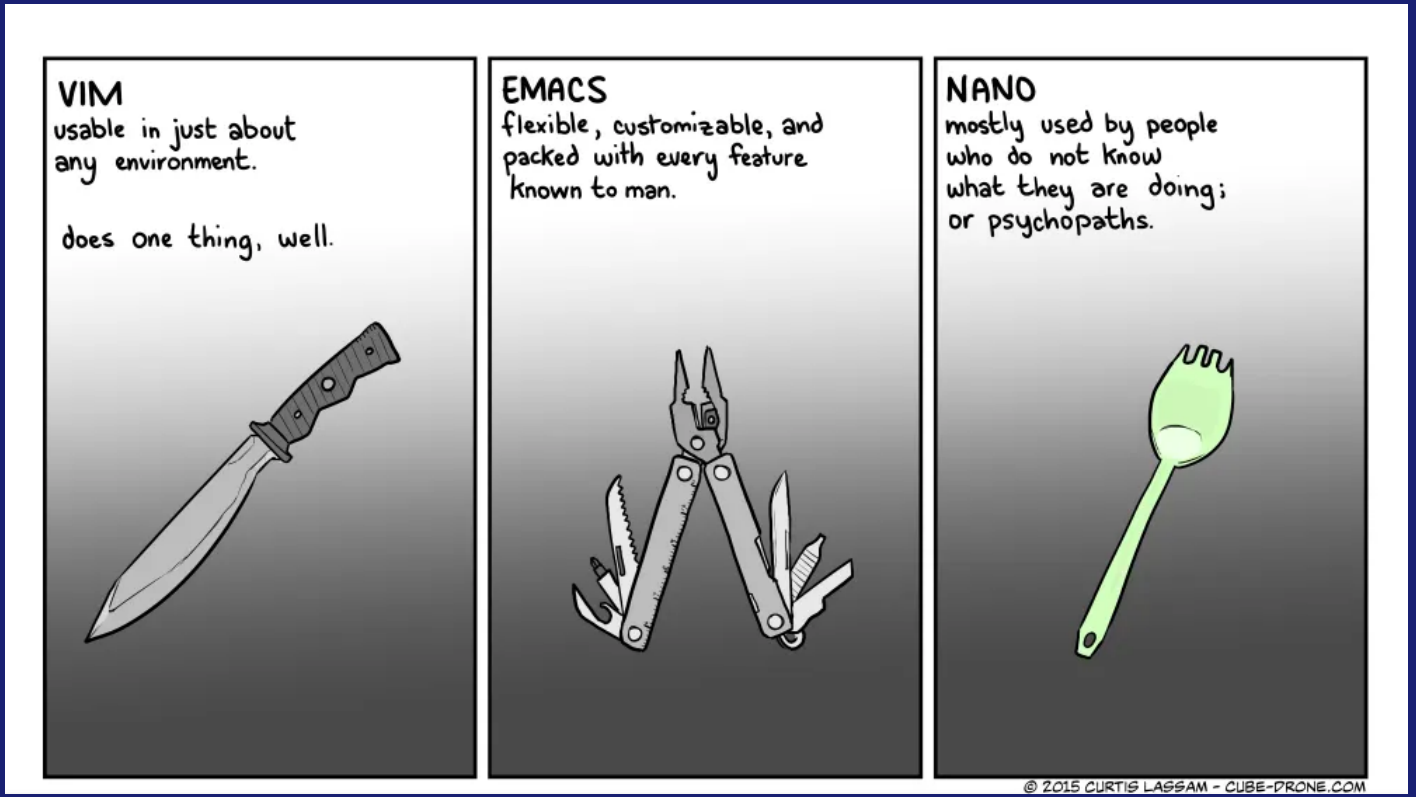
기본기능

Ctrl+A: 맨 처음으로 이동하기

Ctrl+E: 맨 끝으로 이동하기

Ctrl+Y: 스크롤 업(한 화면)

Ctrl+V: 스크롤 다운(한화면)



나노 에디터 사용

편집

`Ctrl+K`: 잘라내기

`Ctrl+U`: 붙여넣기(버퍼 포함)

`Ctrl+O`: 이름확인 후 저장

`Ctrl+X`: 에디터 종료

검색 및 문자열 변경

`Ctrl+W`: 문자 계속 검색

`Ctrl+Q`: 문자 뒤로 검색

`Alt+R`: 문자 치환

"Y", "V"는 TTY터미널이 아닌 경우 올바르게 동작이 되지 않을 수 있습니다.

나노 에디터 확장

나노 에디터의 기능 확장을 원하는 경우, 아래 링크를 참조한다.

https://github.com/scopatz/nanorc

```
$ dnf install nano wget unzip -y
$ curl https://raw.githubusercontent.com/scopatz/nanorc/master/install.sh |
sh
```

ALE+NeoVIM

VI/VIM를 사용하는 경우에는 기존의 "vi/vim"보다는 ALE기반으로 사용을 권장한다. VIM의 최대 단점은 비주얼 스튜디오 코드나 혹은 서브라임 에디터처럼 편의 기능이 많이 부족하다.

이러한 부분을 해결하기 위해서 "ALE(Asynchronous Lint Engine : Lint while you type in Vim)" 설치를 권장한다. 확장 패키지에 대한 정보는 아래에서 확인이 가능하다.

https://www.vim.org/scripts/script.php?script_id=5449

https://github.com/dense-analysis/ale

편하게 설치를 하기 위해서, 아래 링크를 참고한다.

https://webinstall.dev/vim-ale/

```
$ sudo dnf install git curl -y
$ curl -sS https://webi.sh/vim-ale | sh
```

COMMAND

기본 명령어

기본 명령어

"freedesktop"와 같은 프로젝트가 시작이 되면서 리눅스 구조는 이전과 많이 달라졌다. 하지만, 여전히 기본적인 명령어는 똑같이 사용하지만, 관리용도로 사용하는 명령어가 많이 변경이 되었다.

변경된 기본 명령어는 다음과 같다. 이 부분은 아래 **시스템 블록**에서 좀 더 다루도록 한다.

```
systemctl
hostnamectl
journalctl
networkctl
```

그 이외 시스템 운영 시 아래와 같은 명령어 학습을 권장한다.

lsof

특정 프로그램이 어떤 파일을 사용 중인지 확인한다. 보통 의심이 되는 프로그램에 대해서 해당 명령어로 확인한다. 또한, 링크가 깨진 바이너리에 대해서 검사 시 사용한다.

```
# lsof -u root | head -5
COMMAND     PID USER   FD      TYPE             DEVICE SIZE/OFF       NODE NAME
systemd       1 root  cwd       DIR              253,0      235        128 /
systemd       1 root  rtd       DIR              253,0      235        128 /
systemd       1 root  txt       REG              253,0   102120   67259583 /usr/lib/systemd/systemd
systemd       1 root  mem       REG              253,0   581925     806262
/etc/selinux/targeted/contexts/files/file_contexts.bin

# lsof -i TCP:22 | head -5
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd      746 root    3u  IPv4  19170      0t0  TCP *:ssh (LISTEN)
sshd      746 root    4u  IPv6  19172      0t0  TCP *:ssh (LISTEN)
sshd    32133 root    4u  IPv4  70864      0t0  TCP node1.mshome.net:ssh->tang-laptop.mshome.net:57469
(ESTABLISHED)
sshd    32183 root    4u  IPv4  70864      0t0
```

lsof

```
# lsof -nP -iTCP -sTCP:LISTEN
COMMAND PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
httpd   873   root    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
sshd    874   root    3u  IPv4  20728      0t0  TCP *:22 (LISTEN)
sshd    874   root    4u  IPv6  20737      0t0  TCP *:22 (LISTEN)
httpd   913 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
httpd   914 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
httpd   996 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)

# lsof -nP -i:80
COMMAND PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
httpd   873   root    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
httpd   913 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
httpd   914 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
httpd   996 apache    4u  IPv6  20773      0t0  TCP *:80 (LISTEN)
```

sar

시스템 모니터링 시, 'ps', 'top'와 같은 명령어를 많이 사용하지만, 자세하게 시스템의 내용을 모니터링을 원하는 경우 "sar"기반으로 확인을 권장한다. "sar"는 "sysstat"에 포함이 되어 있으며, 이 패키지에는 다음과 같은 명령어들이 함께 포함이 되어 있다.

프로그램	설명
sar	시스템의 모든 부분에 대해서 모니터링 및 조회가 가능합니다.
sadc	“system activity data collector”의 약자. 이를 통해서 sar에서 사용한 데이터가 추출이 된다.
sa1/2	"sar1/2"는 이전에는 "crond"를 통해서 수집이 되었지만, 지금은 systemd의 "timer", "sleep", "service"를 통해서 수집 및 동작한다.
iostat	CPU 및 I/O 기록에 대해서 수집 및 기록한다.
mpstat	CPU상태에 대해서 출력한다.
pidstat	특정 PID에 대해서 상태 정보를 출력한다.
nfsiostat	NFS ¾에 대해서 성능 기록을 출력한다.
cifsiostat	Samba에 대해서 성능 기록을 출력한다.

sar

CPU 사용 상태를 확인 시, 다음과 같은 명령어로 수행합니다.

```
# sar -u 2 5
Linux 5.14.0-362.8.1.el9_3.x86_64 (node1.example.com)   03/31/24        _x86_64_        (2 CPU)

22:41:07        CPU     %user     %nice   %system   %iowait    %steal     %idle
22:41:09        all      0.00      0.00      0.00      0.00      0.00    100.00
22:41:11        all      0.00      0.00      0.25      0.00      0.00     99.75
22:41:13        all      0.00      0.00      0.50      0.00      0.00     99.50
22:41:15        all      0.00      0.00      0.00      0.00      0.00    100.00
22:41:17        all      0.00      0.00      0.00      0.00      0.00    100.00
Average:        all      0.00      0.00      0.15      0.00      0.00     99.85
```

sar

MEM 사용 상태를 확인 시, 다음과 같은 명령어로 수행합니다.

```
# sar -r 2 5
Linux 5.14.0-362.8.1.el9_3.x86_64 (node1.example.com)   03/31/24        _x86_64_        (2 CPU)

22:42:49    kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit
kbactive   kbinact   kbdirty
22:42:51        66428    117924    121756     28.23        12    171432    313796     11.18
120588     62740
0
22:42:53        66428    117924    121756     28.23        12    171432    313796     11.18
120588     62768
0
22:42:55        66428    117924    121756     28.23        12    171432    313796     11.18
120588     62768
0
```

sar

"FIlesystem"사용 상태를 확인 시, 다음과 같은 명령어로 수행합니다.

```
# sar -F 2 5
22:44:23     MBfsfree   MBfsused   %fsused  %ufsused     Ifree     Iused    %Iused FILESYSTEM
22:44:25        69546       2070      2.89      2.89  36658628     41532      0.11 /dev/mapper/almalinux-root
22:44:25          744        216     22.52     22.52    524266        22      0.00 /dev/sda2
22:44:25          592          7      1.17      1.17         0         0      0.00 /dev/sda1
22:44:25        53945        411      0.76      0.76  27863037         3      0.00 /dev/mapper/almalinux-home
```

mpstat

CPU코어 혹은 소캣에서 사용 상태를 출력한다. 별도로 명령어를 명시하지 않고 실행하면, 실시간으로 사용 정보를 출력한다. "sysstat"명령어에 포함된 명령어이다.

```
# mpstat 1 5
# mpstat -P 1 5
```

nstat

TCP스택 상태에 대해서 확인한다. 일반적으로 TCP 모니터링 시 사용한다.

```
# nstat -z | head -10
IpInReceives                    43                 3.2
IpInAddrErrors                  0                  0.0
IpInDelivers                    38                 2.9
IpOutRequests                   27                 2.0
IpOutNoRoutes                   0                  0.0
IcmpOutMsgs                     0                  0.0
IcmpOutRateLimitHost            0                  0.0
IcmpOutDestUnreachs             0                  0.0
IcmpMsgOutType3                 0                  0.0
```

ltrace

바이너리에서 사용하는 라이브러리를 추적한다. 올바르게 동작하지 않는 프로그램 디버깅 시 사용한다.

```
# ltrace hostname
rindex("hostname", '/')                                                    = nil
strcmp("hostname", "dnsdomainname")                                        = 4
strcmp("hostname", "domainname")                                           = 4
strcmp("hostname", "ypdomainname")                                         = -17
strcmp("hostname", "nisdomainname")                                        = -6
getopt_long(1, 0x7ffd7d610528, "aAdfbF:h?iIsVy", 0x55c5c04d6aa0, nil)      = -1
__errno_location()                                                         =
0x7fa34ca416c0
malloc(128)                                                                =
0x55c5c244c2a0
```

strace

프로그램에서 사용하는 시스템 콜을 확인한다.

```
# strace hostname
execve("/usr/bin/hostname", ["hostname"], 0x7ffe7a5c63f0 /* 32 vars */) = 0
brk(NULL)                               = 0x55da657ba000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffda3a91f70) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
uname({sysname="Linux", nodename="node1.example.com", ...}) = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "node1.example.com\n", 18node1.example.com
)     = 18
exit_group(0)                           = ?
+++ exited with 0 +++
```

ss

호스트에 연결된 세션 상태를 확인한다. 또한, 'ss'명령어로 특정 세션을 종료할 수 있다.

```
# ss src 172.25.20.198
# ss -K dport 80233
# ss -antp
```

ip

호스트에 구성된 네트워크 장치 정보에 대해서 확인한다. 명령어를 통해서 네트워크 장치 설정 및 구성이 가능하지만, 가급적이면 "NetworkManager"로 구성을 권장한다.

'ip'명령어는 뒤에서 더 다룬다.

리눅스 커널/쉘

새로운 커널 기능

커널

커널 아키텍처 및 기능

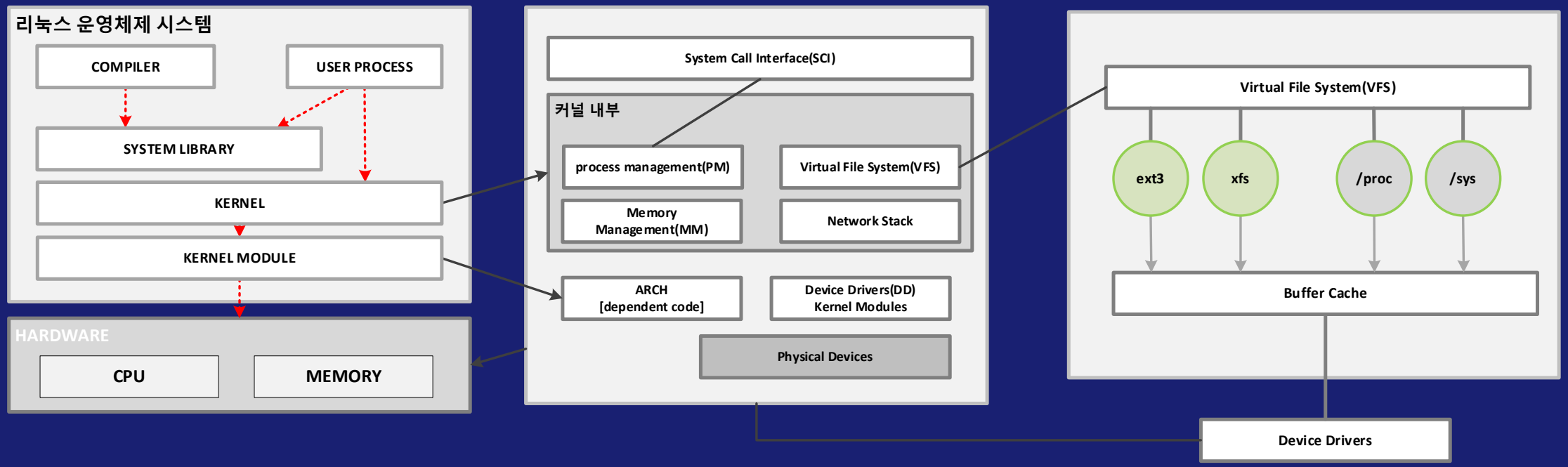
커널 아키텍트

리눅스 커널은 초기에 x86만 지원하였다. 현재 리눅스 커널은 대다수 CPU아키텍처를 지원하고 있다. 현재 대표적인 지원하는 CPU는 다음과 같다.

- Intel
- AMD
- IBM PowerPC
- Sparc
- IBM s390

커널 아키텍트

리눅스 커널은 처음에는 모놀릭(Monolic Kernel) 커널 구조 기반으로 작성이 되었다. 현재는 모듈기능 지원 및 커널 핫 패치(Hot-Patch)를 지원하고 있기 때문에, 모놀릭 커널의 단점을 많이 제거 하였다.



커널 버전 방식

리눅스 커널은 두 가지 형태로 현재 나누어져 있다. 커널 버전은 "https://www.kernel.org"에서 확인이 가능하다.

LT Release

LTS버전은 Long Term Support 버전이다. 해당 버전의 소스코드는 5년에서 최장 10년 이상 지원하기도 하다. 이 커널은 오래된 하드웨어 모듈도 지속적으로 지원하며, 커널 버그 또한, 최소화 되어있기 때문에, 안정적인 시스템 운영에는 적절한 커널이다. 다만, 성능이나 새로운 기능 사용은 어려운 부분이 있다.

Normal Release(stable)

일반 버전은 새로운 기능 추가 혹은 특정 기능이 패치가 되면, 보통 일반 커널 버전으로 릴리즈가 된다. 해당 버전은 x.y.z형태로 계속 y/z패치가 유지가 된다. LT와 다르게 새로운 기능도 추가가 되는 버전이다. 일반적으로 유지 기간은 대략 3년 정도 된다.

Mainline Release

자주 업데이트 되는 커널 릴리즈 버전. 새로운 기능이 많이 추가가 되며, ML버전 기준으로 지속적으로 커널 기능 및 성능 개선이 된다. 보통 3개월 주기로 갱신된다.

커널 버전 방식

w = Kernel Version = 4

xx = Major Revision = 4

y = Minor Revision = 0

zzz = Patch number = 45

mainline:	**6.4-rc7**	2023-06-18	[tarball]		[patch]	[inc. patch]	[view diff]	[browse]
stable:	**6.3.9**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**6.1.35**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**5.15.118**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**5.10.185**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**5.4.248**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**4.19.287**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
longterm:	**4.14.319**	2023-06-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse] [changelog]
linux-next:	**next-20230623**	2023-06-23						[browse]

커널 버전 방식

대다수 리눅스 배포판은 LTS 버전을 채용 혹은 선택하는데, 이것도 각각 기업마다 입장이 조금씩 다르다. 레드햇 경우에 LTS커널을 사용하지 않으며 별도로 LTS버전을 구성 및 운영한다. 최근, 리누즈 토발즈는 리눅스 커널 릴리즈 방식을 변경 하였는데, 이유는 다음과 같다.

"내 마음에 안 들어서"(이것도 루머 입니다..ㅎㅎ)

좀 더 공식적으로 풀어서 이야기 하면, 다음과 같이 발표 하였다.

“since we no longer do version numbers based on features, but based on time, just saying ‘we’re about to start the third decade'”

아래 링크에서 넘버링 변경에 대한 이유를 확인 할 수 있다.

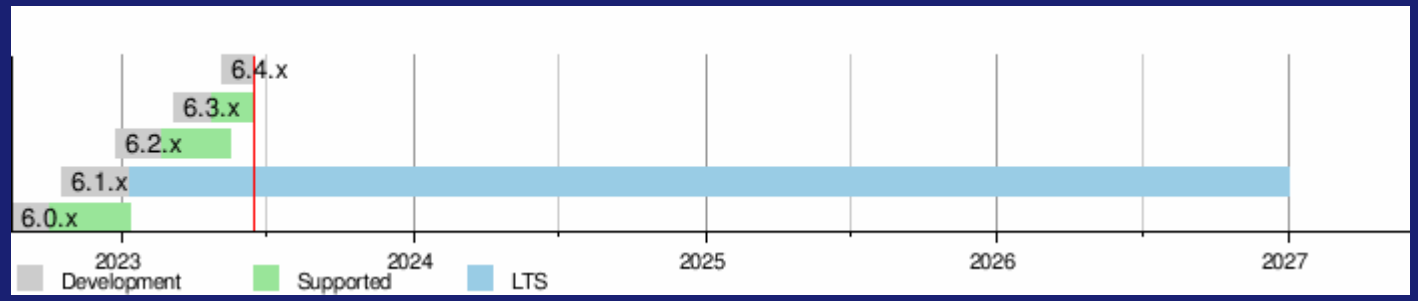
왜 리누즈 토발즈는 넘버링을 변경했는가?

커널 버전 방식

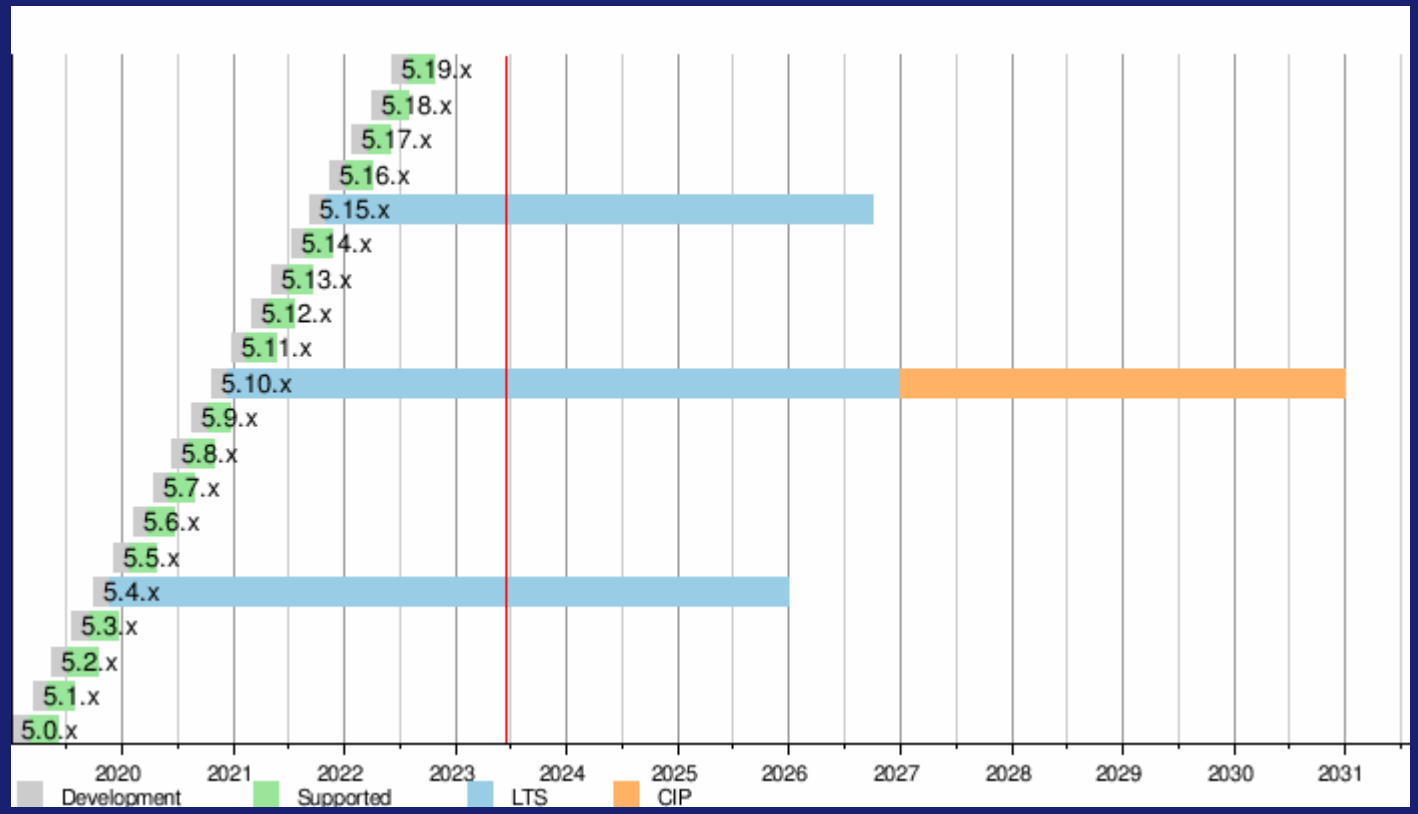
- Linux pre-1.0 versions ran from 1991 to 1994: Three years.
- Linux 1.x ran from 1994 through mid-1996: Two and half years.
- Linux 2.0.x was the mainline stable kernel from mid-1996 through early 1999: Less than three years.
- Linux 2.2.x was the mainline stable kernel from early 1999 through early 2001: About two years.
- Linux 2.4.x was the mainline stable kernel from early 2001 through late 2003: About three years.
- Linux 2.6.x is the mainline stable kernel since mid-December 2003: More than seven years.

결론은 2.x에서 너무 오랜 시간이 걸렸다. 거의 19년 정도 릴리즈 시간이 발생.

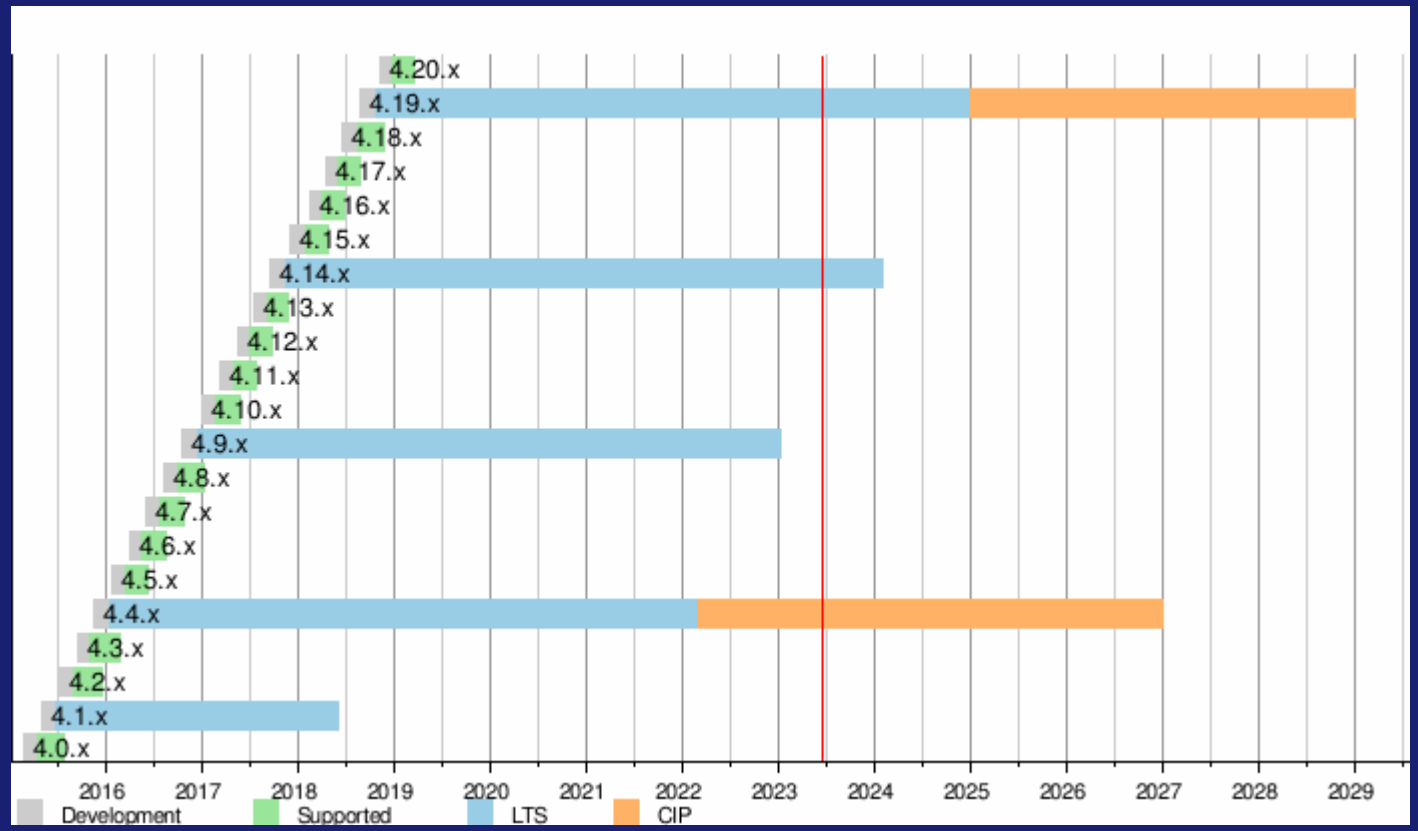
커널 버전 6.x


6.4.x
6.3.x
6.2.x
6.1.x
6.0.x
2023
2024
2025
2026
2027
Development
Supported
LTS

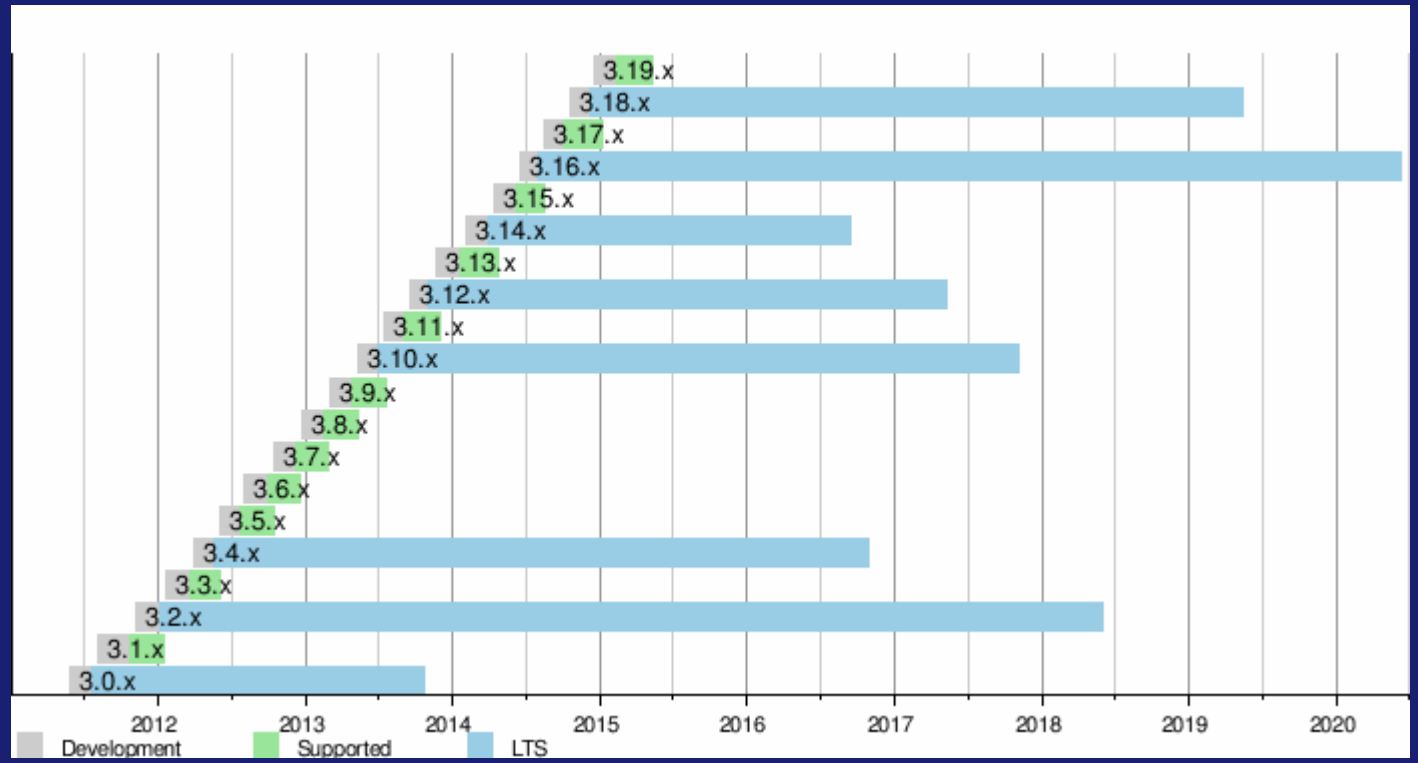
커널 버전 5.x


5.19.x
5.18.x
5.17.x
5.16.x
5.15.x
5.14.x
5.13.x
5.12.x
5.11.x
5.10.x
5.9.x
5.8.x
5.7.x
5.6.x
5.5.x
5.4.x
5.3.x
5.2.x
5.1.x
5.0.x
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
Development
Supported
LTS
CIP

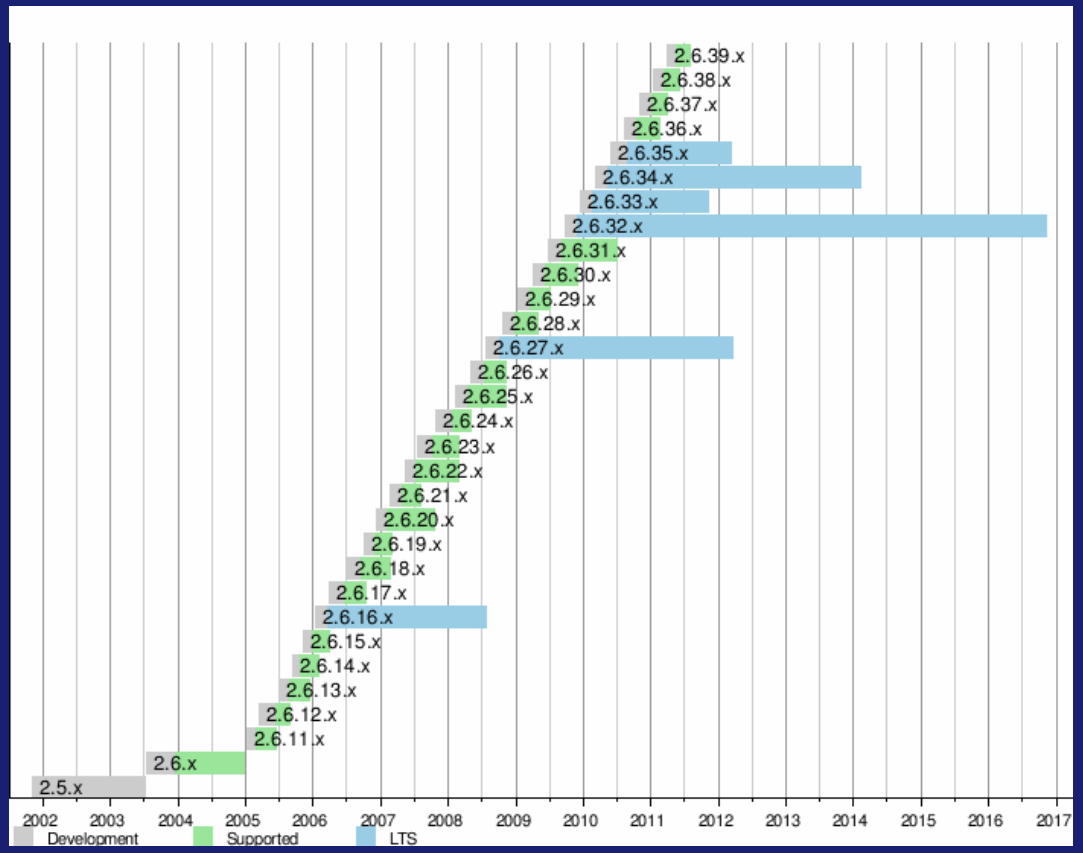
커널 버전 4.x


4.20.x
4.19.x
4.18.x
4.17.x
4.16.x
4.15.x
4.14.x
4.13.x
4.12.x
4.11.x
4.10.x
4.9.x
4.8.x
4.7.x
4.6.x
4.5.x
4.4.x
4.3.x
4.2.x
4.1.x
4.0.x
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
Development
Supported
LTS
CIP

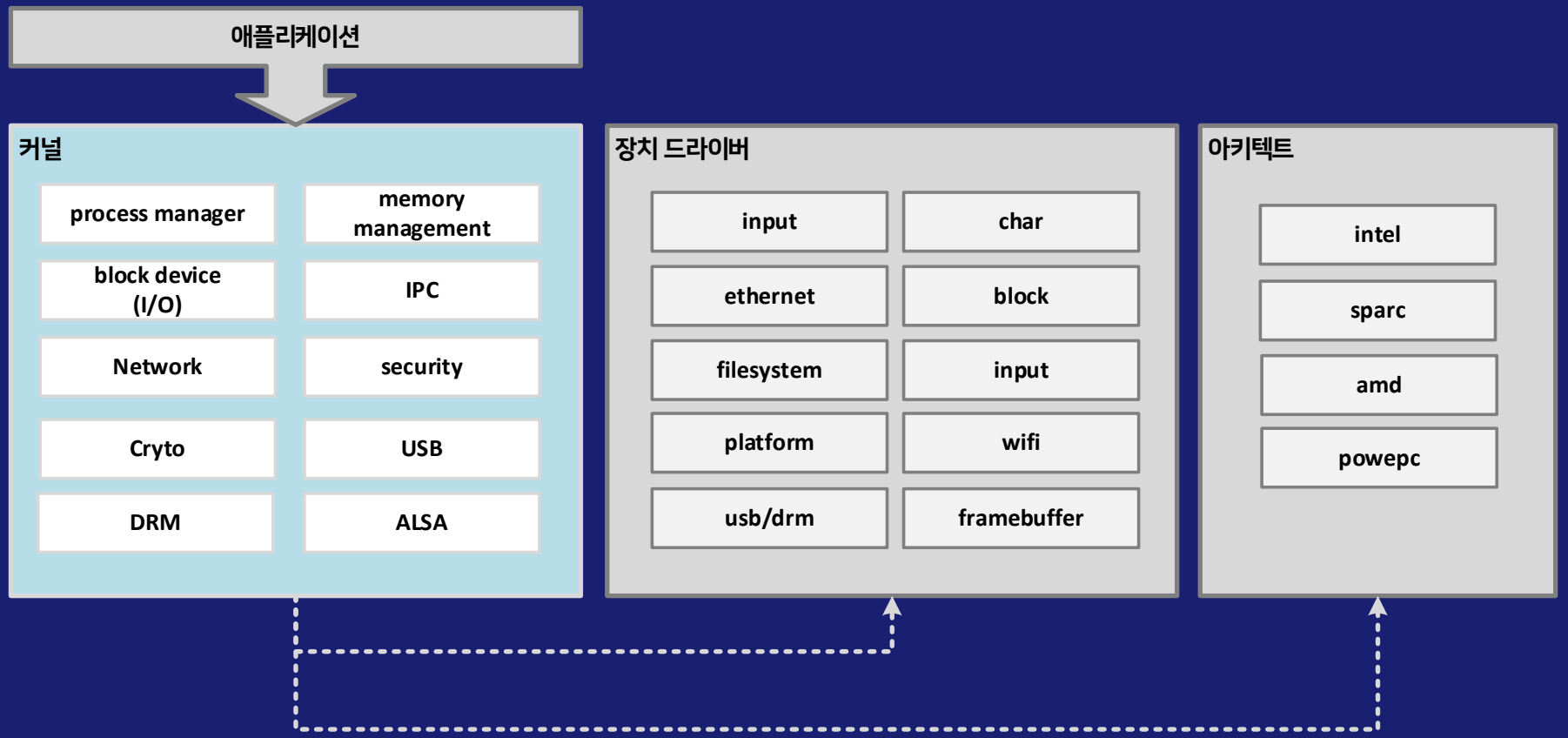
커널 버전 3.x


3.19.x
3.18.x
3.17.x
3.16.x
3.15.x
3.14.x
3.13.x
3.12.x
3.11.x
3.10.x
3.9.x
3.8.x
3.7.x
3.6.x
3.5.x
3.4.x
3.3.x
3.2.x
3.1.x
3.0.x
2012
2013
2014
2015
2016
2017
2018
2019
2020
Development
Supported
LTS

커널 버전 2.x


2.6.39.x
2.6.38.x
2.6.37.x
2.6.36.x
2.6.35.x
2.6.34.x
2.6.33.x
2.6.32.x
2.6.31.x
2.6.30.x
2.6.29.x
2.6.28.x
2.6.27.x
2.6.26.x
2.6.25.x
2.6.24.x
2.6.23.x
2.6.22.x
2.6.21.x
2.6.20.x
2.6.19.x
2.6.18.x
2.6.17.x
2.6.16.x
2.6.15.x
2.6.14.x
2.6.13.x
2.6.12.x
2.6.11.x
2.6.x
2.5.x
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
Development
Supported
LTS

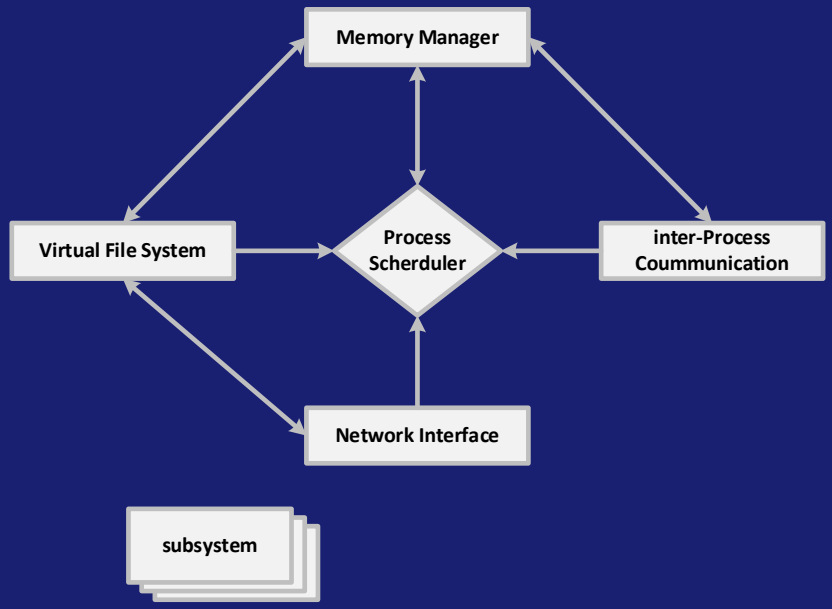
커널 아키텍트


애플리케이션
커널
process manager
memory management
block device (I/O)
IPC
Network
security
Cryto
USB
DRM
ALSA
장치 드라이버
input
char
ethernet
block
filesystem
input
platform
wifi
usb/drm
framebuffer
아키텍트
intel
sparc
amd
powepc

커널 아키텍트

리눅스 커널은 모놀릭(Monolithic Kernel) 커널 구조이다. 마이크로 커널 혹은 하이브리드 커널과 달리, 서브 시스템 기반으로 각 자원을 관리 및 사용한다. 아래 그림은 모놀릭 커널의 구조이며, 핵심 커널 서비스 이외 나머지 영역은 하위 영역으로 별도로 구성이 된다.

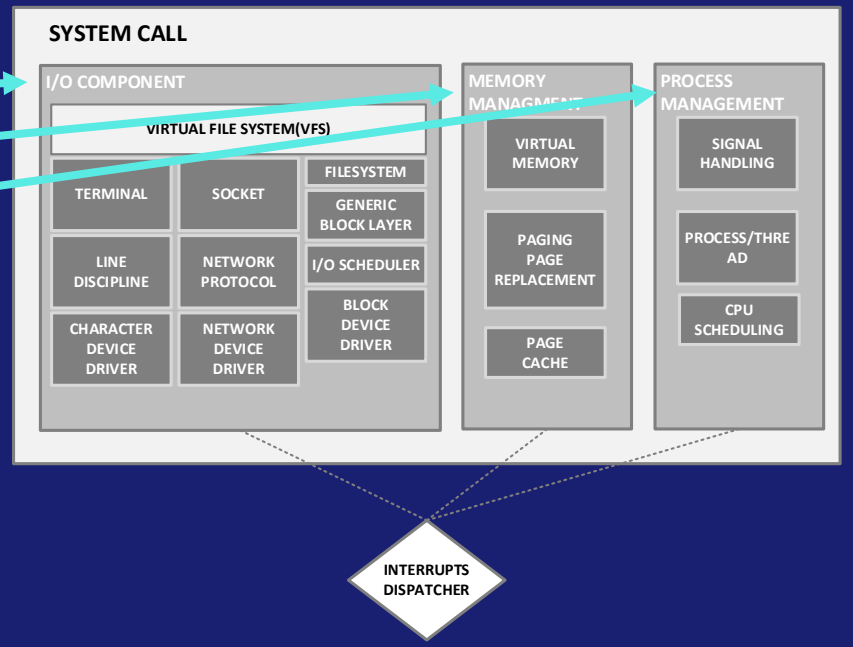
https://docs.kernel.org/subsystem-apis.html



커널 아키텍트

리눅스 시스템 콜은 다음과 같다.

- I/O
- Memory
- Process



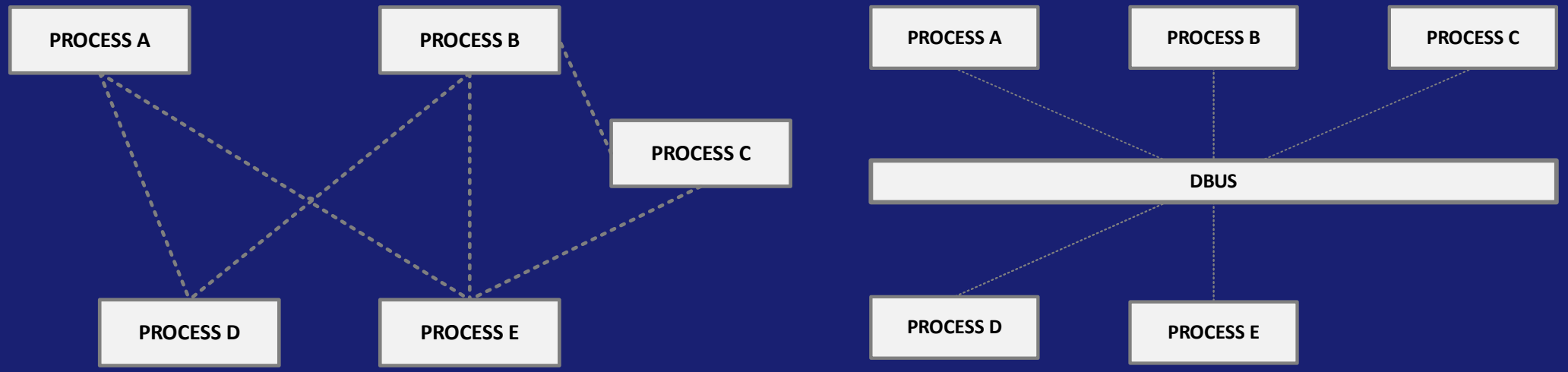
각 서브 시스템은 스케줄러가 있으며, 커널은 스케줄러를 통해서 작업을 할당 및 처리한다. API관련 문서는 아래 링크에서 확인이 가능하다.

https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

dbus

윈도우 시스템 계열은 시스템 컴포넌트는 com/dcom기반으로 구성이 되어 있다. 현재 리눅스도 현대적으로 구조가 변경이 되면서 dbus서비스 중심으로 컴포넌트를 통합하고 있다.

시스템 블록에서 특정 정보가 변경이 되면, 다른 컴포넌트들도 해당 정보를 참조하여 시스템 운영에 반영한다. 대표적으로 윈도우 서버는 네트워크 정보를 "Windows Service Bus, (WSB)"를 통해서 다른 서비스와 공유한다.



dbus

dbus서비스는 다음과 같은 명령어로 조회가 가능하다.

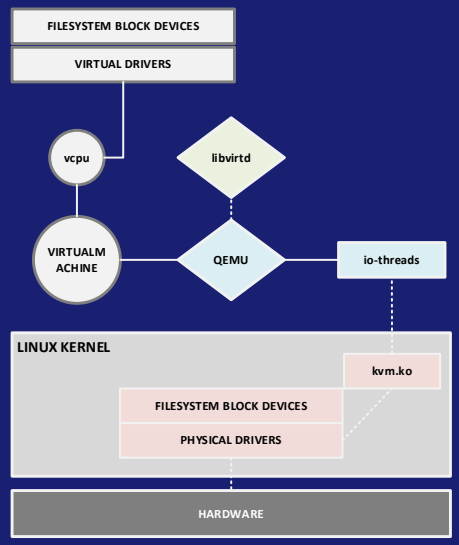
```
$ systemctl status dbus
$ gdbus introspect --system --dest org.freedesktop.systemd1 --object-path /org
```

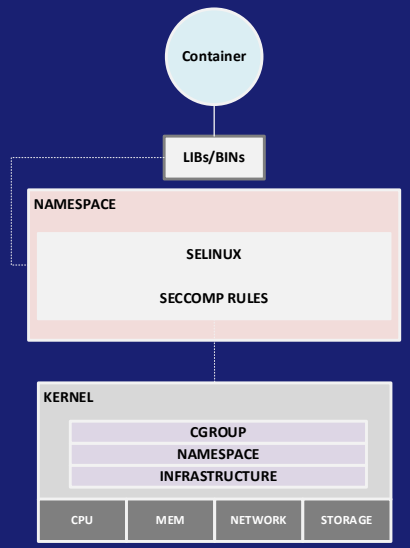
실제로 시스템 운영시에는 많이 사용하지 않는 명령어 이다. 다만, systemd 기반으로 통합이 되고 있기 때문에, 앞으로 dbus에서 문제가 발생하는 경우, 서비스가 올바르게 동작이 되지 않을 수 있다.

컨테이너 및 가상화

커널 2.4이후로 컨테이너 및 가상화 기능을 지원하고 있다. 본래 리눅스의 가상화 및 컨테이너 프로젝트는 "Linux vServer"에서 시작하였다. 컨테이너 기능은 구글에서 제공한 cgroup, 그리고 레드햇에서 제공한 namespace를 통해서 최종적으로 구현이 되었으며, 이를 기반으로 rootless컨테이너 런타임인 docker가 탄생하게 되었다.

현재 표준 컨테이너 런타임은 Docker에서 Podman으로 전환이 되고 있다.





리눅스 모듈

초기 리눅스는 소스코드에 빌트인 형태로 드라이버, 즉, 커널 모듈을 배포하였다. 이 방법에 문제는 새로 추가하거나 혹은 업데이트 시, 매번 커널을 다시 컴파일을 해야 하는 문제가 있었다.

초창기 커널은 이러한 기능이 없었고, **"리눅스 커널 버전 1.0"**이후로는 모든 커널에서 모듈 기능을 제공한다. 초창기 리눅스는 x86만 고려하여 제작이 되었기 때문에, 모듈 및 CPU를 확장 지원을 할 계획이 없었다.

리눅스 커널에 포함이 되어 있는 모듈은 대다수가 **GNU GPLv2/3**기반으로 되어 있다. BSD 및 MIT라이센스 모듈도 종종 있지만, 해당 모듈은 **mainline module**로 포함이 되지 않는다.

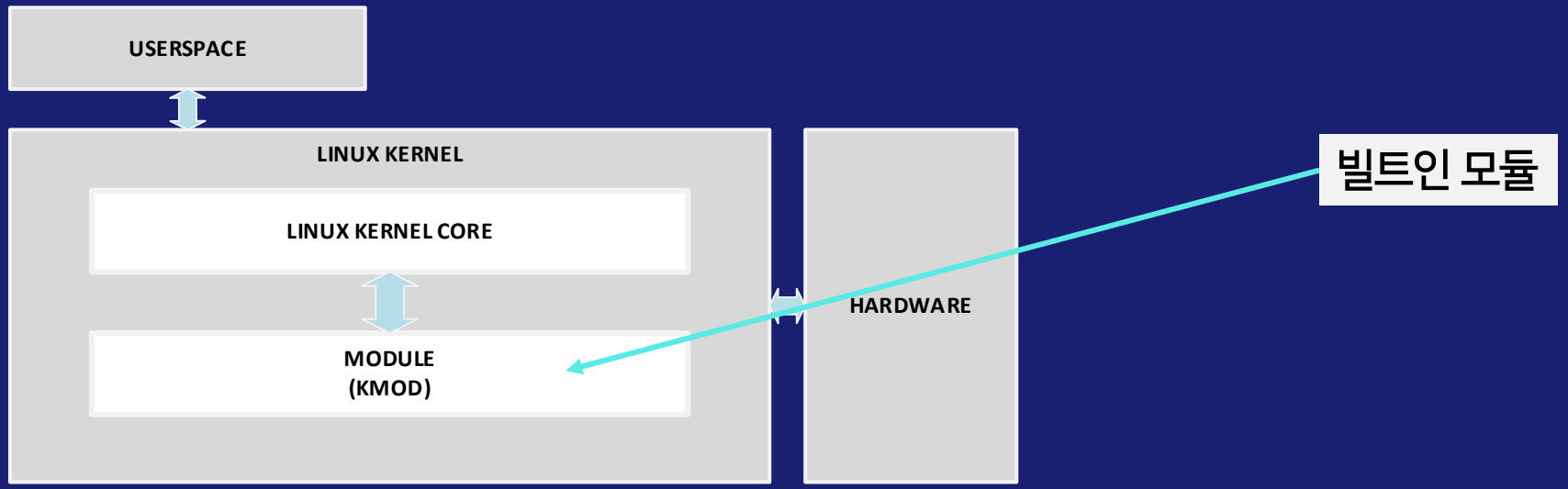
대표적인 모듈이 바로 **Nvidia** 모듈이다. 우분투 배포판을 제외하고, 대다수 GNU배포판은 Nvidia드라이버를 저장소에서 제공하지 않는다. 심지어 라이브러리 링크(ldconfig)도 제공하지 않는다.

리눅스 모듈(kmod)

일반적인 리눅스 커널 모듈은 전부 "kmod"모듈 기반으로 구성이 되어 있다. "kmod"는 일반적인 리눅스 커널 모듈이다. 'modprobe', 'insmod'같은 명령어를 통해서 메모리에 커널을 상주하여 하드웨어를 초기화한다.

장점: 커널과 호환성 및 안정성이 매우 높다. 빌트인 혹은 모듈 형태로 활성화 된 모듈은 바로 사용이 가능하다.

단점: 다른 커널 버전에서 동작이 불가능하다.

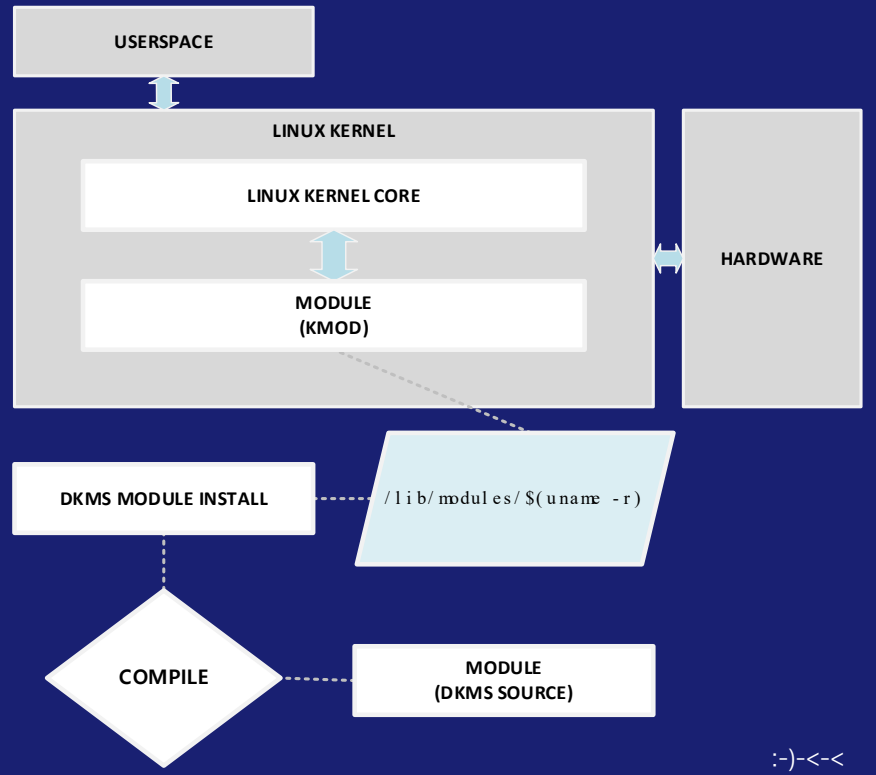


리눅스 모듈(dkms)

커널 컴파일을 별도로 요구하지 않는 부분이 장점이며, 모듈만 컴파일 혹은 바이너리 설치하여 바로 모듈로 사용이 가능하다. 보통 dkms, akmod라는 이름으로 배포판에서 제공한다. 다만, 사용하기 위해서 라이브러리 링크 컴파일이 종종 필요하기 때문에 컴파일러 도구가 필요하다.

장점: kmod의 보다 사용하기 편한다.

단점: 호환성 문제가 발생할 수 있다.



커널모듈 정리

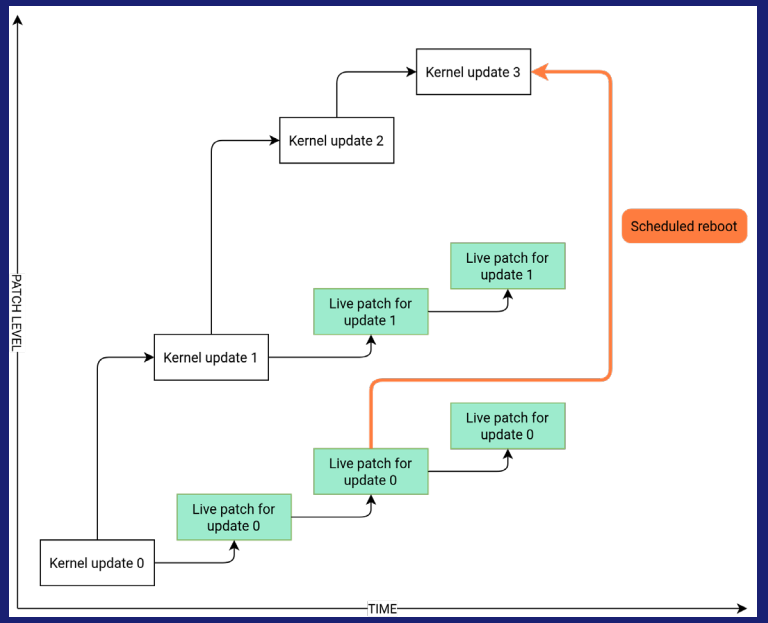
커널 모듈 기능을 정리하면 다음과 같다.

이름	설명
kmod	빌트인 커널. 다른 커널 버전에서 사용이 불가능하다.
akmod	빌트인 커널과 비슷하지만, 이 모듈은 커널 업데이트를 신경 쓰지 않고 사용이 가능하다.
dkms	외부에서 별도로 제작해서 제공하는 커널. 보통 Nvidia커널이 dkms형태로 제공이 되며, 이 형태로 되어 있는 모듈은 커널과 링크 형태로 제공이 된다.

엔비디아 리눅스 모듈

커널 라이브 패치

이전 리눅스 커널은 유닉스 커널처럼 라이브 패치 기능이 없었다. 현재, 라이브 패치 기능이 추가가 되었으며, 공식적인 이름은 "kpatch"이다. 현재 최신 리눅스 커널은 이 기능 사용이 가능하다. 사용하기 위해서는 다음과 같은 기능이 활성화가 되어 있어야 한다.



수세 kpatch 문서

커널 라이브 패치 예제(kpatch)

커널 라이브 패치를 하기 위해서 다음처럼 간단하게 구성이 가능하다. 다만, 버전에 따라서 동작이 안될 수 있으니, 이점 주의한다. 아래는 간단하게 레드햇 계열에서 사용하는 "kpatch"를 구현한다.

```
# dnf update kernel
# reboot

# dnf install kexec-tools
# kexec -l /boot/vmlinuz-3.10.0-862.3.2.el7.x86_64 --
initrd=/boot/initramfs-3.10.0-862.3.2.el7.x86_64.img --reuse-cmdline
# sync; sudo umount -a; sudo kexec -e
```

커널 컴파일 혹은 리빌드

대다수 리눅스 배포판은 커널 컴파일을 더 이상 요구하지 않는다. 만약, 필요한 경우 사용자가 스스로 커널 컴파일하여 리눅스 시스템에 적용 및 제공이 가능하다. 다만, 상용 리눅스 배포판 경우에는 사용자라 리눅스 컴파일을 하여 적용한 경우, 모든 기술지원이 무효가 되기 때문에 추천하지 않는다.

참고로 uEFI에서는 더 이상 32비트 커널 이미지 부팅을 제공하지 않기 때문에, 컴파일시 64비트로만 커널 이미지를 컴파일 해야 한다. 수동으로 커널 구성 시, uEFI의 shim signed 문제가 종종 발생한다.

```
# curl http://dl.rockylinux.org/pub/rocky/9/RT/source/tree/Packages/k/kernel-rt-5.14.0-284.30.1.rt14.315.el9_2.src.rpm -O kernel-rt-5.14.0-284.30.1.rt14.315.el9_2.src.rpm
# cd rpmbuild/SPECS
# cp configs/kernel-rt-5.14.0-x86_64.config .config
# rpmbuild -bp --target=$(uname -m) kernel.spec
# make oldconfig
# make menuconfig
# rpmbuild -bb --with baseonly --clean --target=$(uname -m) kernel.spec 2> # build-err.log | tee build-out.log
```

쉘

사용자 영역

쉘은 무엇인가?

쉘은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == SHELL라는 오해를 가지고 있음.

쉘은 일반적으로 사용하는 CLI 도구와는 다름. BASH와 같은 도구는 "CLI(COMMAND LINE INTERFACE)"와 비슷하기는 하지만, 대화형 형태로도 사용이 가능하다. 그래서, BASH쉘에서 사용할 명령어의 이름을 넣어주면, BASH는 사용자가 입력한 이름의 바이너리 파일을 찾아서 실행한다.

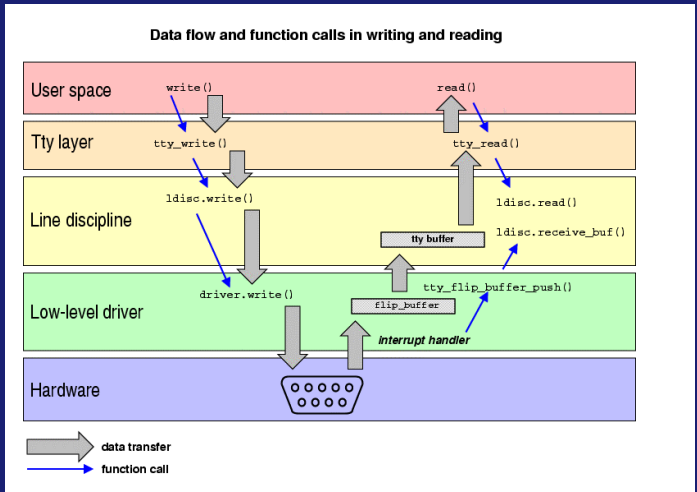
BASH사이트 다음과 같이 BASH를 설명하고 있다.

https://www.gnu.org/software/bash/manual/bash.txt

THE TTY

TTY는 "Teletypewriter"의 약자. 초기 유닉스 시스템은 로컬만 지원하였으며, BSD의 유닉스 프로젝트를 통해서 TCP/IP를 도입 후, 원격에서 시스템 접근이 필요. 모든 리눅스 배포판도 커널의 TTY장치를 통해서 터미널 구성 후 쉘에 연결한다.

"A Teleprinter just needed a single operator to easily convey a message. While it did not have a modern-layout keyboard, its system was later evolved by Donald Murray in 1901 to include a typewriter-like keyboard.



현재 리눅스에서 제공하는 쉘 목록

리눅스 배포 판별로 다르지만, 현재 대다수 리눅스에서 제공하는 쉘은 "bash"기반으로 제공한다. 아래는 일반적으로 배포판에서 제공하는 쉘 목록이다.

- tcsh(csh)
- zsh
- fish
- ksh
- dash
- mksh
- **bash**

모던 리눅스에서 사용하는 쉘

현재 대다수 리눅스 배포판은 더 이상 "tcsh", "ksh"를 사용하지 않는다. 대다수 사용자는 현재 다음과 같은 쉘로 환경을 변경하고 있다.

1. zsh
2. csh
3. bash

압도적으로 "bash"사용자가 많지만, 최근에는 "zsh"사용자도 많이 늘어났기 때문에, 이전에 비해서 사용자가 늘어남. 현재 레드햇 계열 배포판은 "bash"가 기본 쉘이다.

SHELL POSIX

POSIX에서 명시한 'echo', 'pwd', 'cd'와 같은 빌트인 명령어는 불행하게도 **UNIX System V, BSD,** 그리고 **GNU** 기반에 따라서 각기 다르게 동작한다. 예를 들어서 "GNU echo"명령어는 표준은 UNIX System V, BSD에서는 다르게 동작 및 결과를 출력한다.

그러한 이유로, "echo"명령어 대신 호환성을 높이기 위해서 "printf"명령어 사용하는 것을 권장하였다. 대다수 주요 쉘은 POSIX.1-2017를 지원 및 제공하고 있다. 지원하지 않는 경우 쉘 스크립트 호환성을 매우 낮아진다.

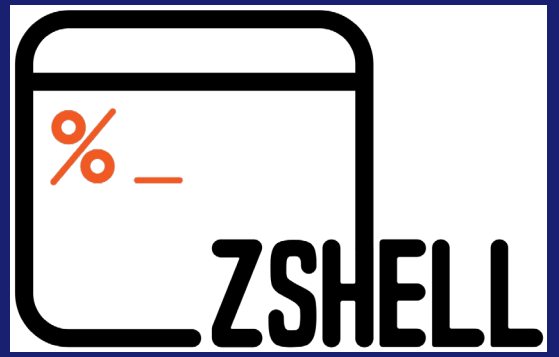
POSIX.1-2017: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

쉘 종류(zsh)

zsh는 오랫동안 사용하던 쉘이며, 1990년도에 만들어졌다. zsh는 ksh, tcsh에서 사용하는 명령어 자동완성 및 함수와 같은 부분이 호환이 된다. zsh은 마이크로소프트에서 윈도우 UnixUtils에 포함 시키면서 다시, 많은 사람들이 사용하기 시작하였고, 현재는 MacOS, Kail Linux에서는 기본쉘로 사용하고 있다.

현재 확장 기능인 "Oh My Zsh"이라는 사용자 커뮤니티를 통해서 플러그인 및 테마와 같은 확장 기능을 추가적으로 설치 및 구성이 가능하다.

하지만, zsh에서 쉘 스크립팅을 사용하는 경우, POSIX표준을 완벽하게 지원하지 않기 때문에, 스크립트 작성이 필요한 경우, bash를 권장한다.



ZSH 스크립트: https://rwx.gg/advice/dont/zsh/

쉘 종류(zsh)

```
# dnf install zsh -y

# chsh -s /bin/zsh

# sh -c "$(curl -fsSL
https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"

# vi .zshrc

> ZSH_THEME="cloud"
```

KSH/BASH 차이점

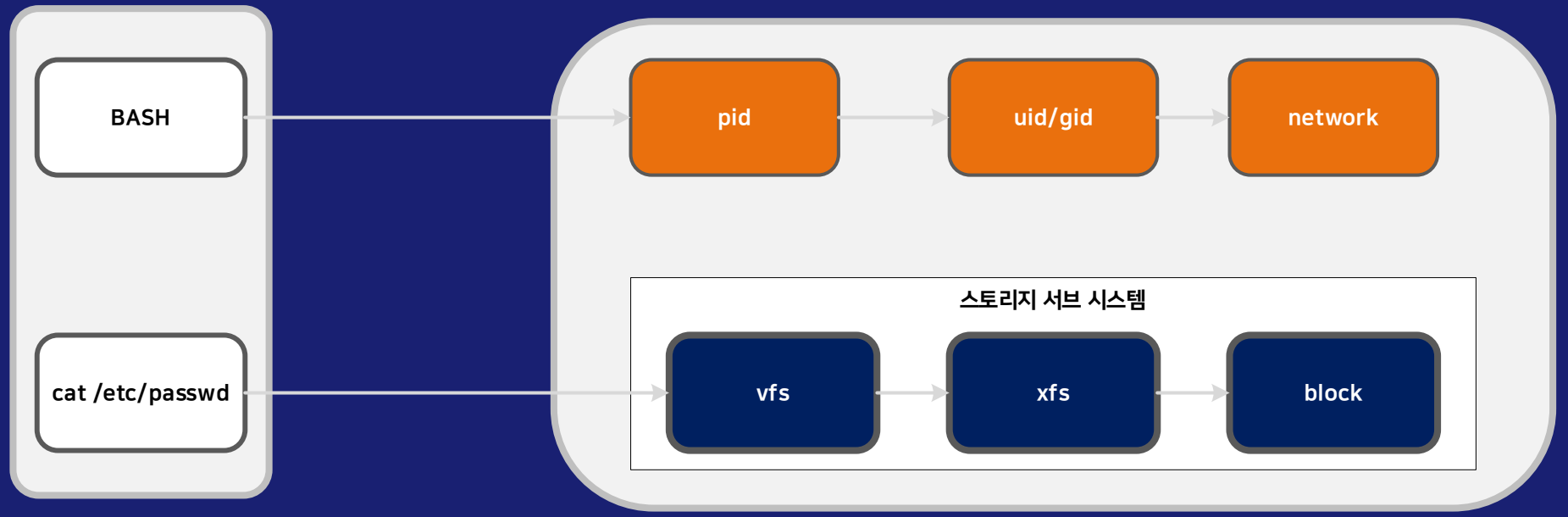
ksh와 bash의 큰 차이점은 실제로는 거의 없다.

하지만, Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base의 내용에 충돌이 되기 때문에 서로 호환이 되지 않는 부분이 있다. 예를 들어서 LSB에서는 /usr/xpg4/bin를 사용하지 않으며 또한 솔라리스도 이러한 POSIX규격을 따르지 않는다.

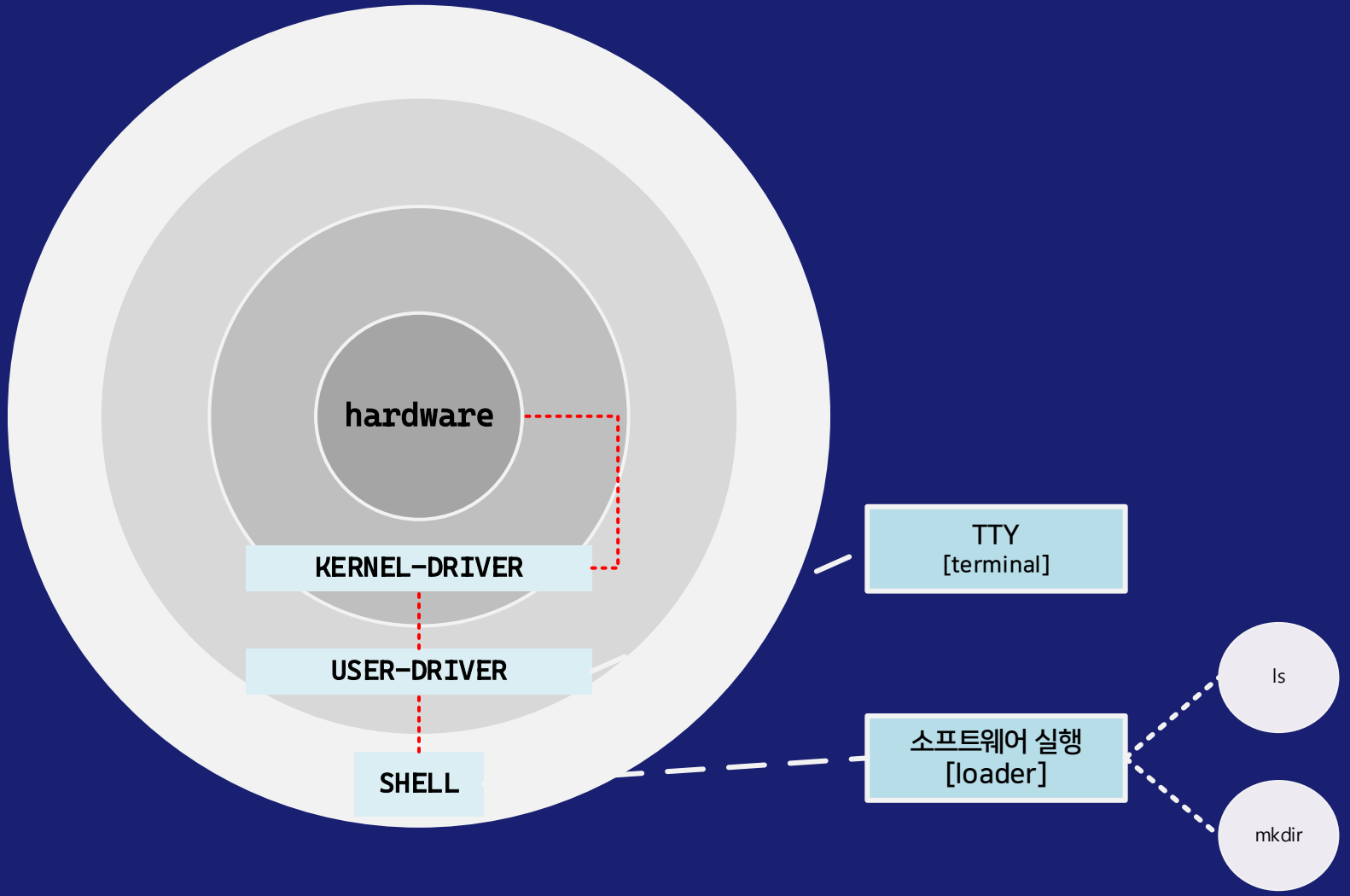
리눅스 및 솔라리스 경우 2004년도 기준으로 양쪽 진영은 표준 사양을 변경하였다. 이를 통해서 리눅스 및 솔라리스 같은 유닉스와 스크립트 격차를 많이 줄었다.

XPG4: https://www.unix.com/solaris/164073-what-difference-between-xpg4-bin-usr-bin.html

쉘 동작 방식


BASH
cat /etc/passwd
pid
uid/gid
network
스토리지 서브 시스템
vfs
xfs
block

쉘과 하드웨어 인트럽트


hardware
KERNEL-DRIVER
USER-DRIVER
SHELL
TTY
[terminal]
소프트웨어 실행
[loader]
ls
mkdir

최근 리눅스 구조

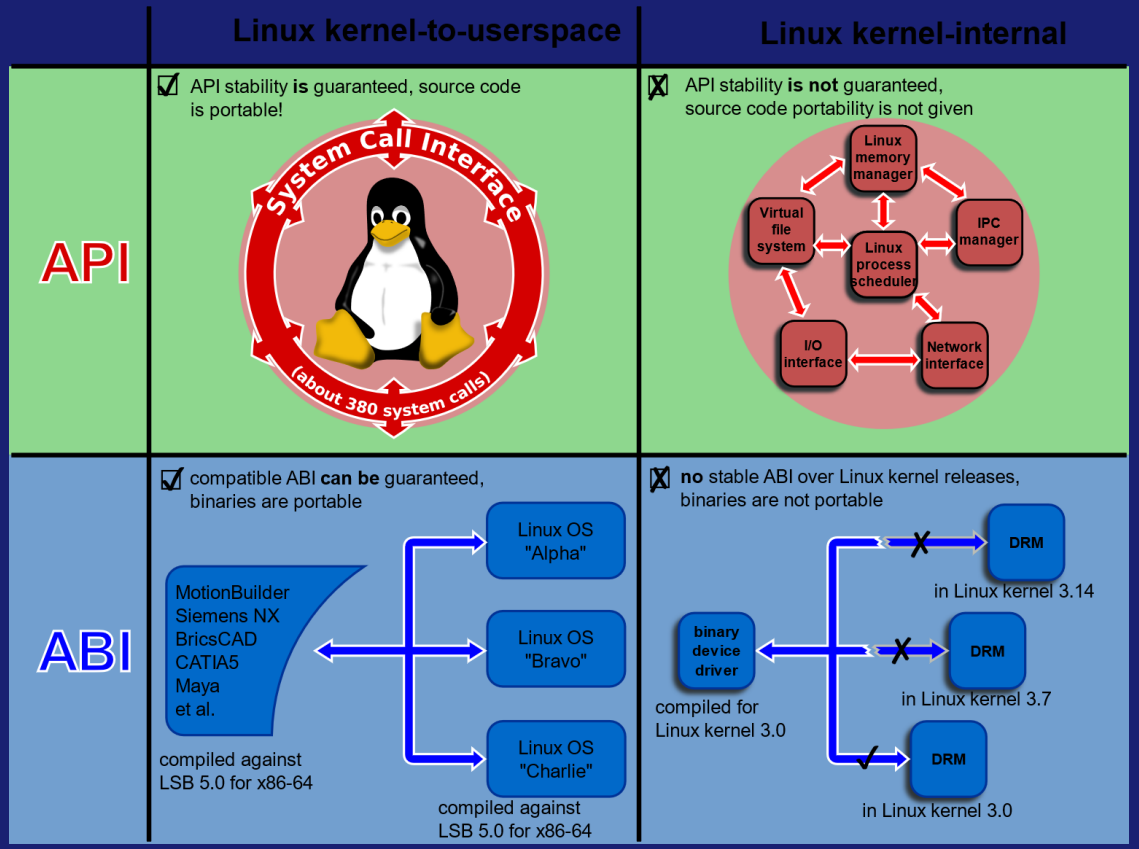
현대화된 리눅스 시스템

현대화된 리눅스 구조

최근에 나온 리눅스 아래와 같은 도구로 통일 및 통합이 되고 있다.

1. nftables/firewalld
2. NetworkManager/NetPlan/systemd-networkd
3. dbus/systemd
4. API/ABI/KABI
5. Linux Kernel/POSIX LAYER

API/ABI



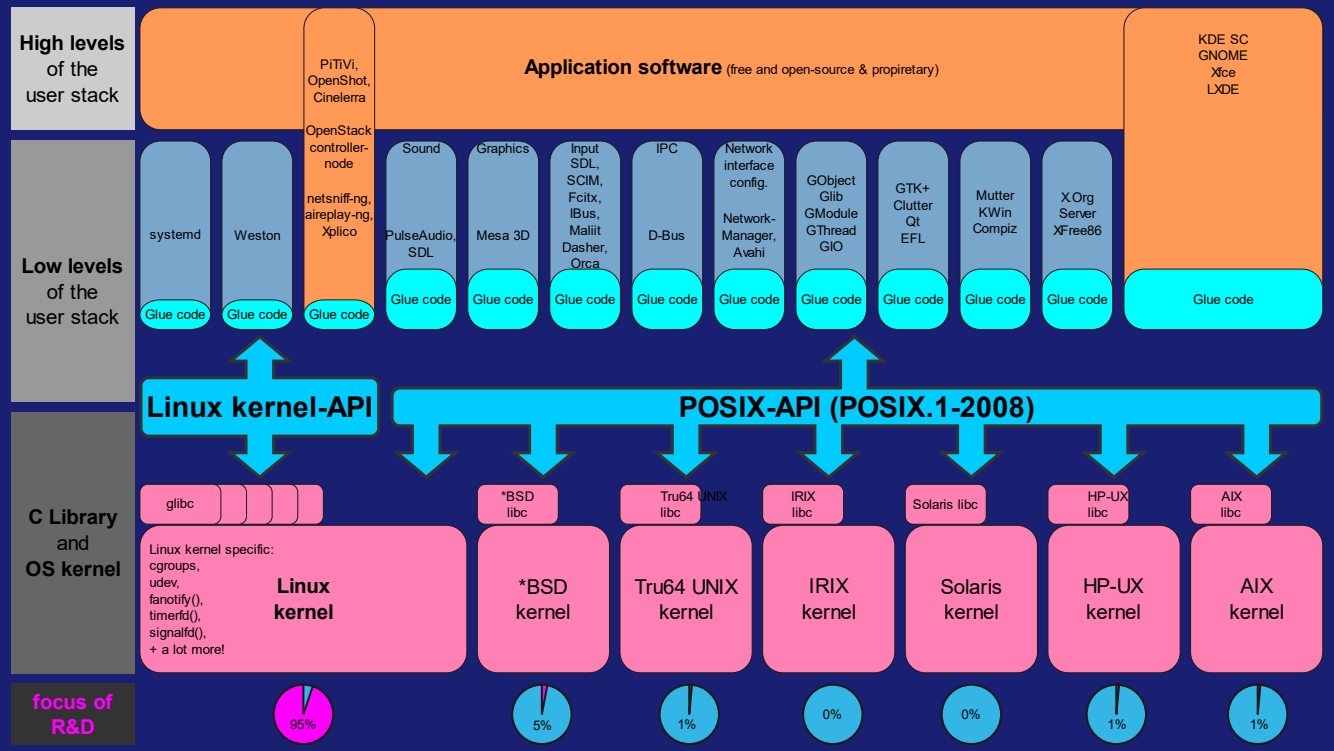
Application Programming Interface

리눅스 커널 혹은 애플리케이션에서 사용이 가능하다. 하지만, 리눅스 커널 기반의 API는 서로 간 호환성을 보장하지 않는다.

Application Binary Interface (ABI)

일반 사용자들이나 혹은 최종 개발자들은 ABI에 관심이 더 많다. 애플리케이션은 같은 LSB 버전을 사용하는 경우 호환이 된다. 하지만, 커널은 같은 버전이 아닌 경우, 호환성이 거의 보장되지 않는다.

LINUX/POSIX



POSIX는 각기 다른 유닉스 커널의 시스템 콜의 호환성을 위해 만들어졌다.
리눅스도 POSIX를 지원을 하며, 각기 유닉스에서 사용하던 POSIX를 거의 지원한다.

Portable Operating System Interface의 약자이다.

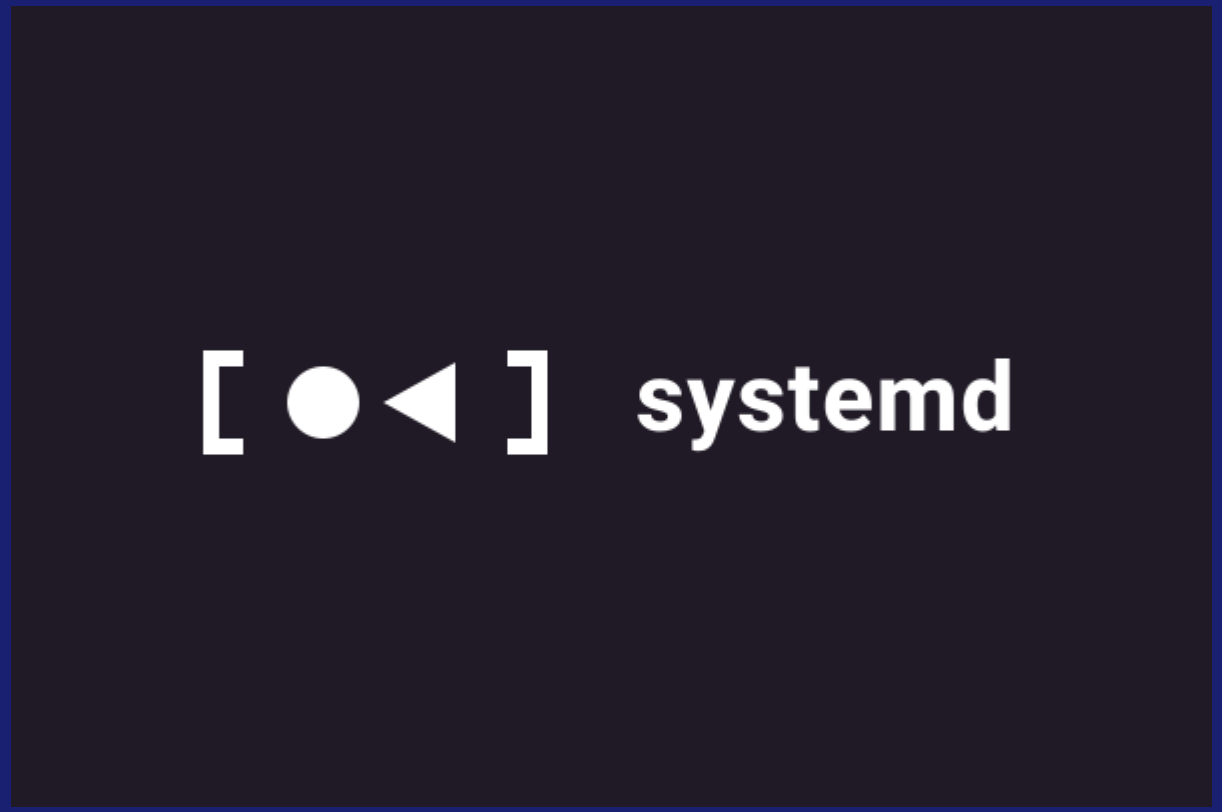
4/1/2024

systemd

시스템 설명 및 통합 및 변경된 부분

간단한 서비스 및 명령어 소개

systemd



systemd

systemd는 다음과 같은 구조로 부팅이 된다. target은 일종의 서비스 묶음, 그리고 이전의 런 레벨과 비슷한 역할을 한다. 옆에 그림은 간단하게 systemd에서 부팅 단계를 다이어그램으로 표시 하였다.

systemd는 레드햇의 업 스트리밍 버전인 페도라 20버전에서 실험적으로 도입이 되었으며, 안정화 및 기능 정립이 완료가 된 후, 정식적으로 RHEL버전에 도입 되었다. 기존 System V와 많이 다른 부분은 cgroup, D-Bus기능이 시스템 systemd의 시스템 블록으로 통합 되었다. 운영체제 업데이트 부분은 systemd의 initrd(ramdisk)에서 실행한다.

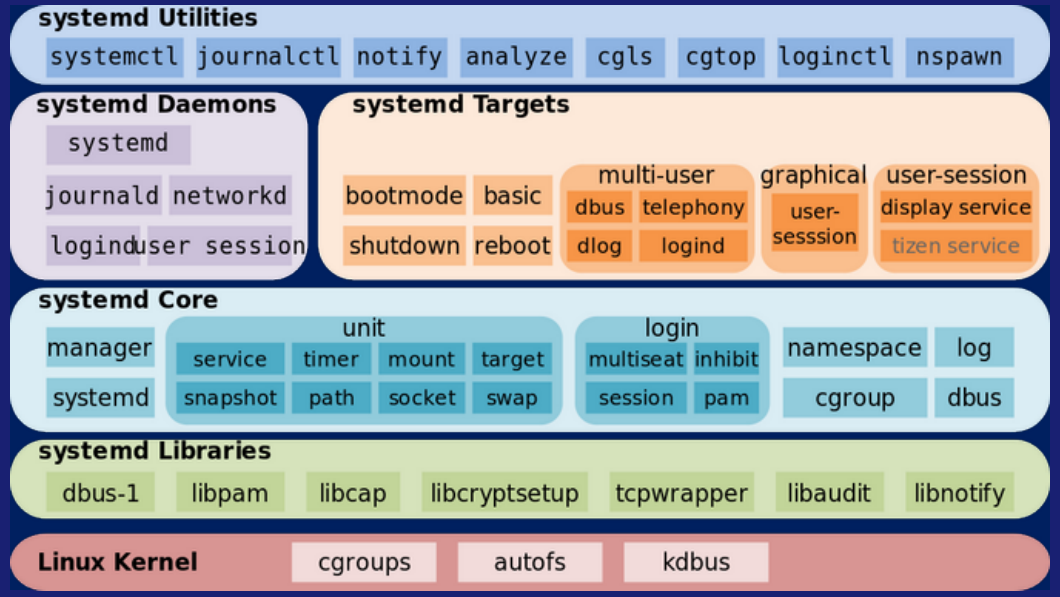
현재 LSB사양을 따르는 대다수 배포판은 systemd기반으로 init의 PID 1번으로 사용하고 있다.

systemd


systemd
dracut-pre-mount.service
sysroot.mount
dracut-mount.service
initrd-parse-etc.service
ramdisk service
sysroot-user.mount
x-initrd.mount
[/etc/fstab]
initrd-fs.target
initrd.target
switch-root
initrd-switch-root.service
initrd-switch-root.target
initrd-udevadm-cleanup-db.service
initrd-cleanup.service
isolate "target"
initrd-switch-root.target
custom ramdisk service
dracut-pre-pivot.service

systemd

systemd는 기존에 사용하던 "system-v", "up-start"를 대신한다. 현재 사용중인 LSB(Linux Standard Base)에서는 systemd기반으로 구성이 되어있다. 레드햇 계열의 리눅스는 systemd를 채용하고 있다.



systemd

"systemd"는 PID 1번을 대체함. 기존에 사용하던 "System-V INIT", "Up-Start(INIT)" 혹은 "BSD INIT"를 더 이상 사용하지 않는다. 기존에 사용하던 init들은 시작만 바이너리이며, 대다수 자원은 쉘 스크립트 기반으로 구성 및 관리 하였다.

예를 들어서, 기존에 사용하던 쉘 스크립트는 어떤 서비스가 어떤 파일이나 혹은 디렉터리는 **생성/삭제/제거**하는지 알 수 없다. 하지만, systemd는 어떠한 부분에 대해서 자원 변경 및 수정이 발생하였는지 확인이 가능하다.

또한, systemd는 이전에 외부 프로그램으로 사용하였던 부분도, 시스템 블록 영역으로 통합하고 있다. 대표적인 서비스는 "chronyd", "crond"와 같은 서비스가 있다.

관리 명령어(ctl1)

systemd로 통합이 되면서, 장치 영역 및 커널 모듈과 같은 부분은 **"조정 명령어"**가 생기면서, 조정 명령어로 대다수 하드웨어 및 커널에 대해서 조정이 가능하다.

너무 명령어가 많기 때문에 아래 명령어로 간단히 **ctl(조정 명령어)**를 확인한다. 아래 명령어는 가급적이면 BASH에서 실행.

```
# find /usr/bin -name "*ctl" -type f -print -exec grep -IL . "{}" \; | uniq
/usr/bin/keyctl
/usr/bin/teamdctl
/usr/bin/powerprofilesctl
/usr/bin/wdctl
/usr/bin/busctl
/usr/bin/hostnamectl
/usr/bin/journalctl
```

관리 명령어(ctl2)

```
/usr/bin/localectl
/usr/bin/loginctl
/usr/bin/systemctl
/usr/bin/timedatectl
/usr/bin/bluetoothctl
/usr/bin/boltctl
/usr/bin/switcherooctl
/usr/bin/bootctl
/usr/bin/coredumpctl
```

관리 명령어(ctl3)

```
/usr/bin/evmctl
/usr/bin/kdumpctl
/usr/bin/udisksctl
/usr/bin/pactl
/usr/bin/wpctl
/usr/bin/flatpak-coredumpctl
/usr/bin/sg_stream_ctl
/usr/bin/sg_bg_ctl
/usr/bin/machinectl
/usr/bin/swtpm_ioctl
```

관리 명령어

대표적인 관리 명령어는 다음과 같다. 모든 명령어를 다 확인이 불가능 하지만, 대표적인 몇몇 명령어만 다루어 본다.

ż╝♂̓n╓ł ╓ĐČ̓n┌

호스트 이름을 설정하는 도구. 사용 파일은 **"/etc/hostname"**이지만, 수동으로 변경은 권장하지 않으며, 호스트 관련 정보는 이 명령어로 수정을 권장 한다. 추가적으로 **CPE**정보를 추가 제공한다.

https://www.redhat.com/en/blog/common-platform-enumeration

ʼnδ ╓Đă ł ʼnĐČ̓n┌

시간/날짜 및 타임존은 더 이상 **'tzselect'**명령어가 아닌, 이 명령어로 수정한다. 가급적이면 수동으로 하지 않는다. "NTP" 서버 및 클라이언트 정보 확인도 가능합니다. 다만, "chronyd"아닌, "systemd-timesyncd", "systemd-timedated"를 통해서 확인이 가능하다.

ʼnĐł ╓ă Č̓n┌

리눅스 본딩를 대안으로 나온 티밍 서비스 관리 명령어. 여전히 본딩을 지원하고 있으나, 가급적으면 Teamd기반으로 사용을 권장한다. 본딩 티밍은 이전 rh-ifcfg형식을 지원하지 않는다.

https://github.com/jpirko/libteam

관리 명령어

╔╣ Čž δ ╔ĐČn ┌

systemd기반으로 가상머신이나 혹은 컨테이너 서비스를 동작한다. 지원 파일 형식은 "raw", "tar(Containerfile)"를 지원한다. "rootless", "rootful"형태로 구성이 가능.

▪ ă δ♂─♂Čn ┌

기존의 'lsblk', 'mount', 'umount'으로 관리하는 부분을 dbus를 통해서 통합 및 관리. 앞으로 장지 연결 및 구성은 이 명령어로 관리한다. "systemd-mountd"와 같이 상호작용한다. ".mount"유닛 통해서 구성된 자원을 관리한다.

♂ī ♂Čn ┌

리눅스 커널 파라메터 조정 및 변경 명령어. 해당 명령어는 systemd과 통합이 되지 않았음. 이 과정에서는 이 부분에 대해서 다루지 않는다.

```
# vi /etc/sysctl.d/ipv4.conf
```

통합된 자원

’nδ ╔Ð▣╕ Č▣╝ ╔ă t

기존 "crond" 서비스와 비슷하게, 특정 시간에 명령어 실행을 도와주는 서비스 유닛. 기존의 crond데몬은 ".timer" 로 대체가 될 예정이다. 이 부분은 뒤에서 더 다루도록 한다.

╒╝Čł ╔Ð

'localectl'으로 시스템 로케일을 변경한다. 이전에는 수동으로 로케일 설정을 변경하였으나, 현재는 dbus 및 systemd통합으로 인하여 해당 명령어로 변경을 권장하고 있다.

이 명령어는 **시스템 전역(system profile)**으로 적용 시, 사용을 권장. **사용자 영역(user session)**은 기존과 동일하게 로케일 시스템 변수를 그대로 사용이 가능함.

통합된 자원

↕⌟▪◙╔ ┌Čn┌

"rsyslog(/var/log/)"를 대신하여 앞으로 "systemd-journald"서비스로 관리한다. 현재 대다수 리눅스 배포판은 "journald"기반으로 넘어가고 있다. 이 부분도 뒤에서 더 다루도록 한다.

systemd에서 로그 기록을 남기는 곳은 다음과 같다.

- /run/log/journald
- /var/log/journald

systemd

`systemd dbus`

시스템에서 사용하는 dbus채널 및 연결 상태를 확인한다. 이는 윈도우 서버에서 사용하는 com/dcom와 비슷한 개념이다. '**busctl**'명령어를 통해서 확인이 가능하다.

`logind`

사용자 로그인 정책을 관리 시 사용하는 명령어. 이 명령어를 통해서 사용자 시작 프로그램 및 시스템에 연결이 되어 있는 사용자 잠금 및 차단 기능을 사용할 수 있다. 로그인 관리 명령어는 '**loginctl**'를 통해서 관리한다.

`bootctl`

systemd에서 제공하는 **부트 로더(boot loader)**. 일반적인 배포판은 grub2기반으로 되어 있기 때문에 모든 기능을 사용할 수는 없지만, 기본적인 상태 확인 기능은 사용이 가능하다.

관리 명령어

┌╝ţ δ ╔Čn ┌

이전에는 로그인 세션을 프로세스 단위로 관리하였지만, 지금은 systemd에서 cgroup기반으로 세션 관리를 한다.

이전에 사용하였던 'last', 'lastlog', 'w'와 그리고 'pgrep', 'pkill'를 통해서 사용자 프로세스 종료가 가능하다. 역시, 이 부분도 systemd를 통해서 통합 관리가 되고 있다.

```
# loginctl list-users
 UID USER LINGER STATE
1000 tang no     active

1 users listed.
# loginctl list-sessions
SESSION  UID USER SEAT TTY   STATE  IDLE SINCE
      1 1000 tang      pts/0 active no

1 sessions listed.
```

관리 명령어

┌╝ţ δ ╔Čn ┌

세션 종료 이후에도 계속 프로그램이 동작이 되려면, 다음처럼 'loginctl'명령어로 세션 설정을 변경해야 한다.

```
# loginctl enable-linger
loginctl show-user test1 | grep Linger
Linger=yes
```

통합된 명령어

systemd에서는 네트워크 기능이 통합이 되었다. 현재 사용중인 "ifcfg-script", "NetworkManager"는 앞으로 "systemd-networkd"로 변경이 된다. "

'networkctl'명령어는 다음과 같이 사용이 가능하다. 이 부분에 대해서는 **네트워크 세션**에서 한번 더 다룬다.

```
$ sudo networkctl list
IDX LINK TYPE     OPERATIONAL SETUP
  1 lo   loopback carrier     unmanaged
  2 eth0 ether    routable    unmanaged
  3 eth1 ether    routable    unmanaged
```

통합된 명령어

systemd는 다음과 같은 영역 및 서비스들이 통합되고 있다. 아래는 영역별로 간단하게 정리한 내용이다.

hostnamectl

호스트 이름은 'hostnamectl'로 통합이 되고 있다. "hostnamectl"은 systemd의 자원의 일부분이며, **"호스트 이름"** 이외, 컴퓨터가 사용하는 **"머신 아이디"** 및 **"폼-펙터(Form Factor)"** 정보를 가지고 있다.

```
# hostnamectl set-hostname servera.example.com
# hostnamectl deployment lab
# hostnamectl icon-name vm
# hostnamectl chassis laptop
# hostnamectl location SEOUL
```

통합된 명령어

timedate

호스트의 시간 및 날짜 그리고, NTP를 담당하는 서비스. 앞으로 NTP관련된 부분은 "systemd-timedated.service"으로 변경될 예정이다. 사용 방법은 아래와 같다.

```
# timedatectl ntp-servers
# timedatectl revert
# timedatectl timesync-status
# timedatectl show-timesync
# timedatectl set-ntp false
```

통합된 명령어

```
# timedatectl status
# timedatectl show
# timedatectl set-time 13:20:00
# timedatectl list-timezones
> Asia/Tokyo
# timedatectl set-timezone Asia/Tokyo
# timedatectl set-local-rtc true
# timedatectl set-ntp true
```

통합된 명령어

machinectl

systemd기반으로 컨테이너 혹은 가상머신을 서비스 형태로 사용 시 사용한다. 아래 명령어로 간단하게 컨테이너를 구성한다.

```
# dnf install guestfs-tools libvirt -y
# systemctl enable --now libvirtd
# systemctl is-active libvirtd
# export LIBGUESTFS_BACKEND=direct
# virt-builder cirros-0.3.5
# setenforce 0
# machinectl import-raw cirros-0.3.5.img cirros
# machinectl list-images
# systemd-nspawn -M cirros
# passwd
# exit
# machinectl start cirros
# machinectl login cirros
# machinectl list
```

nspawn(spawn, light-weight container)

systemd에서 네임스페이스 및 cgroup기반으로 컨테이너 생성하는 명령어.

현재 대다수 배포판은 "flatpack", "snap"같은 비설치형 애플리케이션 관리자를 제공한다. 이러한 애플리케이션 관리자는 "systemd-nspawn"기반으로 관리 및 구성한다.

```
# dnf install systemd-container
# dnf -y --releasever=9 --nogpg --installroot=/srv/test install systemd passwd dnf
centos-release vim-minimal
# setenforce 0
# systemd-nspawn -D /srv/test
# passwd
# exit
# setenforce 1
# systemd-nspawn -D /srv/test/ -b
```

통합된 명령어

localectl

이전에는 "LC_ALL", "LANG"를 "/etc/sysconfig/locale". 하지만, 지금은 **systemd**으로 통합이 되었기 때문에, 'localectl' 명령어로 수정 및 변경해야 한다.

```
# localectl
System Locale: LANG=ko_KR.UTF-8
    VC Keymap: kr
   X11 Layout: kr
# localectl set-locale C
# localectl
System Locale: LANG=C
    VC Keymap: kr
   X11 Layout: kr
```

통합된 명령어

udisksctl

'mount', 'umount'명령어로 관리하던 시스템 파일 시스템 마운트는 'udisksctl'로 통합이 될 예정이다. 지원하는 기능은 이전 명령과 비슷하게 블록장치 연결 및 상태 정보를 확인한다.

```
# udisksctl status
Msft Virtual Disk         1.0       600224800179ee329d0fd1ca5a1efecc sda
Msft Virtual DVD-ROM      1.0       4d53465420202020207305e34377035446949 57d7ced624a7d sr0
# udisksctl mount -b /dev/sdb
Mounted /dev/sdb at /run/media/root/c7b98cdb-de0b-4b6f-a22e-36e4a2cb8669
```

systemd 명령어 자동화

진행 전, bash completion이나 혹은 fish, zsh 쉘 기반으로 작업을 권장. bash completion를 사용하기 위해서는 다음과 같이 명령어 실행. zsh에서도 같이 사용이 가능.

```
# dnf install bash-completion
# rpm -qa bash-completion
# complete -r -p
```

다시 로그인 혹은 bash명령어 실행. 'source'명령어는 종종 오류가 발생함.

```
# bash
# source /etc/profile.d/bash_completion.sh
```

런-레벨 변경

"emergency.target", "resuce.target"경우에는 init(Sys-V)에서 Single혹은 S와 비슷함. 단, 응급복구 모드인 rd.break(ramdisk break)경우, 이전 ramdisk기반 복구 기능과 조금 다르다.

- `multi-user → init 3`
- `graphical  → init 5`

런-레벨 확인을 하기 위해서 아래와 같은 명령어로 조회가 가능하다.

```
# systemctl -t target list-units -all
# systemctl get-defaults
# systemctl list-dependencies <TARGET_NAME>
# systemctl isolate multi-user (# init 3)
```

TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
default.target	링크파일	graphical.target multi-user.target	기본적으로 **multi-user.target**이나 혹은 **graphical.target**으로 링크가 되어 있음.
graphical.target	5	runlevel5.target	multi-user에서 GUI로 전환
	4	runlevel4.target	사용하지 않음. 본래 런 레벨에서는 4번은 사용하지 않았음. 서비스를 등록하는 경우 multi-user.target를 수정하지 않는 선에서 사용이 가능하다.
multi-user.target	3	runlevel3.target	CLI(console)용 대상. 이전에는 런 레벨 3번이었다.

TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
없음	2	runlevel2.target	multi-user기반. 하지만, CLI기반으로 부팅이 되며, NFS 및 네트워크 서비스는 시작이 되지 않음.
rescue.target	1	runlevel1.target	읽기 전용 상태로 **싱글모드(single mode)**로 디스크의 루트 파일 시스템을 램 디스크에 마운트. "rd.break", "rd.rescue"
emergency.target	S		램 디스크에서 어떠한 서비스도 동작하지 않고, 어떠한 블록 장치도 디스크에 연결이 되어 있지 않는 상태. 오직 메인 콘솔만(tty1) 열려 있는 상태.
halt.target			시스템 종료. 하지만 powering단계는 무시.
reboot.target	6	runlevel6.target	재시작
poweroff.target	0	runlevel0.target	시스템 종료 및 powering도 같이 적용.

systemctl unit

systemd에서 관리하는 대다수 자원은 'systemctl'명령어 기반으로 대다수 유닛을 제어 및 관리한다.

'systemctl'명령어는 별도로 옵션을 지정하지 않으면 기본적으로 ".service"유닛으로 명령어를 실행한다. 이 자원 이외에도 꽤 많은 자원이 있으며, systemd버전에 따라서 자원이 형식이 추가가 될 수 있다.

현재, systemd에서 지원하는 시스템 자원 유닛은 아래와 같다.

```
".service", ".socket",".device", ".mount", ".automount", ".swap",
".target", ".path", ".timer", ".slice", ".scope"
```

systemctl 명령어

systemd기본 명령어는 아래와 같다. 자세한 내용은 뒷부분에서 더 다룰 예정이다.

- **stop:** 동작중인 유닛을 중지한다.
- **status:** 동작중인 유닛 상태 정보를 확인한다. 이 정보는 유닛의 로그도 같이 출력한다.
- **is-active:** 유닛 동작 상태를 확인한다.
- **is-enabled:** 유닛이 부트-업이 가능한지 확인한다.
- **enable:** 서비스 부트-업을 활성화 한다.
- **start/restart:** 서비스 시작 및 재시작을 한다. restart는 stop+start가 동시에 수행이 된다.
- **reload:** 특정 서비스가 메모리 갱신을 지원하면, reload로 설정 파일을 다시 메모리에 적용한다. 대표적인 서비스는 sshd이다. 모든 서비스에서 동작하지 않는다.

notify

다른 유닛의 작업이 완료가 될 때가지 대기 후, 해당 작업이 완료가 되면 나머지 작업을 수행한다. 일반적으로 사용자가 notify를 작성하는 경우는 드물다. 수동으로 서비스를 구성 시 사용한다.

```
# vi /usr/local/bin/waldo.sh
#!/bin/bash
mkfifo /tmp/waldo
sleep 10
systemd-notify --ready --status="Waiting for data…"
while : ; do
        read a < /tmp/waldo
        systemd-notify --status="Processing $a"
        sleep 10
        systemd-notify --status="Waiting for data…"
done
```

notify

```
# vi /etc/systemd/system/waldo.service
[Unit]
Description=My Test
[Service]
Type=notify
ExecStart=/usr/local/bin/mytest.sh
[Install]
WantedBy=multi-user.target
# systemctl daemon-reload
# systemctl start waldo.service
# systemctl status waldo.service
# echo "Hello WalDo!" | tee /tmp/data
# systemctl status waldo.service
```

analyze

systemd에서 서비스 부트 업 시간을 확인하기 위해서 다음과 같이 명령어를 사용한다.

1. 제일 느린 서비스 순서대로 보여주는 명령어

```
# systemd-analyze blame
```

2. 느린 유닛들 위주로 트리 형태로 출력(target)

```
# systemd-analyze critical-chain
```

3. 전체 부팅 시간 확인은 다음과 같은 명령어

```
# systemd-analyze time
```

4. 이미지 형태로 확인

```
# systemd-analyze plot > boot.svg
```

cgls/cgtop

systemd에서 관리하는 유닛들의 CPU, Memory, Disk상태를 추적한다.

systemd로 변경이 된 후, 모든 자원들은 cgroup를 통해서 추적 및 감시를 한다. **가상/컨테이너** 및 **모든 프로그램**들은 전부 cgroup를 통해서 자원 제한 및 추적이 되기 때문에 중요한 자원 중 하나이다.

```
# systemd-cgls
# systemd-cgtop
```

systemd에서는 cgroup를 관리하기 위해서 ".slice"라는 자원을 생성한다.

```
# systemctl set-property httpd.service CPUWeight=200 MemoryMax=2G
# systemctl set-property httpd.service IPAddressDeny=10.10.10.250
```

systemd subsystem

systemd-timesyncd

systemd-timers

systemd-run

systemd-tmpfiles

systemd-mount

systemd-timesyncd

NTP PROTOCOL

chronyd

레드햇 계열 및 데비안 계열에서 아직까지 표준으로 사용하는 NTP표준 서버 및 클라이언트 도구이다. 현재 레드햇 계열은 chronyd기반으로 구성이 되어 있다. 설정 방법은 간단하다. 아래 설정 내용은 기본적으로 설정된 내용이다.

```
# grep -Ev '^#|^$' /etc/chrony.conf
pool 2.rocky.pool.ntp.org iburst
offline
auto_offline
driftfile /var/lib/chrony/drift
makestep 1.0 3
offset  -0.00005
rtcsync
```

NTP서버 정보를 변경 한다.
- pool
- server

iburst/burst의 차이점은 burst는 4번 iburst는 4~8번의 요청을 서비스 시작 시, 바로 요청한다. 다만, burst는 각 요청에 대해서 응답을 기다리기 때문에 동기화가 느리다.

특정한 문제로 시간 값을 잃어버린 경우, 평균 값을 가지고 지속적으로 시간을 조정한다.

chronyd

현재 대다수 리눅스 배포판은 ntpd에서 chronyd로 변경. 설정 내용은 크게 차이는 없다. chronyd를 사용하면 다음과 같은 이점이 있다.

1. 빠른 동기화 속도. 최소로 동기화 하면서 동기화를 한다. 보통 데스크탑 같이, 24시간 동기화가 필요하지 않는 시스템
2. CPU 클럭이 불안전한 시스템에 적합. 예를 들어서 가상머신이나 불안한 클럭 상태의 CPU.
3. 동기화 드리프트 기능 지원. 예를 들어서 데이터베이스 같은 미들웨어.
4. 트래픽이 포화일때 비대칭 처리를 안정적으로 가능.
5. 대다수 배포판은 현재 ntpd를 사용하지 않음.

systemd-timesyncd, ntpd

systemd로 넘어 오면서 "systemd-timesyncd"를 통해서 동기화가 가능하다.

파일 위치는 다음 중 둘 중 하나를 사용해도 된다. 레드햇 포함, 다른 배포판들은 아직 기본으로 설치가 안되어 있기 때문에 사용을 원하는 경우, 설치를 해야 한다.

"systemd-timesyncd"를 실행하는 경우 자동으로 "chronyd.service"는 중지가 된다.

- `/etc/systemd/timesyncd.conf`
- `/etc/systemd/timesyncd.conf.d/local.conf`

```
# dnf install systemd-timsyncd
# systemctl enable --now systemd-timsyncd
# systemctl disable --now chronyd
```

systemd-timesyncd, ntpd

```
# vi /etc/systemd/timesyncd.conf
[Time]
NTP=kr.pool.ntp.org
FallbackNTP=0.pool.ntp.org 1.pool.ntp.org 0.fr.pool.ntp.org
ServerName=kr.pool.ntp.org
# systemctl enable --now systemd-timesyncd
# systemctl disable --now chronyd
# timedatectl timesync-status
       Server: 121.174.142.82 (kr.pool.ntp.org)
Poll interval: 1min 4s (min: 32s; max 34min 8s)
         Leap: normal
      Version: 4
      Stratum: 3
```

NTP서버(chronyd)

내부적으로 NTP서버를 구축하기 위해서 "chronyd"를 사용한다.

```
# vi /etc/chrony.conf
server ntp.lab.int iburst
allow 192.168.0.0/24
driftfile /var/lib/chrony/drift
makestep 1.0 3
rtcsync
keyfile /etc/chrony.keys
leapsectz right/UTC
logdir /var/log/chrony
# firewall-cmd --add-service=ntp
# firewall-cmd --runtime-to-permanent
```

systemd-timers

예약작업

crond/at

현재 시스템은 20년 넘게 "at", "crontab","anacrond"기반으로 예약 작업이 관리가 되었다. 기존 레거시 기반에서는 문제가 없지만, systemd에서는 통합이 어려웠다. 이러한 이유로 "at", "crontab"은 레거시 소프트웨어로 설정이 되었다.

레드햇 및 다른 리눅스 계열 배포판에서 여전히 사용이 가능하다. 다만, 많은 기능이 systemd의 ".timer"자원으로 변환이 되었다.

```
# crontab -l
# crontab -e -u
# atq batch
```

생성된 작업 리소스는 "/var/spool/cron/", "/var/spool/at"에 사용자 이름으로 작업이 생성이 된다. 이 생성된 내용을 crond, at 프로그램이 모니터링 하여, 정해진 시간에 작업을 수행한다.

timer

레드햇 리눅스 기준, 앞으로 RHEL 8,9부터는 crond대신 systemd-timer기반으로 자원 구성을 권장한다. 현재, 이전에 사용하였던 예약된 시스템 작업들은 대다수가 systemd의 timer기반으로 이전이 되었다.

```
# systemctl list-timers | awk '{ print $14 }'
dnf-makecache.service
logrotate.timer
systemd-tmpfiles-clean.service
```

.timer example

이전 "crond"처럼 "timer"기반으로 설정 및 사용이 가능하다.

```
# vi /etc/systemd/system/test.timer
[Unit]
Description=test timer as crond
Requires=test.service

[Timer]
Unit=test.service
OnCalendar=*-*-* *:*:00

[Install]
WantedBy=timers.target
```

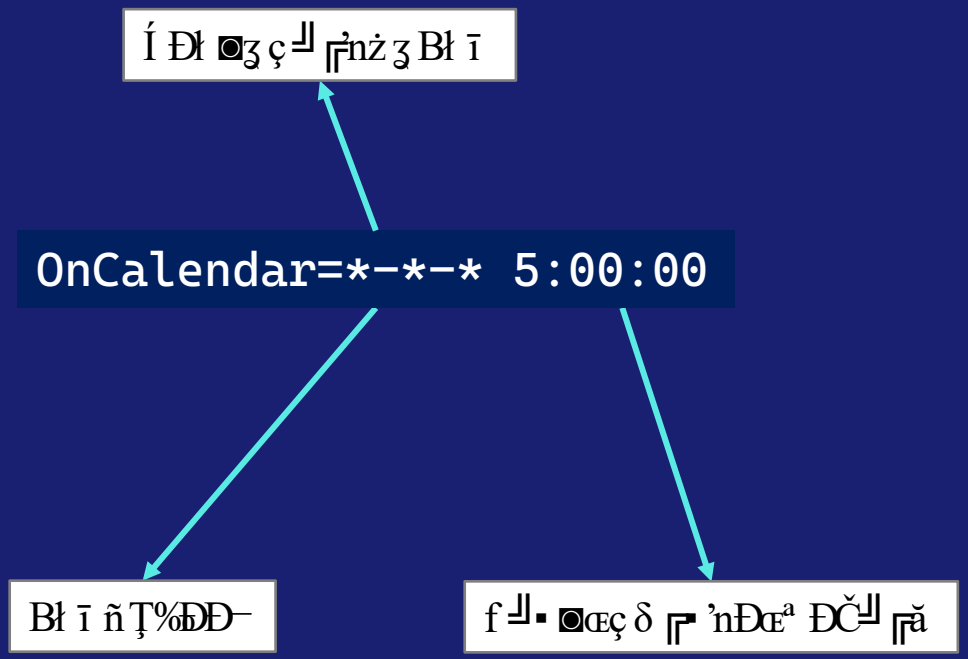
.timer example

타이머 기반으로 예약 작업을 구성하면 아래와 같이 설정 파일을 작성한다.

```
# vi /etc/systemd/system/test.service
[Unit]
Description=test service

[Service]
Type=oneshot
ExecStart=/usr/bin/free

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

systemd-timers 사양

상세하게 작업일정을 설정하기 위해서 "OnCalendar"옵션 사용이 가능하다. OnCalendar에 사용이 가능한 옵션은 아래와 같다.



systemd-runs

일시 예약 작업

at

기존에 사용하던 'at'명령어는 아래처럼 사용한다.

```
# dnf install at
# rpm -ql at
# systemctl status atd
# systemctl start atd
# date
# at 14:10
warning: commands will be executed using /bin/sh
at> echo "it's at!" > /tmp/at.txt
job 2 at Fri Nov 11 15:27:00 2022
# atq
# at -l
# ls -l /var/spool/at
a0000101b39196  spool
```

systemd-run

특정 명령어 및 특정 서비스를 임시적으로 스케줄링에 등록하기 위해서 아래와 같이 사용이 가능하다. 뒤에 단위가 붙어있지 않으면 보통은 **초**(sec)단위로 동작한다.

앞으로 systemd에서는 이전에 사용한 "job(at)"를 ".timer"로 대체한다.

```
# systemd-run --on-active=30 /bin/touch /tmp/ihatesystemd
# systemd-run --on-active="30m" --unit vsftpd.service
# systemd-run -p IOWeight=10 updatedb
# systemd-run --on-boot=1800 --on-unit-active=1800 /usr/bin/free
# systemd-run --on-boot=1800 --on-unit-active=1800 -s vsftpd.service
# systemctl list-times
```

systemd-tmpfiles

임시파일 제거

systemd-tmpfiles

이전에 crond, tmpfiles를 통해서 관리 하였던 임시 파일 관리자 기능이 systemd로 통합이 되었다. 임시 파일 관리자는 서비스 대몬 형태로 동작한다.

```
# systemctl status systemd-tmpfiles-clean.service
○ systemd-tmpfiles-clean.service - Cleanup of Temporary Directories
     Loaded: loaded (/usr/lib/systemd/system/systemd-tmpfiles-clean.service; static)
     Active: inactive (dead) since Thu 2024-03-28 11:06:21 KST; 3h 59min ago

# systemd-tmpfiles --cat-config
# /usr/lib/tmpfiles.d/criu.conf
d /run/criu 0755 root root -

# /usr/lib/tmpfiles.d/cryptsetup.conf
d /run/cryptsetup 0700 root root -
```

systemd-tmpfiles

임시 파일을 설정 관리 및 구성하기 위해서 다음과 같이 파일을 생성 및 관리한다. 모든 내용은 다 다루기에 어렵기 때문에 몇가지 대표적인 내용만 다룬다. 전체 내용은 아래 링크에서 확인한다.

- "d"는 해당 디렉터리가 존재하지 않으면 생성 및 구성한다.
- "D"는 해당 디렉터리가 존재하면, 내부 파일을 전부 삭제한다.

```
# vi /etc/tmpfiles.d/test.conf
d /run/test 1755 root root 30d
# systemd-tmpfiles --create
# systemd-tmpfiles --clean

# /etc/tmpfiles.d/abrt.conf
D /run/test 1755 root root -
# systemd-tmpfiles --create
# systemd-tmpfiles --clean → --remove
```

systemd-mount

블록장치 마운트

systemd-mount

이 부분은 아래 시스템 영역에서 다룬다. 이 기능은 이전에 사용하던 "/etc/fstab"기능을 대신하는 영역이다. 이를 통해서 "mount" 및 "autofs"와 같은 기능을 대신할 수 있다.

간단하게 **마운트 유닛 사용법(일반 블록)을** 다루어 본다.

```
# systemctl edit mnt-sdb.mount --force --full
[Unit]
Description=sdb

[Mount]
What=/dev/sdb
Where=/mnt/sdb
Type=xfs
Options=rw,noatime

[Install]
WantedBy=multi-user.target
# systemctl status sdb.mount
```

systemd-mount

이 부분은 아래 시스템 영역에서 다룬다. 이 기능은 이전에 사용하던 "/etc/fstab"기능을 대신하는 영역이다. 이를 통해서 "mount" 및 "autofs"와 같은 기능을 대신할 수 있다.

간단하게 **마운트 유닛 사용법(NFS)을** 다루어 본다.

```
# systemctl edit nfs.mount --force --full
[Unit]
Description=nfs

[Mount]
What=$YOUR_SERVER:/$YOUR_SHARE
Where=$YOUR_MOUNT_PATH
Type=nfs
Options=_netdev,auto

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

시스템 관리

서비스 관리(systemctl)

SELinux/Audit

디스크 관리(swap, vdo, lvm2)

로그관리

서비스 관리

systemd service

서비스 관리

systemd는 서비스 관리를 위해서 'systemctl'명령어를 통해서 관리한다. 이전 'service'명령어는 여전히 지원하지만, 호환성으로 존재하며 'service'명령어를 실행하면 'systemctl'로 재실행 한다.

모든 시스템 블록 자원은 "systemd"로 통합이 되고 있기 때문에, 'systemctl'명령어에 대한 이해도가 낮으면, 운영 및 장애처리에 많은 문제가 발생한다.

또한, **'systemctl'**명령어 이외에도 다른 대표적인 **'journalctl'**명령어에 대해서도 학습이 필요하다.

'systemctl'의 하위 명령어에 대해서 간단하게 살펴 본다.

서비스 관리

systemd에서 서비스 관리 시 제일 많이 사용하는 기본 명령어이다.

명령어	설명	예제
status	systemd유닛 상태 확인	systemctl status sshd
restart	유닛 재시작	systemctl restart httpd
start	유닛 시작	systemctl start httpd
stop	유닛 중지	systemctl stop httpd
reload	유닛 메모리 갱신	systemctl reload httpd
is-active	유닛 동작 상태 확인	systemctl is-active httpd
is-enabled	유닛 부트-업 확인	systemctl is-enabled httpd

서비스 관리

유닛 상태 확인 및 자원 상태에 대해서 확인하는 명령어이다. 밑에 두개 명령어는 예외처리가 들어간 명령어이다.

명령어	설명	예제
list-units	서비스 유닛 목록을 확인한다.	systemctl list-units -t mount
list-automounts	자동 마운트 장치 목록을 확인한다.	systemctl list-automounts
list-unit-files	시스템에 등록된 유닛 파일 목록을 출력한다.	systemctl list-unit-files
list-sockets	서비스가 사용하는 소켓 목록을 출력한다.	systemctl list-sockets
list-timers	타이머 서비스 목록을 출력한다.	systemctl list-timers
try-restart	재시작이 가능한지 시도한다.	systemctl try-restart httpd
reload-or-restart	재시작 혹은 메모리 갱신 시도를 한다.	systemctl reload-or-restart vsftpd
try-reload-or-restart	메모리 갱신 혹은 재시작 한다.	systemctl try-reload-or-restart vsftpd

서비스 관리

아래 명령어는 프로세서 및 네임스페이스 관리 명령어.

명령어	설명	예제
clean	시스템에 기록된 상태 정보를 초기화한다.	systemctl clean sshd
freeze	서비스를 임시로 중지한다.	systemctl freeze httpd
set-property	프로세스의 cgroup에 자원 우선 순위를 선언한다.	systemctl set-property foobar.service CPUWeight=200
bind	특정 파일 및 디렉터리를 네임스페이스 장치에 할당한다.	
mount-image	컨테이너나 혹은 가상머신 이미지를 네임스페이스에 연결한다.	systemctl mount-image --mkdir bar.service /tmp/img.raw /var/lib/baz/img

서비스 관리

로깅 및 프로세서가 어떠한 유닛에서 관리되는지 확인하는 명령어.

명령어	설명	예제
service-log-level	특정 서비스에 대해서 로깅 수준을 설정한다.	systemctl service-log-level httpd 2
service-log-target	특정 타겟에 대해서 로깅 수준을 설정한다.	systemctl service-log-target multi-user 2
reset-failed	유닛에 기록된 "실패"기록을 초기화 한다.	systemctl reset-failed sshd.socket
whoami	systemd 254버전부터 지원한다. 어떤 프로세스가 어떤 유닛으로 구성되었는지 확인한다.	systemctl whoami $(pgrep sshd | head -1)

서비스 관리

명령어	설명	예제
reenable	특정 유닛 혹은 유닛들에 대해서 'disable', 'enable'를 동시에 수행한다.	systemctl reenable sshd
preset	유닛의 기본값을 확인한다.	systemctl preset httpd
preset-all	모든 유닛에 대해서 기본값을 확인한다.	systemctl preset-all
mask	유닛이 동작되지 않도록 마스크 한다.	systemctl mask httpd
unmask	마스크된 유닛을 활성화한다.	systemctl unmask httpd
revert	벤더 기본값으로 유닛 설정을 변경한다.	systemctl revert httpd
edit	유닛 파일을 수정 합니다. 기본값은 'nano'로 동작 합니다.	systemctl edit httpd
list-machines	시스템에 구성된 컨테이너 머신을 확인한다.	systemctl list-machines
is-system-running	호스트 컴퓨터의 상태를 확인한다.	systemctl is-system-running

SELINUX/ADUIT

시스템

SELINUX

SELinux는 미국 NSA에서 제작 후, 오픈소스 커뮤니티에 기여. 정확히는 레드햇이 해당 소스코드를 받았으며, 이 코드 기반으로 커널 기반의 MAC보안 시스템을 구성하였음. 기존 리눅스 시스템은 DAC만 지원 및 구성하였기 때문에, 미국 NIST기준에 맞지 않았다.

- DAC: Discretionary Access Control
- MAC: Mandatory Access Control

대다수 리눅스 시스템은 MAC 둘 중 하나를 사용하고 있다.

AppArmor

상대적으로 SELinux보다 사용하기 쉬우며, 대다수 GNU배포판은 이를 채택하고 있다. 레드햇 계열 배포판은 "AppArmor"사용이 어렵다.

SELinux

레드햇 계열 및 컨테이너 시스템에서 많이 채용하고 있다. 커널 빌트인 기반으로 동작하기 때문에 사용이 복잡하다.

SELINUX

SELinux사용 상태 확인.

```
# getenforce
Enforcing
# setenforce 1
```

1: selinux 일시적으로 사용

0: selinux 일시적으로 중지

일시적으로 SELinux 사용 상태를 중지 혹은 사용으로 변경.

SELINUX

부팅 시, SELinux적용 상태를 변경하기 위해서는 아래를 수정 혹은 명령어를 실행한다.

```
# vi /etc/selinux/config
SELINUX=enforcing
SELINUXTYPE=targeted
# grubby --update-kernel ALL --args selinux=1
# grubby --update-kernel ALL --remove-args selinux
```

1. enforcing: 강제로 SELinux 정책 적용.
2. permissive: 감사만 하며, 정책은 적용하지 않음.
3. targeted: 프로세스 중심으로 정책 적용.
4. mls: 다중 계층 보안으로, 각각 등급별로 접근하는 영역을 다르게 한다.
5. minimum: 특정 프로세스만 검사한다. 일반적으로 컨테이너 시스템에 권장한다.

SEMANAGE

SELinux에서 사용하는 모든 컨텍스트에 대해서 관리 및 수정이 가능.

```
# semanage fcontext -l | grep httpd
# semanage fcontext -a -t httpd_sys_content_t  '/srv/htdocs(/.*)?'
# semanage port -l
# semanage port -a -t http_port_t -p tcp 81
# semanage boolean -m --on httpd_can_sendmail
# semanage boolean -l -C
```

일반적으로 보통 -l 옵션은 "list"이다. -C 옵션은 "Customized" 옵션이다. 사용자가 수정하거나 혹은 변경한 부분만 출력한다. -a는 "add" selinux policy파일에 정책을 추가한다.

BOOLEAN

프로그램에서 사용하는 기능을 허용 및 제한하는 기능이다. 프로그램에 기능이 활성화 되어도, Boolean으로 차단이 되면, 올바르게 사용이 불가능 하다.

getsebool: 프로그램에서 사용하는 특정 기능(콜) 목록을 확인.

```
# getsebool xdm_write_home
xdm_write_home ⟶ off
```

BOOLEAN

특정 기능을 사용 혹은 미사용 할지 수정 명령어. 이 명령어는 프로그램의 시스템 콜을 제한 및 제어한다.

```
# setsebool -P xdm_write_home=1
# setsebool xdm_write_home=1
```

변경된 내용에 대해서 확인하기 위해서는 다음과 같이 실행한다.

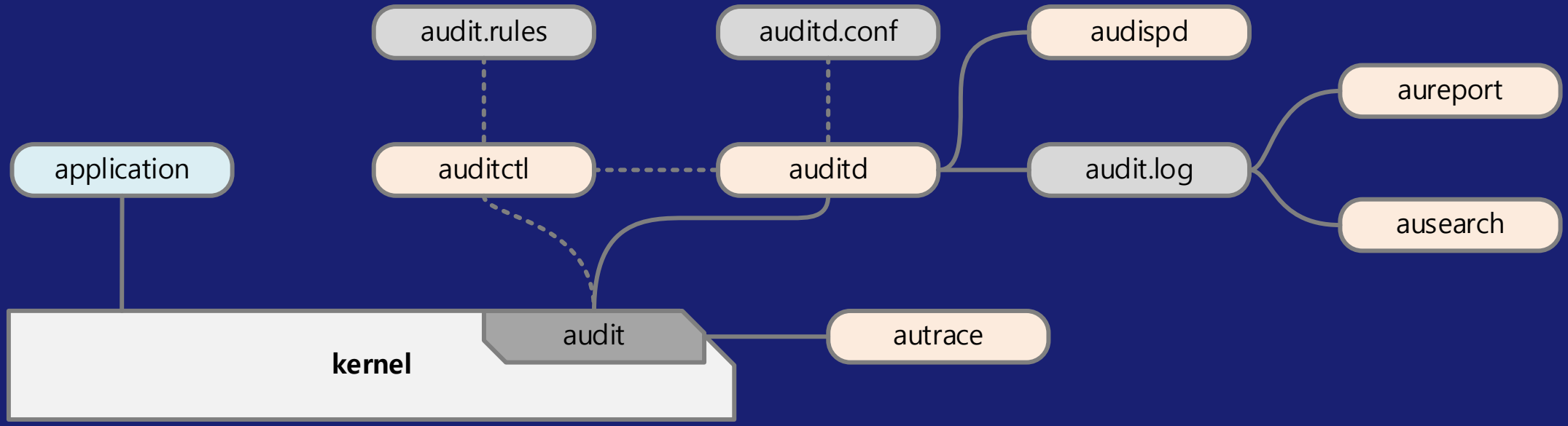
```
# semanage boolean -lC
```

`-l: list`

`-C: Customized`

AUDIT

"auditd"는 시스템에서 발생하는 시스템 콜, 사용자, 파일 및 디렉터리 같은 자원에 대해서 감사한다.



AUDIT

특정 프로그램에서 실행하는 모든 콜에 대해서 등록 후 확인한다.

```
# auditctl -a exit,always -S all -F pid=1001
```

특정사용자가 파일 접근(open)에 대해서 등록 후 확인한다.

```
# auditctl -a exit,always -S open -F auid=1001
```

성공적으로 파일을 접근한 콜에 대해서 기록한다.

```
# auditctl -a exit,always -S open -F success=0
```

옵션	설명
-a	콜 액션, exit, always 프로그램 종료, 사용 중일때 콜 기록을 남긴다.
-S	콜 이름. "open"경우에는 파일에 읽기 접근 시 기록을 남긴다. 모든 콜에 대해서 기록이 필요한 경우 "all"로 한다.
-F	조건 필터. 명시한 조건에 따라서 기록을 남긴다.
exit, always	종료 및 모든 콜 이벤트 기록.

AUDIT

특정 파일에 변경사항 확인하기

```
# auditctl -w /etc/shadow -p wa
# auditctl -a exit,always -F path=/etc/shadow -F perm=wa
```

디렉터리 퍼미션 확인 및 감사

```
# auditctl -w /etc/ -p wa
# auditctl -a exit,always -F dir=/etc/ -F perm=wa
```

옵션	설명
-w	감사할 대상자원을 명시한다. 보통 파일이나 디렉터리.
-p	퍼미션. "rwxa"로 구별해서 적는다. "a"는 "attribute"이다.
-F	조건 필터. 명시한 조건에 따라서 기록을 남긴다.
wa	쓰기 및 속성 기록.

AUDIT(search/report)

발생한 이벤트에 대해서 **검색 및 확인**하기 위해서 다음과 같이 조회가 가능하다. 또한, 파일 접근에 대한 **보고서 생성**도 아래 명령어로 가능하다.

```
# ausearch --pid $(pgrep sshd | head -1)
# ausearch -ua 1000 -i
# ausearch --start yesterday --end now -m SYSCALL -sv no -i
# aureport --start 03/20/2024 00:00:00 --end 03/21/2013 00:00:00
# aureport -x
# aureport -x --summary
# aureport -u --failed --summary -i
# aureport --login --summary -i
# ausearch --start today --loginuid 1000 --raw | aureport -f --summary
# aureport -t
```

디스크 관리

일반정보

LVM2/STRATIS/VDO

디스크 관리 명령어

┌♂ć ┌╴

블록 장치 목록을 확인하는 명령어. -o옵션을 통해서 장치의 모델명 및 마운트 정보 확인이 가능하다.

```
# lsblk -o PATH,SIZE,RO,TYPE,MOUNTPOINT,UUID,MODEL
```

ć ┌╴δă

슈퍼 블록에서 생성된 UUID정보를 확인한다.

```
# blkid -i /dev/sda
```

Ţăδ♂╴

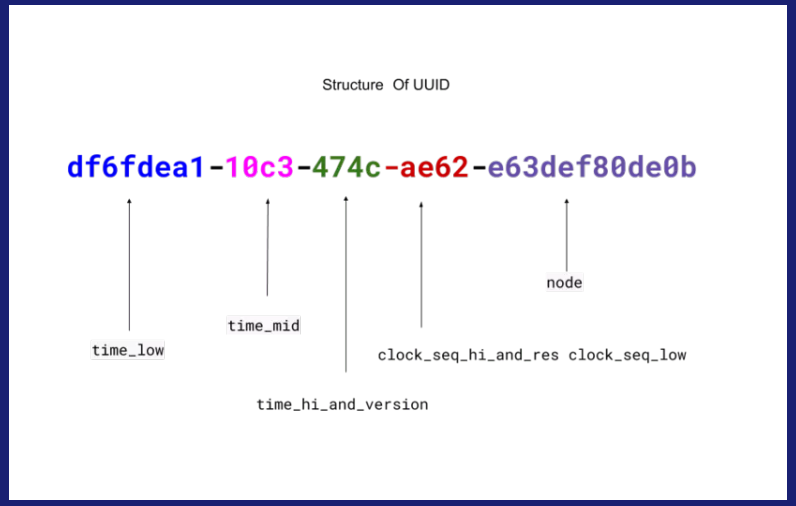
MBR형태로 디스크에 파티션을 생성한다. 프라이머리는 최대 4개까지 생성, 확장 파티션을 운영체제 별로 다르기는 하지만 보통 32개까지 가능하다. x86에서는 더 이상 MBR형태로 파티션 구성은 권장하지 않는다.

blkid

슈퍼 블록에 기록이 되어 있는 파티션 혹은 파티션 정보를 읽어와서 UUID정보를 출력한다.

여기서 말하는 UUID는 Universal Unique Identifier의 약자이며, 128비트로 구성이 되어 있는 정보이다. 참고로 UUID는 GUID(Global Unique Identifier)에서 유례가 되었다. 현재 사용중인 UUID는 버전 5이다. 소스코드를 보면 다음처럼 UUID생성 값 설정이 되어 있다.

https://github.com/torvalds/linux/blob/master/lib/uuid.c



blkid

PARTUUID

"part-uuid"는 시스템에서 사용하고 있는 디스크 혹은 드라이브의 UUID. 이 정보는 "DM(Device Mapper)"로 생성이 되며, 보통 "/dev/disk"에서 소프트 링크로 구성이 되어 있다.

DISKUUID

"disk-uuid"는 일반적으로 "/etc/fstab"에 등록된 내용. 보통 특정 파티션에 링크가 되어 있다. 위의 심볼릭 링크는 /dev/disk에서 확인이 가능하다.

참고로 위의 "partuuid", "diskuuid"와 같은 정보들은 "/dev/disk/by-*"에 형식별 장치 이름이 구성되어 있다. 이 정보들은 또한 명령어로 확인이 가능하다.

```
# udevadm info /dev/sdb
# dmsetup ls
```

디스크 관리 명령어

ţ ă δ♂

EFI(GPT)형태로 디스크에 파티션을 생성한다. 최대 128개까지 생성이 가능하다. 현재 대다수 리눅스는 GPT기반으로 구성한다. 'gdisk'명령어는 CLI모드를 지원하지 않는다.

◀◘n◀┘ć Đ

추가된 디스크를 커널의 비트맵(bitmap)에 갱신한다. 다만, 사용 중에 갱신하는 경우, 잠깐 I/O가 중지가 될 수 있다.

```
# partprobe -d -s /dev/sdb
# partprobe /dev/sdb
```

디스크 관리 명령어

◀ o'nī

'partx'는 'kpartx'와 비슷하지만, 'partx'는 일반 블록장치에 사용한다. 'partprobe'는 커널에 모든 블록장치 영역을 조회하지만, 'partx'는 특정 블록 장치 및 파티션만 커널 메모리에서 수정한다. 또한, 'partx'는 이미지 디스크 관리도 지원한다.

```
# partx --show - /dev/sdb1
# partx --add --nr 3:5 /dev/sdd       ## 파티션 3에서 5번까지 메모리에서 추가
# partx --delete --nr :-1 /dev/sdd    ## 맨 마지막 파티션만 메모리에서 삭제
# partx --add /dev/sdd                ## 모든 디스크의 파티션 정보를 메모리애 추가
```

디스크 관리 명령어

⌐◀ o'nĪ

단일 디스크 혹은 파티션을 추가 및 삭제하는 명령어.

보통 멀티패스로 구성된 디스크를 디바이스 **맵퍼(devicemapper)**에 추가 시 사용하며, 일반 디스크 추가에는 사용하지 않는다. 다만, 인식에 문제가 있으면 'partprobe'명령어를 같이 사용한다.

```
# kpartx -av /dev/mapper/mpathb
```

멀티패스로 올바르게 인식이 되지 않으면, 보통 다음과 같이 명령어를 수행한다.

```
# kpartx -pp /dev/mapper/mpathb
# partprobe /dev/mapper/mapthb
# kpartx -av /dev/dm-7
```

디스크 관리 명령어

◀ o'nĐă

CLI기반으로 디스크의 파티션 관리를 한다. 기존의 'fdisk', 'gdisk'는 대화형이기 때문에 자동화 하기에는 적절하지 않는 도구이다.

```
# parted /dev/mapper/mpathb print
# parted /dev/mapper/mpathb mklable gpt
# parted /dev/mapper/mpathb1 --align opt mkpart data 1 100%
```

"xfs", "ext4"로 50:50으로 구성하기 위해서는 다음과 같이 명령어를 수행한다.

```
# parted /dev/sdb --align opt mkpart xfs 1 50%
# parted /dev/sdb --align opt mkpart ext4 51% 100%
```

디스크 관리 명령어

◀ o'nĐă

파티션에 이름을 설정 혹은 출력하기 위해서 다음처럼 명령어를 실행한다.

```
# parted /dev/sdb name 1 data-part
# parted /dev/sdb print
```

디스크 관리 명령어

♂Ţă δ♂

여러 디스크 정보 확인이나 혹은 조작하기 위해서 사용하는 명령어. 보통은 백업이나 정보 확인 시, 사용한다.

```
# sfdisk -Z /dev/sdb
# sfdisk -n 0:0:+200M -t 0:ef02 -c 0:"bios_boot" /dev/sdb
# sfdisk -n 0:0:+1G -t 0:8300 -c 0:"linux_boot" /dev/sdb
# sfdisk -n 0:0:+1G -t 0:8200 -c 0:"linux_swap" /dev/sdb
# sfdisk -p /dev/sdb
```

혹은 파티션 편집하기 전에 복구를 위해서 다음처럼 실행하여 백업이 가능하다.

```
# sfdisk -d /dev/sdb > sdb.backup
# sfdisk -f /dev/sdb < sdb.backup
# sgdisk --backup=/tmp/gpt.backup /dev/sdb
# sgdisk --load-backup=/tmp/gtp.backup
```

디스크 관리 명령어

ČŢăδ♂

TUI기반으로 사용이 가능한 텍스트 에디터. 이 도구는 "EFI", "MBR" 둘 다 지원한다.

żĥδ ╔Ţ╝

하드웨어 정보를 확인하는 명령어. 블록 장치 정보 확인을 원하는 경우 다음과 같다.

```
# hwinfo --block --short
```

부트-업 블록장치

리눅스 배포판은 아직까지 "/etc/fstab" 혹은 **커널모듈 변경**, 관리 및 연결할 블록 장치, 파티션을 관리한다. 다만, systemd로 변경이 되면서, "fstab"정보는 **램-디스크(ramdisk)**에 저장되기 때문에, 꼭 램 디스크를 갱신해야 한

```
# dracut --list
# dracut -m systemd-initrd --force
# lsinitrd -m
# lsinitrd
# dracut -f
# systemctl reload
```

가급적이면, 램 디스크는 전부 갱신하는 게 제일 안전하다.

블록장치 관리

systemd에서 마운트 정보는 다음처럼 확인이 가능하다.

```
# systemd-mount --list
# systemd-mount -u
```

앞으로 systemd에서는 "/etc/fstab"를 사용하지 않고, "systemd-mount"를 통해서 시스템 블록 장치를 연결 및 구성을 한다. 이 자원은 이미 사용하고 있으며, 추후에는 더 이상 "/etc/fstab"를 사용하지 않는다.

```
# systemctl list-unit-files --type mount
```

기존에 사용하고 있는 "/etc/fstab"의 정보는 'systemd-fstab-generator'를 통해서 유닛 파일을 생성 및 관리 한다.

디스크 이름 명명

리눅스는 다음과 같은 디스크 네이밍 규칙을 가지고 있다.

1. virtual disk: /dev/vd?
2. USB/SCSI/External disk: /dev/sd?
3. IDE disk: /dev/hd?
4. cdrom: /dev/sr0

파티션은 보통 다음과 같은 네이밍 규칙을 가지고 있다.

```
/dev/sd[a-z][a-z][1-15]
```

리눅스 SCSI는 최대 16개의 **마이너 숫자(miner number mapped to a single disk)**까지 사용이 가능하다. 메이저 숫자도 마이너와 비슷하게 최대 16개까지 가능하다.

일반적으로 디스크 넘버링은 1부터 시작하며, 파티션 넘버링은 보통 0부터 시작한다.

디스크 이름 생성

리눅스 배포판 종류마다 각기 다른 형식으로 블록장치 생성 및 관리 개수는 다르지만, 일반적으로 128개 정도의 장치를 생성한다.

수세나 혹은 다른 배포판은 16개 혹은 30개 정도 생성하며, 'mknod', 'makedev'같은 명령어로 추가적으로 생성이 가능하다. 커널에서 인식이 된 블록 장치들은 "/proc/partitions", "/dev/disk/"에서 장치 정보들이 생성 및 공유가 된다.

디스크에 관련된 커널 파라메터는 아래에서 조정이 가능하다.

1. `/sys/class/scsi_host/`
2. `/sys/block`

udev

사용자 영역(userspace)에서 동작하는 소프트웨어이다. "/dev/"에 등록되어 있는 장치는 사용자가 사용하기 위해서는 장치를 확인 후, 블록 장치를 파일 시스템에 마운트 한다.

하지만, 점점 많은 장치가 시스템에 연결 및 구성이 되면서 시스템 사용자가 사용에 불편함을 느꼈으며, 이를 해결하기 위해서 **"udev"**를 구성하였다.

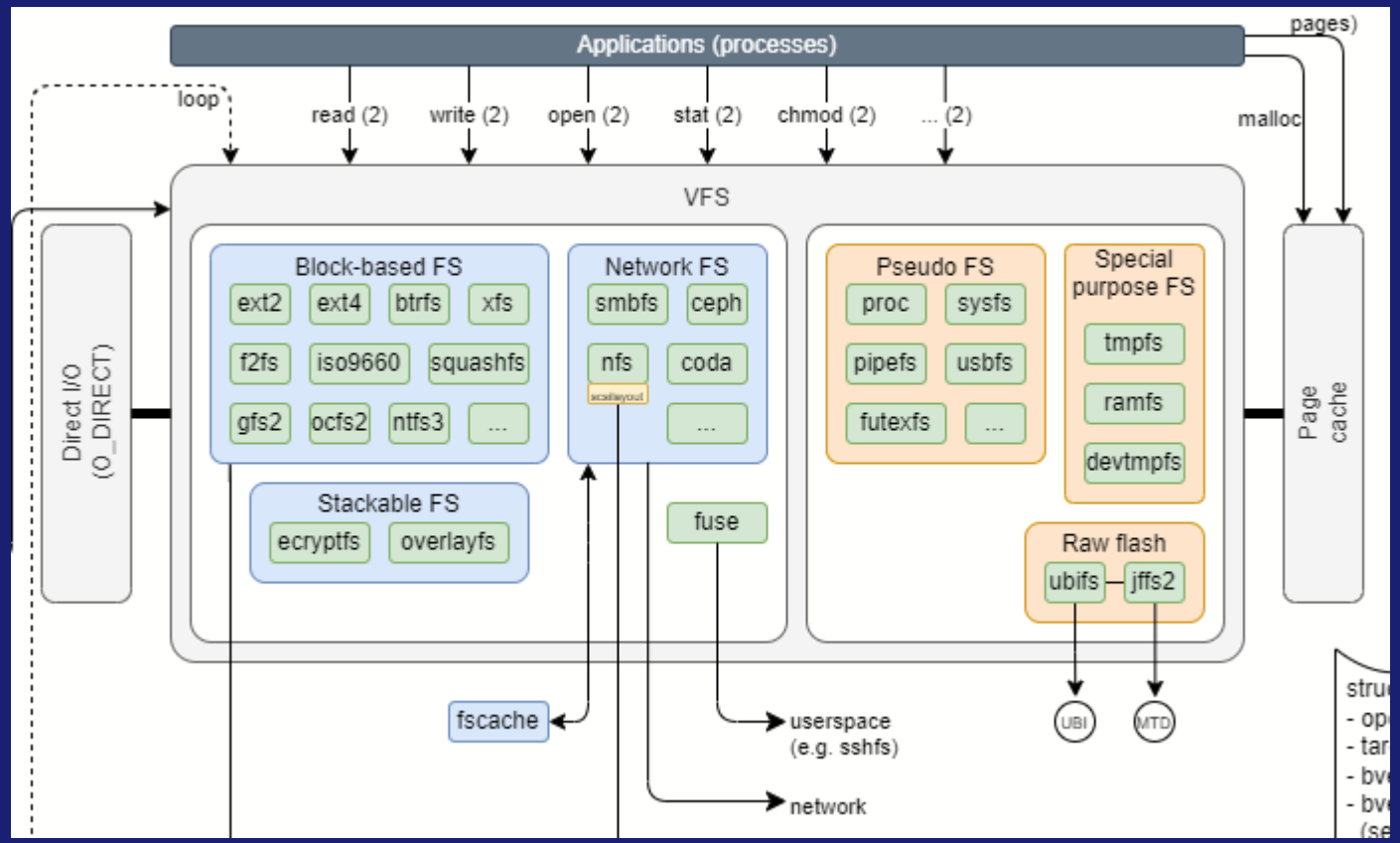
현재 "udev"는 systemd 블록에 통합이 되어서 모든 대다수 리눅스 배포판은 "udev"기반으로 구성한다. "udev"의 최대 장점은 **사용자가 규칙(rule)**기반으로 손쉽게 작성 및 구성이 가능하다.

```
# udevadm info /sys/class/net/eth0
```

이 기능은 커널 2.5에 도입이 되었으며, 완성형은 2.6버전이 넘어가면서 완전히 커널 및 시스템과 통합이 이루어 졌다. "udev"는 하드웨어 정보가 필요하기 때문에 "hwdb"기반으로 하드웨어 정보 및 장치 이름을 결정한다.

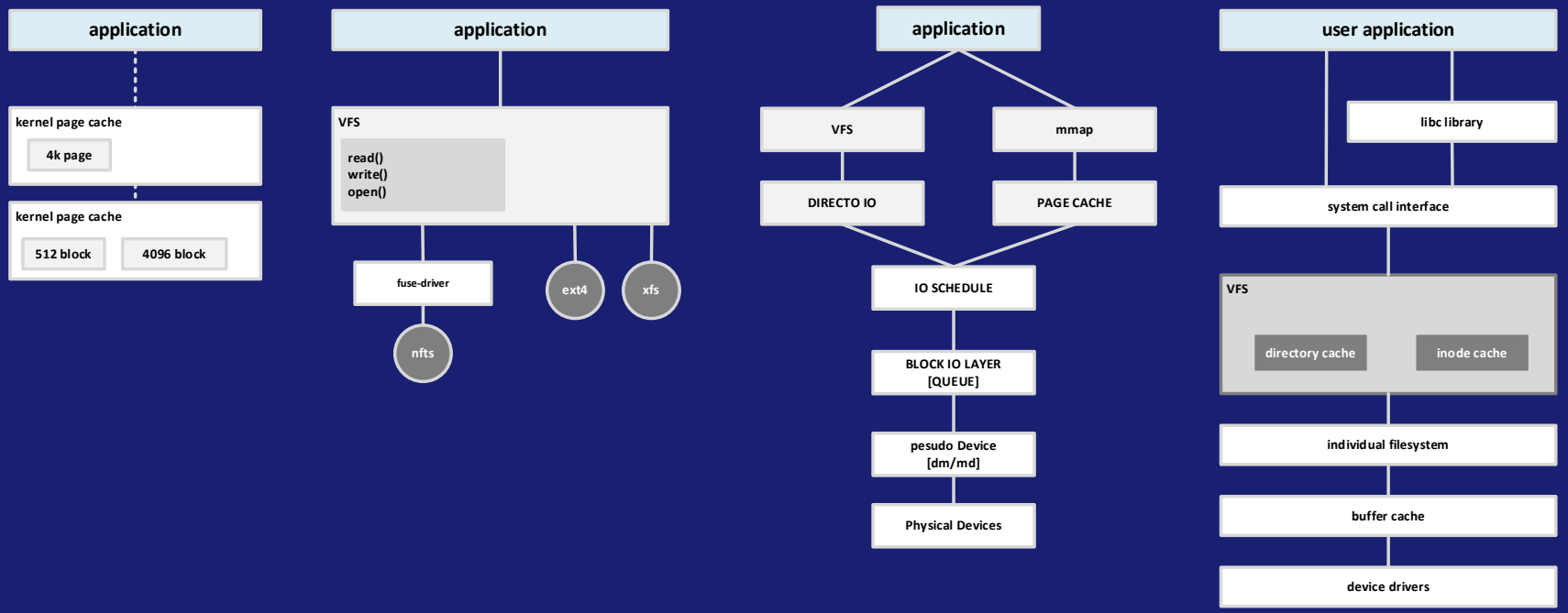
```
# /usr/bin/systemd-hwdb query mouse:*:name:*Trackball*:*
```

리눅스 블록 구조


Applications (processes)
pages)
loop
read (2)
write (2)
open (2)
stat (2)
chmod (2)
... (2)
malloc
VFS
Direct I/O (O_DIRECT)
Block-based FS
ext2
ext4
btrfs
xfs
f2fs
iso9660
squashfs
gfs2
ocfs2
ntfs3
...
Network FS
smbfs
ceph
nfs
coda
...
Pseudo FS
proc
sysfs
pipefs
usbfs
futexfs
...
Special purpose FS
tmpfs
ramfs
devtmpfs
Raw flash
ubifs
jffs2
Stackable FS
ecryptfs
overlayfs
fuse
Page cache
fscache
userspace (e.g. sshfs)
network
UBI
MTD

VFS

VFS는 Virtual File System의 약자이다. 가상 파일 시스템이 아니라, 여러가지의 파일 시스템을 통일된 레이어에서 제공하기 때문에 사용자가 파일 시스템 명시가 필요 없이 접근이 가능하다. 또한, 메모리 버퍼 기능을 제공하기 때문에 이를 통해서 성능 향상도 가능하다.



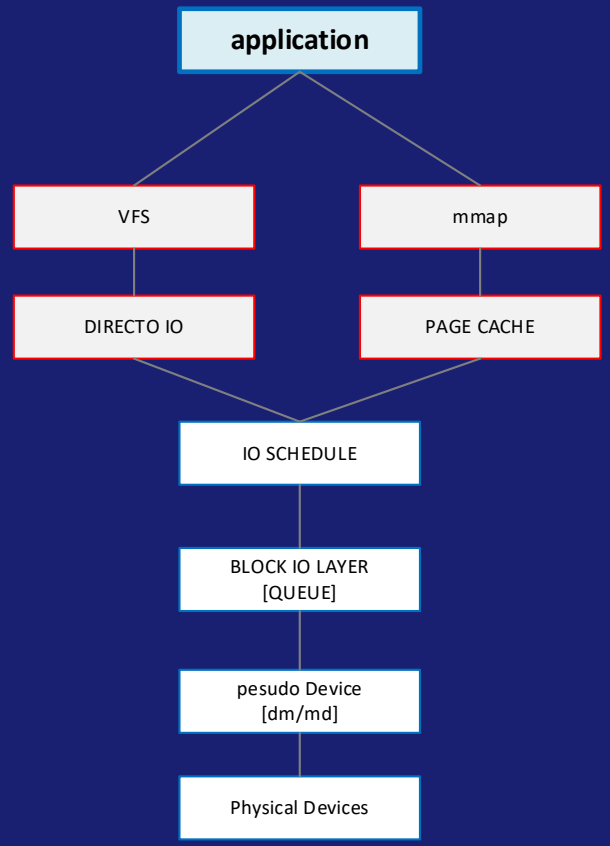
디바이스 맵퍼

커널에서 인식된 장치를 직접적으로 사용하는 경우, 장치 이름을 손쉽게 확인이 어렵지 않다. 또한, 많은 장치를 추가하는 경우 관리 하기가 어려운 부분이 있다.

그래서 고수준 레벨(Higher Level Block Device Manager)로 장치를 관리할 수 있다. 보통 "Devicemapper"는 **"DM"**이라고 부르기도 하며, **"LVM2/VDO/Stratis"**에서 블록장치 백-엔드 관리자로 사용한다.

디바이스 맵퍼는 소프트웨어 계층과 연결이 가능하기 때문에, 디스크 암호가 필요한 경우, DM를 통해서 디스크 전체나 혹은 파티션을 암호화하여 사용이 가능하다.

VFS는 디바이스 맵퍼에 하위 레이어로 통합이 되었다.



devicemapper function

- **cache:** cache를 통해서 하이브리드 디스크를 생성한다. 예를 들어서 SSD, HDD디스크를 통시에 사용한다. SSD는 캐시 디스크, HDD데이터 디스크 이러한 형태로 구성이 가능하다.
- **clone:** 디스크에 쓰기(전송이 완료전)에 사용이 가능하도록 합니다. 보통 레이드 구성 시 많이 사용한다.
- **crypt:** 리눅스 커널의 Crypto API를 사용하여 암호화 한다.
- **delay:** 각각 장치에 읽기/쓰기를 다르게 설정한다. 실제 서비스에서는 사용하지 않고, 테스트 용도로 사용한다.
- **error:** 블록 장치에 일부로 I/O장애를 발생 한다.

devicemapper function

- **linear:** DM에 블록 장치를 연결 합니다. 선형 장치를 통해서 정해진 범위 안에서 블록 장치를 구성합니다.
- **mirror:** 논리 장치를 맵(maps)를 통해서 미러링 합니다. 이를 통해서 이중화를 구성 합니다.
- **multipath:** 멀티패스(multipath)장치를 패스 그룹을 통해서 구성 및 맵핑 합니다.
- **raid:** 리눅스에서 소프트웨어 레이드 기반으로 레이드 구성 시 사용할 md장치를 제공 합니다.
- **snapshot and snapshot-origin:** LVM2에서 스냅샷을 생성 및 구성합니다. LVM2에서 사용하는 스냅샷은 COW기반으로 생성 및 구성이 됩니다.
- **striped:** 청크 크기(chunk size)기반으로 LVM2의 물리적 장치에 데이터를 분배해서 저장 합니다.
- **thin:** 쓰기가 발생 시, 물리적 블록 장치를 실제로 사용 합니다.
- **zero:** "/dev/zero"와 동일한 기능. 블록 장치에서 사용하지 않는 데이터를 쓰기를 하지 않는다.

devicemapper application

현재 DM는 다음과 같은 백-엔드와 같이 사용이 가능하다.

- **cryptsetup:** 디스크 및 파티션에 암호화가 필요한 경우, dm-crypt를 통해서 암호화 한다.
- **dm-crypt/LUKS:** 암호화된 장치를 luks를 통해서 관리 및 구성한다.
- **dm-cache:** 하이브리드 볼륨을 구성 시 사용한다.
- **dm-integrity:** lukus나 혹은 레이드 같은 장치에 대한 상태 확인 시 사용한다.
- **dm-log-writes:** DM를 통해서 구성한 장치들의 로그를 DM를 통해서 로그를 기록한다.
- **dm-verity:** DM를 통해서 구성된 블록장치에 있는 암호화 정보 혹은 무결성 정보를 확인 시 사용한다.

devicemapper

- **dmraid(8):** DM를 통해서 가짜 RAID디스크를 구성한다. 소프트웨어 레이드하고 비슷한 구성이다
- **DM Multipath:** DM 기반으로 MPath(multipath)를 구성한다. 이를 통해서 채널이나 혹은 로드밸런싱을 구성하여 안전하게 시스템이 동작 할 수 있도록 한다.
- **Docker/Podmam(container):** COW기반으로 컨테이너 이미지 및 컨테이너 데이터를 관리한다.
- **DRBD:** Ceph스토리지에서 사용하는 DRBD(Distributed Replicated Block Device)를 지원한다.
- **EVMS (deprecated):** 더 이상 사용하지 않음.
- **kpartx(8):** kpartx를 통해서 DM에 추가된 장치나 혹은 파티션 정보를 추가 및 제거한다.
- **LVM2:** LVM2를 통해서 구성된 LV를 DM를 통해서 사용할 수 있도록 한다.
- **VDO:** Virtual Data Optimizer 장치를 구성. LVM2기반으로 구성 및 확장한다.

스왑

GENERAL SWAP

ZRAM SWAP

다중 스왑 및 파일 스왑

리눅스에서 사용하는 스왑(SWAP)은 보통 2가지 형태로 사용한다.

- 파티션 형태의 raw block
- 파일 형태의 file block

대다수 리눅스 배포판은 "raw block"형태를 선호한다. 하지만, 특정 상황에서 "file block"형태를 사용해야 하는 경우도 있다. 최근, 레드햇 계열 배포판은 기존에서 사용하던 **일반 스왑(block swap)**에서, **메모리 스왑(zswap)**으로 변경하고 있다.

일반 블록 스왑 구성은 대다수 엔지니어가 알고 있기 때문에, 해당 부분은 랩에서 다루지 않는다.

파일기반 스왑 구성

"file block"형태를 사용하기 위해서는 다음과 같이 사용이 가능하다.

```
# dd if=/dev/zero of=/tmp/temp_swap.img bs=1G count=1
# mkswap /tmp/temp_swap.img
# swapon /tmp/temp_swap.img
# swapon -s
```

파일기반 다중 스왑

여러 개의 블록 스왑을 사용하는 경우, 아래와 같이 "/etc/fstab"에 설정 및 구성한다.

```
# dd if=/dev/zero of=/var/spool/temp_swap.dat bs=1G count=1
# mkswap /var/spool/temp_swap.dat
# swapon /var/spool/temp_swap.dat
# vi /etc/fstab
/dev/sdd2 swap swap defaults,pri=10 0 0
/root/temp_swap.dat none swap defaults,pri=20 0 0
```

ZSWAP(일반)

만약, 메모리 형태의 스왑을 사용하고 싶은 경우, RHEL 및 CentOS 8에서는 "**zswap**"다음과 같은 명령어로 사용이 가능하다. systemd기반에서 다음과 같이 "zram"생성 및 구성한다.

https://www.kernel.org/doc/html/v5.9/admin-guide/blockdev/zram.html

"zram"를 사용하기 위해서는 반드시 사용하는 디스크는 **메모리 형태 디스크**이어야 한다.

```
# dnf install zram-generator
# cp /usr/share/doc/zram-generator/zram-generator.conf.example /etc/systemd/zram-generator.conf
# systemctl enable --now systemd-zram-setup@zram0.service
```

ZSWAP(특정위치)

혹은 특정 위치에 "zram"구성을 원하는 경우, 아래와 같이 생성 및 구성이 가능하다.

```
# vi /etc/systemd/zram-generator.conf
[zram1]
zram-size=ram/2
mount-point=/var/compressed
options=X-mount.mode=1777
```

ZSWAP(확인)

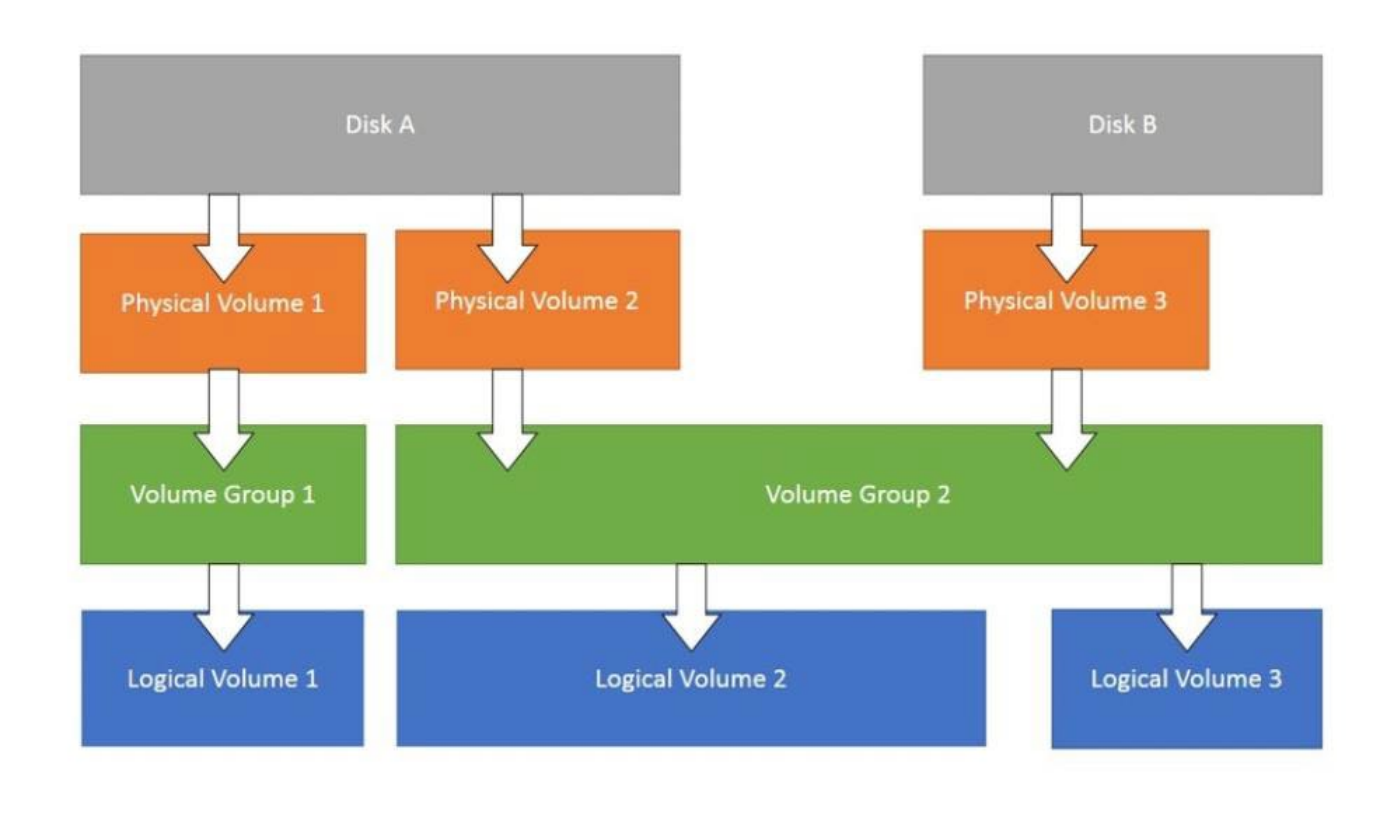
"zswap"는 기존에 사용한 스왑과 같이 사용이 가능하다. 다만, 권장은 동시 사용보다 하나의 스왑 형식을 사용하는 걸 권장한다. 이유는 "zswap"은 메모리를 사용하기 때문에 **오버헤드(overhead)**가 낮으며, **"일반 스왑"**은 블록장치를 사용하기 때문에 오버헤드가 높다.

```
# swapon -s
Filename                                Type            Size            Used            Priority
/dev/dm-1                               partition       4141052         0               -2
# zramctl /dev/zram0
NAME       ALGORITHM DISKSIZE  DATA COMPR TOTAL STREAMS MOUNTPOINT
/dev/zram0 lzo-rle       1.8G  652K 11.2K   84K       4 /var/compressed
```

LVM2

VOLUME MANAGER

LVM2


Disk A
Disk B
Physical Volume 1
Physical Volume 2
Physical Volume 3
Volume Group 1
Volume Group 2
Logical Volume 1
Logical Volume 2
Logical Volume 3

LVM2 명령어

이전에는 LVM2에서 많이 사용하던 명령어는 다음과 같다. 아래 명령어는 독립적인 명령어가 아니라, 배시 쉘에서 함수로 구현한 기능이며, 이들은 보통 심볼릭 링크로 구성이 되어있다.

```
pvcreate(/usr/sbin/pvcreate: symbolic link to lvm)
vgcreate(/usr/sbin/vgcreate: symbolic link to lvm)
lvcreate(/usr/sbin/lvcreate: symbolic link to lvm)
vgchange(/usr/sbin/vgcreate: symbolic link to lvm)
```

하지만, 위의 명령어로 LVM2 디스크를 완벽하게 관리할 수가 없기 때문에 다음과 같은 명령어로 관리 방법도 고려해야 한다. 만약, 'lvmdevices'명령어를 사용한다면, 'lvm'명령어로 다음처럼 관리가 가능하다.

```
# lvm lvmdevices
# lvm pvcreate
```

LVM2 볼륨 및 논리 장치 생성

기본적인 LVM 그룹 및 논리적 장치 생성. 아래 명령어로 생성이 가능하다. 시작 전, "hexedit"를 설치한다.

1기가 파티션 생성

```
# fdisk /dev/sdb
# pvcreate /dev/sdb1
```

VG에 1기가 전부 할당

```
# vgcreate /dev/sdc test-vg
```

논리적 디스크 생성

```
# lvcreate -n test-lv -l 100%Free test-vg
# mkfs.xfs /dev/test-vg/test-lv
# mkdir -p /mnt/test-lv
# mount /dev/test-vg/test-lv /mnt/test-lv
```

LVM2 볼륨그룹 확장

기존에 만든 "test-vg"공간을 확장한다. 먼저 "500MiB" 크기의 파티션를 추가한다.

```
# fdisk /dev/sdb2
```

추가로 만든 파티션 혹은 디스크를 test-vg에 추가한다.

```
# vgextend test-vg /dev/sdb2
```

확장된 볼륨 크기만큼 논리 디스크를 확장한다.

```
# lvextend -r -l [PE_NUMBER] -L [UNIT_SIZE] /dev/test-vg/test-lv
```

- **-r:** Resize to FS는 자동으로 수행한다. 예를 들어서 "ext4"는 'resize2fs', "xfs"는 'xfs_growfs'명령어를 사용한다.
- **-l/-L:** 두 개의 크기 옵션을 동시에 사용할 수 없다. 둘 중 하나만 사용해야 한다.

LVM2 BACKUP

LVM2 백업은 'lvm'명령어를 실행하면 자동적으로 생성이 된다. 다만, 복구 부분은 사용자가 직접 해야 한다. 복구를 하기 위해서 다음과 같이 명령어로 조회 및 복구를 하면 된다.

단, 복구 명령어를 실행하면, "/etc/lvm/backup"에 저장된 내용을 블록 장치의 블록 영역에 다시 덮어씌우기를 진행한다. 백업되는 영역은 "VolumeGroup"만 백업이 된다. "Physical Volume"영역은 x86에서는 그렇게 중요하지 않다.

```
# vgcfgbackup
# vgcfgrestore
# vgcfgrestore testvg -l
  File:          /etc/lvm/archive/testvg_00000-1010607222.vg/testvg_00000-1010607222.vg
  VG name:       testvg
  Description:   Created *before* executing 'lvcreate -n testlv -l 100%Free testvg'
  Backup Time:   Sat Mar 30 15:33:41 2024
```

LVM2 METADATA

LVM2 메타 정보는 물리적 디스크에 저장이 된다. 본래 LVM시스템은 IBM AIX에서 넘어온 시스템이기에, 하드웨어 펌웨어에 저장이 되었지만, x86시스템에서는 그럴 수 없기 때문에 블록장치의 특정 영역에 저장한다.

```
# pvcreate /dev/sdb
# vi /dev/sdb
000001F8   00 00 00 00  00 00 00 00  4C 41 42 45  4C 4F 4E 45  01 00 00 00
00 00 00 00  ........LABELONE........
00000210   73 21 33 53  20 00 00 00  4C 56 4D 32  20 30 30 31  67 58 70 45
55 31 37 45  s!3S ...LVM2 001gXpEU17E
00000228   34 67 5A 67  6C 39 7A 6D  72 74 61 4F  58 43 45 69  6C 75 54 73
61 33 6C 47  4gZgl9zmrtaOXCEiluTsa3lG
```

LVM2 METADATA

"Volume Group"를 생성하면 해당 정보를 디스크에 다음과 같이 저장한다. 해당 부분은 VG클러스터 영역 정보이다.

```
00000FF0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  BA A4 FA B9  20 4C 56
4D  ..................... LVM
00001008   32 20 78 5B  35 41 25 72  30 4E 2A 3E  01 00 00 00  00 10 00 00  00 00 00
00  2 x[5A%r0N*>............
00001020   00 F0 0F 00  00 00 00 00  00 02 00 00  00 00 00 00  59 03 00 00  00 00 00
00  ................Y.......
00001038   05 83 47 4C  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00
00  ..GL....................
```

LVM2 METADATA

VG 및 LV가 구성이 되면, 아래에 메타정보가 생성이 된다. 즉, OS에 저장이 되어 있는 "/etc/lvm"의 내용은 명령어 실행 및 설정 내용만 가지고 있으며, 실제 런타임 정보는 전부 블록 장치에 저장이 된다.

이러한 이유로, LVM2로 구성한 디스크는 블록장치가 나가면 LVM2의 설정 정보가 사라지기 때문에, "/etc/lvm/backup"를 통해서 복구를 해야 한다.

```
00001200   74 65 73 74  76 67 20 7B  0A 69 64 20  3D 20 22 65  43 79 4C 64  52 2D 41 53  testvg {.id = "eCyLdR-AS
00001218   31 75 2D 33  56 33 31 2D  30 55 6A 41  2D 6F 68 73  65 2D 4C 73  74 4D 2D 59  1u-3V31-0UdA-ohse-LstM-Y
00001230   4F 68 55 6C  5A 22 0A 73  65 71 6E 6F  20 3D 20 31  0A 66 6F 72  6D 61 74 20  OhUlZ".seqno = 1.format
00001248   3D 20 22 6C  76 6D 32 22  0A 73 74 61  74 75 73 20  3D 20 5B 22  52 45 53 49  = "lvm2".status = ["RESI
00001260   5A 45 41 42  4C 45 22 2C  20 22 52 45  41 44 22 2C  20 22 57 52  49 54 45 22  ZEABLE", "READ", "WRITE"
00001278   5D 0A 66 6C  61 67 73 20  3D 20 5B 5D  0A 65 78 74  65 6E 74 5F  73 69 7A 65  ].flags = [].extent_size
00001290   20 3D 20 38  31 39 32 0A  6D 61 78 5F  6C 76 20 3D  20 30 0A 6D  61 78 5F 70   = 8192.max_lv = 0.max_p
000012A8   76 20 3D 20  30 0A 6D 65  74 61 64 61  74 61 5F 63  6F 70 69 65  73 20 3D 20  v = 0.metadata_copies =
000012C0   30 0A 0A 70  68 79 73 69  63 61 6C 5F  76 6F 6C 75  6D 65 73 20  7B 0A 0A 70  0..physical_volumes {..p
000012D8   76 30 20 7B  0A 69 64 20  3D 20 22 67  58 70 45 55  31 2D 37 45  34 67 2D 5A  v0 {.id = "gXpEU1-7E4g-Z
000012F0   67 6C 39 2D  7A 6D 72 74  2D 61 4F 58  43 2D 45 69  6C 75 2D 54  73 61 33 6C  gl9-zmrt-aOXC-Eilu-Tsa3l
00001308   47 22 0A 64  65 76 69 63  65 20 3D 20  22 2F 64 65  76 2F 73 64  62 22 0A 0A  G".device = "/dev/sdb"..
00001320   64 65 76 69  63 65 5F 69  64 5F 74 79  70 65 20 3D  20 22 73 79  73 5F 77 77  device_id_type = "sys_ww
00001338   69 64 22 0A  64 65 76 69  63 65 5F 69  64 20 3D 20  22 6E 61 61  2E 36 30 30  id".device_id = "naa.600
00001350   32 32 34 38  30 38 37 38  31 30 61 35  66 34 62 33  39 39 34 62  36 66 30 61  2248087810a5f4b3994b6f0a
00001368   63 66 34 64  62 22 0A 73  74 61 74 75  73 20 3D 20  5B 22 41 4C  4C 4F 43 41  cf4db".status = ["ALLOCA
00001380   54 41 42 4C  45 22 5D 0A  66 6C 61 67  73 20 3D 20  5B 5D 0A 64  65 76 5F 73  TABLE"].flags = [].dev_s
00001398   69 7A 65 20  3D 20 32 30  39 37 31 35  32 30 0A 70  65 5F 73 74  61 72 74 20  ize = 20971520.pe_start
000013B0   3D 20 32 30  34 38 0A 70  65 5F 63 6F  75 6E 74 20  3D 20 32 35  35 39 0A 7D  = 2048.pe_count = 2559.}
000013C8   0A 7D 0A 0A  0A 7D 0A 23  20 47 65 6E  65 72 61 74  65 64 20 62  79 20 4C 56  .}...}.# Generated by LV
000013E0   4D 32 20 76  65 72 73 69  6F 6E 20 32  2E 30 33 2E  32 31 28 32  29 20 28 32  M2 version 2.03.21(2) (2
000013F8   30 32 33 2D  30 34 2D 32  31 29 3A 20  53 61 74 20  4D 61 72 20  33 30 20 31  023-04-21): Sat Mar 30 1
00001410   35 3A 33 31  3A 35 33 20  32 30 32 34  0A 0A 63 6F  6E 74 65 6E  74 73 20 3D  5:31:53 2024..contents =
00001428   20 22 54 65  78 74 20 46  6F 72 6D 61  74 20 56 6F  6C 75 6D 65  20 47 72 6F   "Text Format Volume Gro
00001440   75 70 22 0A  76 65 72 73  69 6F 6E 20  3D 20 31 0A  0A 64 65 73  63 72 69 70  up".version = 1..descrip
00001458   74 69 6F 6E  20 3D 20 22  57 72 69 74  65 20 66 72  6F 6D 20 76  67 63 72 65  tion = "Write from vgcre
00001470   61 74 65 20  74 65 73 74  76 67 20 2F  64 65 76 2F  73 64 62 2E  22 0A 0A 63  ate testvg /dev/sdb."..c
00001488   72 65 61 74  69 6F 6E 5F  68 6F 73 74  20 3D 20 22  74 65 73 74  2D 6C 61 62  reation_host = "test-lab
000014A0   2E 65 78 61  6D 70 6C 65  2E 63 6F 6D  22 09 23 20  4C 69 6E 75  78 20 74 65  .example.com".# Linux te
000014B8   73 74 2D 6C  61 62 2E 65  78 61 6D 70  6C 65 2E 63  6F 6D 20 35  2E 31 34 2E  st-lab.example.com 5.14.
000014D0   30 2D 33 36  32 2E 38 2E  31 2E 65 6C  39 5F 33 2E  78 38 36 5F  36 34 20 23  0-362.8.1.el9_3.x86_64 #
000014E8   31 20 53 4D  50 20 50 52  45 45 4D 50  54 5F 44 59  4E 41 4D 49  43 20 54 75  1 SMP PREEMPT_DYNAMIC Tu
00001500   65 20 4E 6F  76 20 37 20  31 34 3A 35  34 3A 32 32  20 45 53 54  20 32 30 32  e Nov 7 14:54:22 EST 202
00001518   33 20 78 38  36 5F 36 34  0A 63 72 65  61 74 69 6F  6E 5F 74 69  6D 65 20 3D  3 x86_64.creation_time =
00001530   20 31 37 31  31 37 38 30  33 31 33 09  23 20 53 61  74 20 4D 61  72 20 33 30   1711780313.# Sat Mar 30
00001548   20 31 35 3A  33 31 3A 35  33 20 32 30  32 34 0A 0A  00 00 00 00  00 00 00 00   15:31:53 2024.........
```

LVM2의 미래

LVM2는 모든 리눅스 배포판에서 계속 사용한다. LVM2 불편한 부분을 "DeviceMapper", "UDev"와 같은 도구와 통합이 되었기 때문에 성능 및 편의성이 많이 개선이 되었다.

하지만, 기업환경에서는 Enterprise Filesystem Feature기능을 리눅스 시스템에서 요구하기 시작하였으며, 이러한 요구로 리눅스 파운데이션은 "btrfs"를 구성하였다. 다만, "btrfs"가 본래 목적보다 성능이 많이 부족 및 자잘한 버그로 인하여 릴리즈가 늦어지자, 레드햇은 기존의 "XFS"파일 시스템을 개선을 레드햇 7버전 기준으로 가속화 하였다.

XFS 파일 시스템이 개선이 되면서, 기존 LVM2로 시스템 블록을 관리가 복잡하고, Native XFS기능을 사용이 어렵기 때문에 "Stratis"를 만들기 시작하였다. 이 기능은 레드햇 RHEL7부터 Technical Preview로 제공하였고, 현재는 RHEL 8버전 이후부터는 공식 기능으로 제공한다.

결론은, Stratis가 ROOT FILESYSTEM 영역을 통합을 진행하고 있기 때문에, LVM2는 레드햇 계열의 배포판에서는 선택적인 파일 시스템 도구로 될 예정이다.

4/1/2024

https://people.redhat.com/mskinner/rhug/q4.2017/Sandeen_Talk_2017.pdf

VDO

COMPRESSED STORAGE

진행 전 준비사항

시작하기 전에 디스크 하나를 추가한다. 랩에서 필요한 디스크는 **총 3개**의 디스크가 필요하다.

1. /dev/vdb, LVM2
2. /dev/vdc, vdo
3. /dev/vdd, Stratis

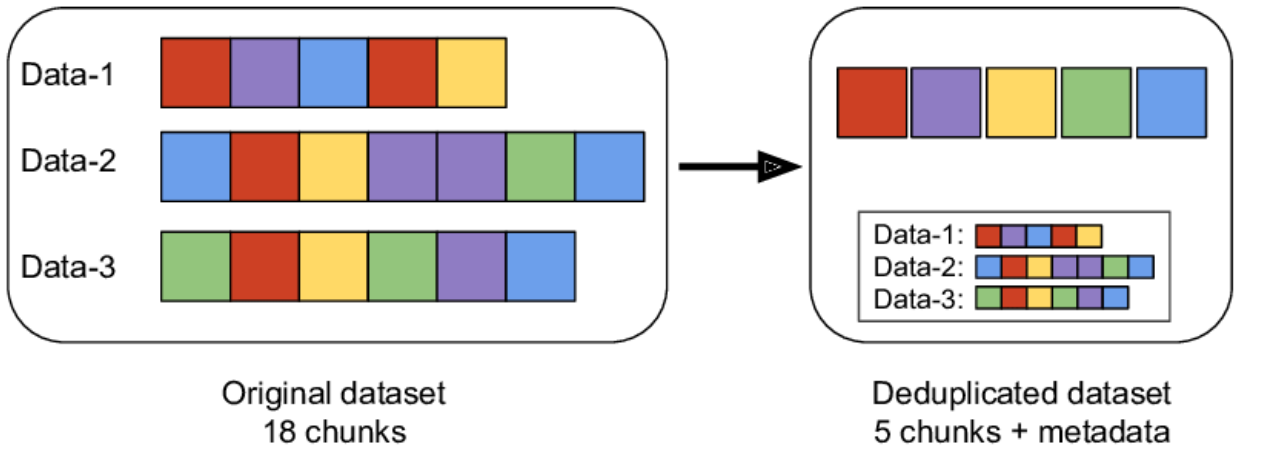
DUP/RAID/SNAPSHOT

설명	XFS	BTRFS	ZFS
압축	VDO기반에서 제공	inline / offline	inline
중복제거(Deduplication)	offline	inline / offline	inline
외부 메타데이터	네	아니요	네, 외부장치 명시 가능
읽기/쓰기 캐쉬	LVO/B-Cache 레이어	LVO/B-Cache 레이어	L2arc / slog
메모리 비트 로테이션 보호	아니요	네	네
레이드 지원 형식	아니요	Raid 1+10	Raid 1+raidz(5)+raidz2(6)+ raidz3(7)+10+50+60...
레이드 재배치 지원	-	네	아니요
중복제거 기능 끄기(조건에 따라)	네	네	아니요
가상머신/컨테이너 스냅샷	가상머신에서만 가능	가상머신/컨테이너	가상머신/컨테이너

VDO(DE-DUPLICATE OPTIMIZATION)

VDO서비스는 중복된 블록을 하나로 압축 혹은 묶어주는 블록 관리 시스템이다. 대다수 엔터프라이즈 파일 시스템은 이 기능을 제공한다. 레드햇 계열 배포판은 본래, "btrfs"으로 넘어가려고 하였으나, 성능문제로 인하여 기존 "xfs"으로 다시 사용하였다.

이러한 이유로 레드햇은 **"xfs"**에서 제공하지 않는, **디스크 볼륨 및 블록 최적화** 기능을 확장 블록장치 기능으로 제공한다. VDO는 아래와 같은 동작 구조를 사용하고 있다.



VDO

VDO(Virtual Data Optimizer)는 이전에 독립적인 서비스로 사용 하였으나, 지금은 LVM2기반으로 vdo스토리지가 구성 및 관리가 된다. 레드햇 기준으로 다음과 같이 요구사항이 필요하다. 데비안 및 다른 배포판 경우에는 "xfs"을 지원하지만, 아직 대다수는 VDO를 지원하지 않는다.

아래 배포판은 "vdo.service"가 필요하다.

- RHEL 7
- RHEL 8

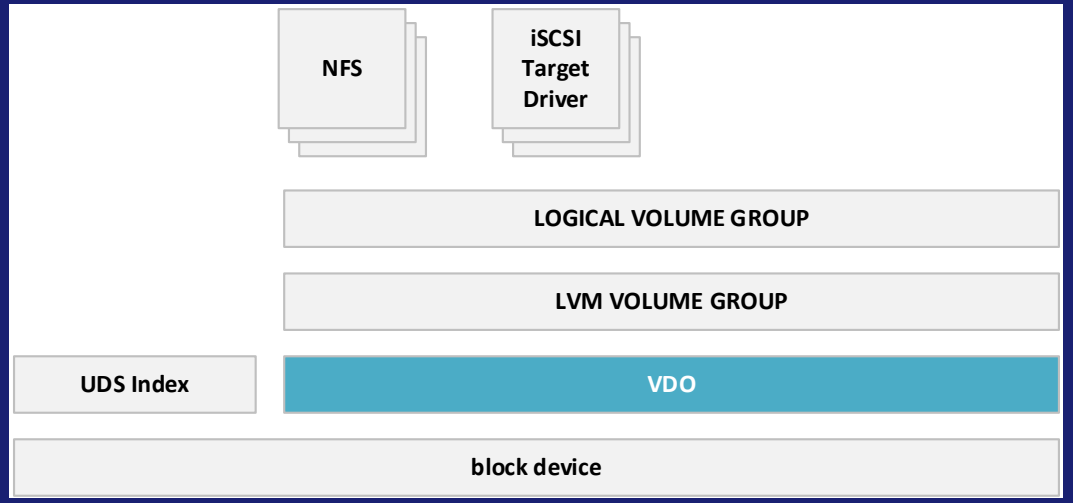
아래 배포판은 "vdo.service"가 필요하지 않는다.

- RHEL 9

VDO

VDO서비스는 가상 머신 기반에서 사용을 많이 한다. 실제로 가상머신이 아니어도 사용이 가능하며, 같은 블록 정보가 발생이 되는 경우, VDO기반으로 블록 데이터 저장을 권장한다. 가상머신의 이미지 디스크는 동일한 블록 데이터를 사용한다. 이를 "vdo.ko"모듈이 블록 장치에서 확인하여 중복된 부분을 압축한다.

현재는 레드햇 기준 RHEL 8버전 이후부터는 LVM2로 통합이 되었기 때문에, **레드햇 계열 7/8버전**과 9과는 사용방법이 다르다.



VDO(LVM2)

	물리적 장치	구성되는 장치
VDO on LVM	VDO pool LV	VDO LV
LVM thin provisioning	Thin pool	Thin volume

VDO

커널 모듈 서명이 올바르지 않아서 커널에 올라가지 못함. "Windows Hypervisor Hyper-V"를 사용하는 경우, "SecureBoot"를 비활성화 후, 수행하면 잘 됨. 최소 크기가 3기가 이상이면 동작.

"vdo"패키지는 버전에 따라서 없을 수 있음. 올바르게 "dm-vdo.ko"모듈을 찾지 못하는 경우 아래처럼, 링크를 생성. 로키 리눅스에서는 패키징 버그가 있음.

```
# dnf install vdo kmod-kvdo -y
# ln -snf ../extra/kmod-kvdo/vdo/kvdo.ko dm-vdo.ko
# depmod -a
```

VDO

VDO를 생성하기 위해서는 레드햇 계열은 8버전 이후부터 "LVM2"기반으로 생성 및 구성해야 한다.

```
# pvcreate /dev/sdb
# vgcreate vg-vdo /dev/sdb
# lvcreate --type vdo --name lv-vdo -l 100%Free vg-vdo
# vod vdostats
```

이 이후 내용은 강사의 지시에 따라서 확인 및 검증한다.

STRATIS

XFS POOL

Stratis

레드햇에서 사용하는 XFS는 POOL기능이 없다. 이러한 부분은 LVM2로 해결을 하였지만, 운영이 복잡하고 "Pool" 기능 보다는 기존 레이드 기술과 가까운 부분이 있다. 이러한 이유로, 대규모 파일 시스템에서 제공하는 "Pool"기능을 별도의 추상적인 블록영역으로 구현하였다.

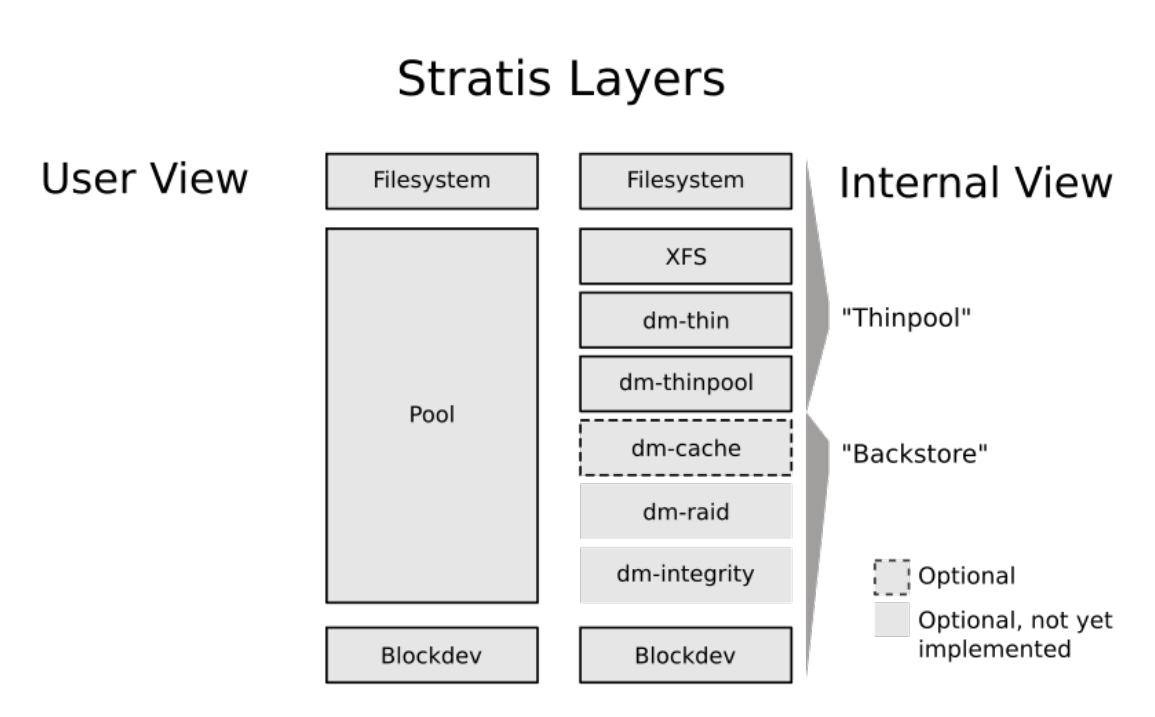
레드햇 계열 배포판에서는 Stratis라는 이름으로 제공하고 있으며, 이를 사용하기 위해서는 레드햇 계열 기준 7버전 이상을 권장한다. 또한, 파일 시스템 및 "Startisd"데몬 버전에 따라서 기능이 다르기 때문에, 가능한 최신 버전 사용을 권장하며, 8버전 이후로 상용 시스템에 사용하기가 적합하다.

현재 Startis는 ROOT FILESYSTEM영역도 적용이 가능하기 때문에, 앞으로 LVM2기반의 ROOT FILESYSTEM은 Stratis로 교체가 될 예정이다.

XFS 기능 링크

Stratis

Stratis는 기존에 사용하던 LVM2과 비슷하게 "DeviceMapper", "Udev"를 활용하여 백-엔드 구성이 되어있다.
LVM2와 미디어 레이어를 DM를 사용하지만, 사용자가 복잡하게 관리 및 구현하는 부분이 없다.



Stratis

스토리지 풀 도구를 설치한다. Stratis 는 Stratisd데몬으로 관리가 되기 때문에, 반드시 데몬 서비스가 동작이 되어야 한다.

```
# dnf search stratis
# dnf install stratisd stratisd-tools stratis-cli -y
# systemctl status stratisd
# systemctl enable --now stratisd
# stratis blockdev list
# stratis pool list
# stratis pool create
# stratis pool create firstpool /dev/sdd
```

Stratis

Stratis는 Pool생성 시, 기본으로 xfs기반으로 구성이 된다. 다른 파일 시스템으로 선택은 어려운 부분이다. 아래 명령어로 디스크를 구성하면 자연스럽게 파일시스템이 xfs으로 생성이 된다.

```
# stratis filesystem create --size 1GiB firstpool first-xfs
# stratis filesystem list
>/dev/stratis/firstpool/first-xfs
# dmsetup ls
# hexedit /dev/stratis/firstpool/first-xfs
# hexedit /dev/sdd
```

Stratis

```
# stratis pool add-data firstpool /dev/sde
# stratis pool list
firstpool   20 GiB / 610.50 MiB / 19.40 GiB   ~Ca,~Cr, Op   63b843af-0277-4751-af5e-f7f0057efc56
# stratis filesystem create --size 2GiB firstpool second-xfs
# stratis fs snapshot firstpool first-xfs snap-first-xfs
# stratis fs list
# lsblk --output=UUID /dev/stratis/test-pool/testfs
# nano /etc/fstab
UUID=<UUID> /mnt/stratis xfs defaults,x-systemd.requires=stratisd.service 0 0
```

로그

systemd-journald

journald

기존에 사용하던 "syslog(rsyslog)"를 대신하는 로깅 데몬 시스템.

제일 큰 차이점은 "journald"는 바이너리 데이터베이스 기반으로 오류 수준별로 기록을 남긴다. 아직까지는 대다수 시스템은 "rsyslogd"기반으로 구성이 되어있지만, 곧 모든 시스템은 "systemd-journald"기반으로 변경될 예정이다.

```
# systemctl status systemd-journald
# vi /etc/systemd/journald.conf
Storage=persistent
# cp -a /run/log/journal/ /var/log/
# journalctl -b
```

journalctl 명령어

systemd기반에서는 더 이상 "(r)syslog"를 사용하지 않는다. 다만, 대다수 시스템은 호환성을 위해서 syslogd를 여전히 지원하고 있다.

여전히 로그는 syslog에도 남기고 있지만, 앞으로 systemd기반에서는 journald서비스로 로그 기록을 바이너리 데이터베이스로 저장한다. 이를 사용하기 위해서는 journalctl명령어로 데이터베이스를 조회하여 유닛 및 커널 관련된 메시지 확인이 가능하다.

제일 큰 장점은 기존에 어려웠던 메시지 우선순위를 손쉽게 조회가 가능하다. 자주 사용하는 옵션은 아래와 같다.

-b	부팅 시 발생한 로그를 확인한다.
-fl	기존에 'tail -f'명령어와 동일하다.
-p	메시지 우선 순위를 필터링 합니다. err, warning, info, notice, debug와 같은 옵션을 지원한다.
-t	확인할 유닛 형식을 선택한다. 일반적으로 .service, .timer와 같이 명시한다.
-u:	유닛 이름을 명시한다.
_SYSTMED_*	systemD키워드 명령어를 통해서 자원을 조회한다. 직접 데이터베이스 필드를 선택한다.

저널 중앙서버

중앙서버 기능을 사용하기 위해서는 아래 패키지를 설치해야 한다. 구현하기 위해서 가상서버 "node1"에 구성한다. "node1"는 journald의 **서버 역할**을 한다.

```
node1]# dnf install systemd-journal-remote
node1]# mkdir -p /var/log/journal/remote
node1]# vi /etc/systemd/journal-remote.conf
SplitMode=host
node1]# vi /etc/systemd/journal-upload.conf
URL=10.10.10.1:19532
node1]# cp /lib/systemd/system/systemd-journal-remote.service /etc/systemd/system/
node1]# vi systemd-journal-remote.service
ExecStart=/usr/lib/systemd/systemd-journal-remote --listen-http=-3 --output=/var/log/journal/remote/
node1]# firewall-cmd --add-port=19532/tcp
node1]# systemctl daemon-reload
node1]# systemctl enable --now systemd-journal-upload.service systemd-journal-remote.service systemd-journal-
remote.socket
```

저널 클라이언트

클라이언트 서버 "node2"는 다음과 같이 패키지를 설치 및 구성한다.

```
node2]# dnf install systemd-journal-remote
node2]# vi /etc/systemd/journal-upload.conf
[Upload]
URL=http://10.10.10.1:19532
node2]# vi systemd-journal-remote.service
ExecStart=/usr/lib/systemd/systemd-journal-remote --listen-http=-3 --
output=/var/log/journal/remote/
node2]# systemctl daemon-reload
node2]# systemctl enable --now systemd-journal-remote
node2]# systemctl enable --now systemd-journal-upload
```

저널 로그확인

아래 명령어로 올바르게 동작하는지 확인한다. 이 명령어는 node1번에서 실행한다.

```
node1]# systemd-cat ls /ls
node1]# systemd-cat cat /etc/hostname
node2]# systemd-cat cat /etc/hostname
# journalctl --file /var/log/journal/remote/remote-10.10.10.1.journal
# journalctl --file /var/log/journal/remote/remote-10.10.10.2.journal
```

저널 서버 인증키

TLS로 전송을 원하는 경우 아래 명령어로 TLS키를 생성 후 node1/2에 배포한다. 이 교육에서는 해당 부분은 다루지 않으며, 명령어만 언급한다.

저널 서버 인증키 생성

```
# openssl req -newkey rsa:2048 -days 3650 -x509 -nodes -out ca.pem -keyout ca.key -subj '/CN=Certificate authority/'
ʁ˩Čł ʼn˩ὴ᷍Gñ] ʌ˩Čł ɴČ╝ ╔Ţ
[ ca ]
default_ca = this
[ this ]
new_certs_dir = .
certificate = ca.pem
database = ./index
private_key = ca.key
serial = ./serial
default_days = 3650
default_md = default
policy = policy_anything
[ policy_anything ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
EOF
```

4/1/2024

저널 서버 인증키 생성

아래 명령어를 순서대로 진행한다.

```
# echo 0001 >serial
# SERVER=node1.example.com
# CLIENT=node2.example.com
# openssl req -newkey rsa:2048 -nodes -out $SERVER.csr -keyout $SERVER.key -subj
"/CN=$SERVER/"
# openssl ca -batch -config ca.conf -notext -in $SERVER.csr -out $SERVER.pem
# openssl req -newkey rsa:2048 -nodes -out $CLIENT.csr -keyout $CLIENT.key -subj
"/CN=$CLIENT/"
# openssl ca -batch -config ca.conf -notext -in $CLIENT.csr -out $CLIENT.pem
```

journalctl boot

```
# journalctl --list-boots
IDX BOOT ID                          FIRST ENTRY                 LAST ENTRY
  0 e19e1af774df49ea84f3e461c71681b1 Fri 2024-03-29 08:55:01 KST Fri 2024-03-29 20:15:54 KST
# journalctl -u httpd -l -f
Mar 29 20:16:43 test-lab.example.com systemd[1]: Starting The Apache HTTP Server...
Mar 29 20:16:44 test-lab.example.com systemd[1]: Started The Apache HTTP Server.
Mar 29 20:16:44 test-lab.example.com httpd[43547]: Server configured, listening on: port
80
# journalctl --since "2023-04-17 12:00:00" --until "2023-04-18 12:00:00"
# journalctl --since yesterday -p err -p crit
-- No entries --
# journalctl --since 09:00 --until "1 hour ago"
```

journald persistent logging

```
# cp -a /run/log/journald /var/log/
# vi /etc/systemd/journald.conf
[Journal]
Storage=persistent
# systemctl restart systemd-journald
# systemctl is-active systemd-journald
# journalctl -b -1
# journalctl -b <BOOT_ID>
```

.service logging

```
# journalctl -u httpd.service -u nginx.service --since today
# journactl _PID=8080
# id -u www-data
33
# journalctl _UID=33 --since today
```

podman journald

컨테이너 런타임에서 발행한 메시지를 syslog가 아닌 journald으로 로깅하기 위해서 다음과 같이 설정한다. 도커는 아직 "journald"를 지원하지 않으며, "Podman"만 백 로깅 기능을 지원한다.

```
$ vi ~/.config/containers/containers.conf
[containers]
log_driver = "journald"

$ podman info | grep logDriver
logDriver: journald

$ podman logs <CONTAINER_NAME>
$ journalctl -f
```

journal for kernel and boot logging

journald에서 커널에서 발생한 메시지 확인하기 위해서 아래와 같이 명령어를 사용한다.

```
# journalctl -k
# journalctl -k -b -5
# journalctl -k -b -p err -p warning
# journalctl --output cat
```

journald priority

"-p"옵션을 통해서 동시에 여러 오류 우선 순위를 검색할 수 있다. 아래는 **우선순위 번호**이다.

우선 순위	우선 순위 문자 분류
0	emerg
1	alert
2	crit
3	err
4	warning
5	notice
6	info
7	debug

journalctl

별다른 수정없이 로그를 보고 싶은 경우, 다음과 같이 명령어 실행

```
# journalctl --no-full
# journalctl -a
# journalctl /usr/sbin/sshd
```

출력 방법은 변경하고 싶으면 다음과 같이 실행한다.

```
# journalctl -b -u httpd -o json
# journalctl -b -u nginx -o json-pretty
# journalctl -b -u sshd -o cat
```

journalctl

최근 메시지를 출력하고 싶으면 다음과 같이 한다.

```
# journalctl -n
# journalctl -n 20
# journalctl -f
```

로그 메시지가 얼마나 디스크 용량을 사용하는지 확인하려면 다음과 같이 한다.

```
# journalctl --disk-usage
```

로그 사이즈를 줄이기 위해서 다음과 같이 명령어를 실행한다.

```
# journalctl --vacuum-size=1G
# journalctl --vacuum-time=1years
```

서비스 관리

콘솔로깅

콘솔로그

시스템에 사용하는 콘솔에 대해서 보안 감사가 필요한 경우, "tlog"를 통해서 로깅을 한다. 이전과 많이 다른 부분은 실시간으로 사용자 화면을 녹화한다.

Cockpit기반으로 사용하기 위해서 다음처럼 설정한다.

```
# yum install -y tlog cockpit cockpit-session-recording cockpit-ws
# systemctl enable cockpit.socket --now
# systemctl enable cockpit.service --now
# systemctl status cockpit.socket
```

그냥 "tlog"만 사용하면, 아래와 같이 설치한다.

```
# dnf install tlog -y
```

콘솔로그

테스트 할 사용자 및 보안 서비스인 "sssd"설정을 변경한다.

```
# groupadd -g 10000 suspicious
# useradd hacker1
# groupmems -g suspicious -a hacker1
# vi sssd-session-recording.conf
[sssd]
services=nss, pam
domains=nssfiles

[domain/nssfiles]
id_provider=proxy
proxy_lib_name=files
proxy_pam_target=sssd-shadowutils

[session_recording]
scope=some
users=hacker1, tang
groups=adm
```

콘솔로그

아래처럼 "tlog"기록 방법에 대해서 설정이 가능하다.

```
# vi /etc/tlog/tlog-rec-session.conf
{
  "shell" : "/bin/bash",
  "notice" : "\nATTENTION! Your session is being recorded!\n\n",
  "file": {
      "path" : "/var/log/tlog/tlog.log""
    },
  "journal": {
      "priority" : "info",
      "augment" : true
    },
  "writer" : "journal"
}
```

네트워크

관리 명령어

네트워크 관리 명령어

현재 레드햇 계열 및 데비안 계열의 배포판에서는 아래와 같은 네트워크 관리 시 다음과 같은 명령어를 많이 사용한다. 아래는 기본적인 명령어.

1. ip ← ifconfig(namespace 지원안됨)
2. ss ← netstat(namespace 지원안됨)
3. ip r ← route(namespace 지원안됨)
4. nftables ← iptables(강화된 버전, 더 이상 지원하지 않음)

이 외에 관리 및 모니터링을 위한 명령어는 정말로 다양하게 있다. 모든 명령어를 다루기는 어렵지만, 최소한 아래 슬라이드와 같이 영역별로 명령어는 사용이 가능해야 한다.

기본 네트워크 관리 명령어

ip 명령어는 이전에 사용하던 'ifconfig', 'route'명령어를 대체하는 명령어 이다. 이 명령어를 통해서 네트워크 카드에 설정된 아이피 정보 확인이 가능하고 수동으로 추가가 가능하다. 사용 방법은 다음과 같다.

구성된 NIC의 아이피 정보 및 NIC상태 확인

```
# ip address show
```

여기에서 구성되는 아이피 정보는 일시적으로 시스템에서 저장. 재시작 시 해당 내용은 제거. 반드시 영구적인 네트워크 설정은 "NetworkManager", "systemd-networkd"에서 구성해야 됨.

```
# ip addresss add <DEV>
```

기본 네트워크 관리 명령어

연결이 되어 있는 NIC카드 정보.

```
# ip link
```

현재 구성이 되어있는 라우팅 테이블 정보.

```
# ip route
```

네트워크 관리 명령어

```
# ip route
default via 192.168.90.250 dev eth0 proto dhcp metric 100
192.168.90.0/24 dev eth0 proto kernel scope link src 192.168.90.226 metric 100
# dnf install net-tools -y
# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    100    0        0 eth0
10.10.10.0      0.0.0.0         255.255.255.0   U     0      0        0 eth1
192.168.0.0     0.0.0.0         255.255.255.0   U     100    0        0 eth0
```

네트워크 관리 명령어

```
# ip monitor
192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d REACHABLE
192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d STALE
192.168.90.91 dev eth0 lladdr 56:6f:08:c8:00:3d REACHABLE
# ss -antp
# tracepath 8.8.8.8 -m 4
# dnf install traceroute -y
# traceroute 8.8.8.8 -m 4
```

네트워크 관리 명령어

```
# ip a add 192.168.1.200/255.255.255.0 dev eth1
# ip -4 -brief address show
lo               UNKNOWN        127.0.0.1/8
eth0             UP             192.168.0.254/24
eth1             UP             10.10.10.2/24 192.168.1.200/24# ip a del
192.168.1.200/255.255.255.0 dev eth1
# nmcli connection show eth1
# nmcli device show eth1
```

더미 장치 생성하는 방법

네트워크 기능 테스트를 위해서 더미 장치 생성이 종종 필요한 경우가 있다. 'ip' 명령어로는 다음과 같이 더미 장치 생성이 가능하다.

```
# ip link add dummy0 type dummy
# ip link add dummy1 type dummy
# ip addr add 192.168.1.100/24 dev dummy0
# ip addr add 192.168.1.200/255.255.255.0 dev dummy1
# ip addr add 192.168.1.255 brd + dev dummy0
```

네임 스페이스 장치

네임 스페이스 장치를 확인하기 위해서 다음과 같이 명령어를 사용한다. 보통 컨테이너 디버깅 시 종종 사용한다. 간단하게 네트워크 네임 스페이스 장치를 생성한다.

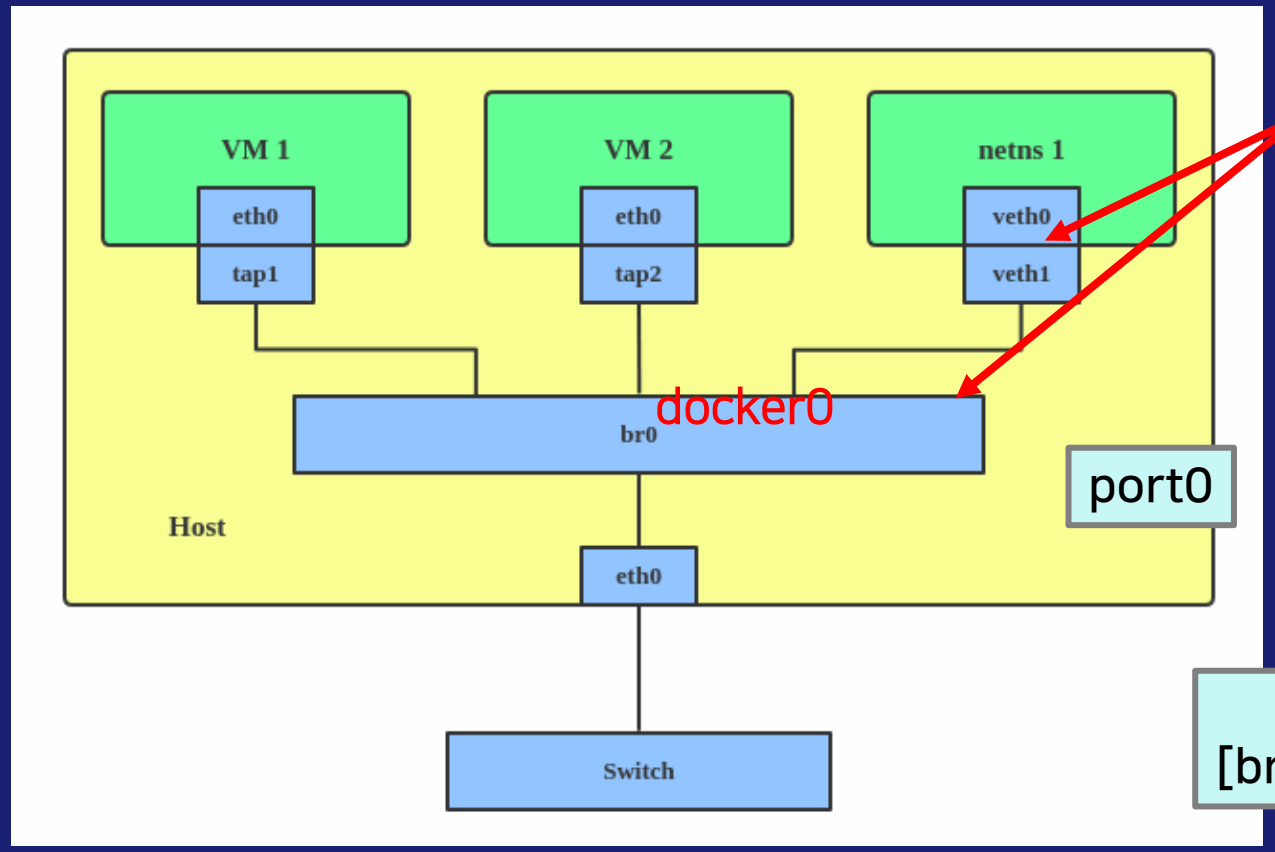
```
# ip netns add test
# ip netns list
test
# ls /var/run/netns/
test
# ip netns exec test ip link
```

네임 스페이스 장치

간단하게 'ip'명령어로 컨테이너에서 사용하는 네트워크 장치를 수동으로 생성한다.

```
# ip netns add port1
# ip netns add port2
# ip netns set port1 10
# ip netns set port2 20
# ip -n port1 netns set port1 10
# ip -n port2 netns set port2 20
# ip -n switch1 netns set switch2 20
# ip -n switch2 netns set switch1 30
# ip netns list-id target-nsid 10
# ip netns list-id target-nsid 10 nsid 20
```

컨테이너 브리지 구현 예



application

"port 0"생성

veth1

port0

port1

port2

podman0
[bridge, switch]

컨테이너 브리지 구현 예

```
# ip link add br0 type bridge
# ip link add dummy0 type dummy
# ip tuntap add mode tap tap1
# ip tuntap add mode tap tap2
# ip link add veth0 type veth peer name veth1
# ip link set eth0 master br0
# ip link set tap1 master br0
# ip link set tap2 master br0
# ip link set veth1 master br0
```

네트워크

네트워크 네이밍 및 마이그레이션

4/1/2024

네트워크 네이밍

네트워크 네이밍을 변경하기 위해서는 다음과 같이 설정들을 수정한다. 리눅스 커널이 3.x에 들어오면서 바이오스 혹은 펌웨어 기반으로 네트워크 장치 네이밍을 사용하였다. 기존에 리눅스에서 사용하던 장치 이름 방식은 보통 "eth1", "eth2:1"이와 같이 사용하였다.

```
# vi /etc/default/grub
…
GRUB_CMDLINE_LINUX="crashkernel=auto resume=/dev/mapper/cs-swap
rd.lvm.lv=cs/root rd.lvm.lv=cs/swap biosdevname=0 net.ifnames=0"
…
# grub2-mkconfig -o /etc/grub2.cfg
```

네트워크 네이밍

혹은 아래 방법으로 전환 가능.

```
# ln -s /dev/null /etc/udev/rules.d/80-net-name-slot.rules
```

위와 같이 하는 경우, 더 이상 Dell Naming를 사용하지 않고 기존 방식으로 사용.

dell bios naming

네트워크 네이밍

systemd-networkd를 통해서 장치 이름 변경이 가능하다.

가급적이면 "bisosdevname", "net.ifnames"옵션을 가급적이면 사용 비권장. 위의 옵션으로 변경하는 경우, 재시작 후, 네트워크 서비스가 올바르게 동작되지 않는 경우가 많다.

```
# udevadm info /sys/class/net/
# udevadm info /sys/class/net/eth0
# vi /etc/systemd/network/eth1.link
[Match]
OriginalName=eth1
MACAddress=00:15:5d:44:6f:ac

[Link]
AlternativeNamesPolicy=
AlternativeName=new-eth1
# ip link property add dev eth1 altname internal
# ip link show eth1
```

ifcfg-rh

현재 대다수 리눅스 배포판은 다음과 같은 네트워크 스크립트를 사용하였다.(혹은, 계속 사용 중)

- `ifcfg-rh`
- `ifcfg-suse`
- `ifcfg-general`

위와 같은 네트워크 관리 스크립트를 사용하였고, 기본 스크립트 기반으로 각각 배포판은 다른 방식으로 설정내용 및 위치를 가지고 있었다. 이러한 이유로 관리가 매우

ifcfg-rh

현재 대다수 리눅스 배포판은 다음과 같은 네트워크 스크립트를 사용하였다.(혹은, 계속 사용 중)

- `ifcfg-rh`
- `ifcfg-suse`
- `ifcfg-general`

위와 같은 네트워크 관리 스크립트를 사용하였고, 기본 스크립트 기반으로 각각 배포판은 다른 방식으로 설정내용 및 위치를 가지고 있었다. 이러한 이유로 관리가 매우

네트워크

NetworkManager

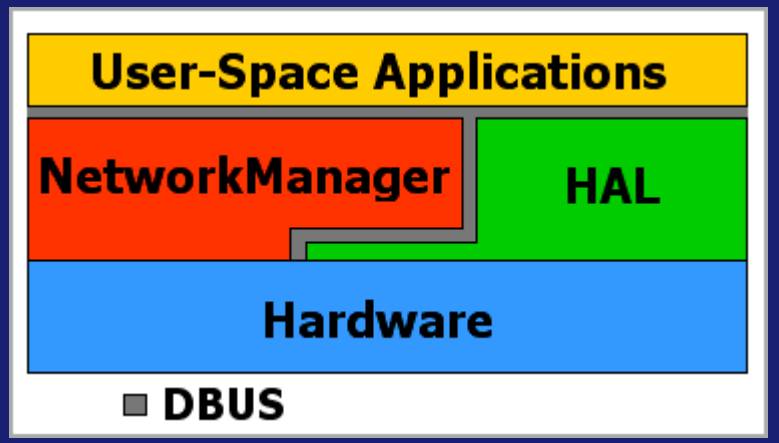
네트워크매니저

네트워크 매니저는 **레드햇 계열은 RHEL 7/8/9**에서 지원한다. 다만, **"RHEL 9"**에서는 **"7/8"**과 다르게 더 이상 **"ifcfg-*"**를 지원하지 않는다.

1. 모든 네트워크 파일은 INI형태로 "/etc/NetworkManager/"에 저장 및 관리.
2. 관리 파일은 'NetworkManager --print-config'명령어로 확인.
3. NetworkManager는 "systemd-networkd"로 대처 및 통합될 예정.

이전에 버전에서 사용하였던 **네트워크 매니저(RHEL기존 7이후)**와 현재 사용하는 네트워크 매너지와는 호환이 되지 않는다.

NetworkManager



네트워크매니저

네트워크 매니저를 사용하기 위한 명령어는 아래와 같다.

```
ʁㄟ♂ī ♂'nĐ╓Č'n ┌ㄟ♂'nɬ ◙'nㄟéĐ'nĥ╝◙─çɬ ╒ţ Đ◙
ʁㄟ╒╓Č ┌δ
ʁㄟ╒╓'n▪ δ
ʁㄟ╒╓ʒ Č╝ ╒╓ĐČ'nδ╝ ╒ʒ Đă δ'n╝◙
```

네트워크매니저

기존에 사용하던 network configuration은 init기반의 "shell scrip(/etc/init.d/)"로 되어 있었다. 보통 유닉스 스크립트는 "/etc/sysconfig/network-scripts"와 "/etc/sysconfig/network"를 통해서 관리 하였다. 현재는 RHEL 7이후로는 "Network Manager"기반으로 변경.

RHEL 7에서는 호환성을 위해서 스크립트 플러그인을 지원하고 있다.

하지만, RHEL 8버전 이후로는 네트워크 스크립트는 선택 사항이며, RHEL 9 이후로는 더 이상 스크립트는 지원하지 않는다. 레드햇 리눅스 기반 리눅스 배포판은 네트워크 매니저를 기본으로 사용한다.

네트워크 매니저 관리 명령어

- nmtui
- nm-connection-editor
- nmcli
- /etc/sysconfig/network-scripts/

네트워크매니저

nmtui

TUI기반으로 네트워크 설정한다. 자동화 용도로 사용하기는 어렵다.

```
# nmtui edit eth0
# nmtui connect eth0
# nmtui hostname
```

nm-connection-editor

엑스 윈도우 기반으로 네트워크 설정 변경. 시스템 관리자는 거의 사용하지 않는 도구이다.

네트워크매니저

nmcli

CLI기반으로 네트워크 인터페이스 변경. 쉽지는 않지만, 자동화나 혹은 반복적으로 수정 시 도움이 된다. 아래 내용은 RHEL 9기반에서 사용하는 NetworkManager설정 내용이다. 더 이상 네트워크 매니저는 "ifcfg-rh"를 사용 및 생성하지 않는다.

```
[main]
# plugins=
# rc-manager=auto
# migrate-ifcfg-rh=false
```

/etc/sysconfig/network-scripts/

RHEL 7/8까지는 지원. RHEL 9부터는 더 이상 지원하지 않는다.

네트워크매니저

nmcli

네트워크 매니저는 설정 파일이 "프로파일(profile)"기반으로 구성이 된다. 모든 정보는 INI형태로 저장이 되며, 이를 통해서 네트워크 매니저 엔진이 인터페이스 카드에 설정 및 구성한다.

```
# nmcli con add con-name eth1 ipv4.addresses 10.10.1.1/24 ipv4.gateway 10.10.1.250
ipv4.dns 10.10.1.250 ipv4.method manual ifname eth1 type ethernet
# nmcli con mod eth1 ipv4.addresses 10.10.1.2/24
# nmcli con up eth1
# nmcli con sh eth1 -e ipv4.addresses -e ipv4.gateway -e ipv4.dns
# nmcli con del eth1
```

네트워크매니저 기존 방식

네트워크 매니저에서 기존 방식으로 다시 사용하기 위해서 다음과 같은 과정이 필요하다. 다만, 아래 과정은 레드햇 계열 9버전부터는 아래 방법으로 사용을 권장하지 않는다.

```
# vi /etc/NetworkManager/NetworkManager.conf
[main]
plugins=keyfile,ifcfg-rh
# systemctl restart NetworkManager
# nmcli connection migrate --plugin ifcfg-rh
# ls -l /etc/sysconfig/network-scripts/
ifcfg-eth0  ifcfg-eth1  readme-ifcfg-rh.txt
# vi ifcfg-eth1
IPADDR=10.10.10.1 -> 10.10.10.2
# nmcli con reload
# nmcli con down eth1 && nmcli con up eth1
```

네트워크

systemd-networkd

systemd-networkd

앞으로 모든 리눅스 배포판은 systemd기반으로 통합이 된다. 현재는 그 작업이 진행중이다. 시스템 운영에 주요 핵심 자원인 **네트워크 영역**은 **"systemd-networkd"**이다. 이를 통해서 네트워크 인터페이스 설정 파일 및 디바이스 관리를 지원한다. 기본 구성 파일은 INI 및 TOML형식을 지원한다.

네트워크 매니저 경우에는 'nmcli'가 관리 명령어처럼, "systemd-networkd"는 'networkctl'이 관리 명령어 이다.

다만, "networkd"는 설정 파일을 수동으로 작성해야 한다. 아래는 간단하게 관리하는 명령어이다.

```
# networkctl list
# networkctl status eth0
# systemctl status systemd-networkd
# systemctl is-active systemd-networkd
```

systemd-networkd

테스트용으로 설정파일을 다음처럼 생성한다. 생성 후 "systemd-networkd"로 인터페이스를 활성화 한다.

```
# /etc/systemd/network/10-static.network
[Match]
Name=eth1
[Network]
Address=10.10.10.1/24
Gateway=10.10.10.254
DNS=10.10.10.254
```

```
# networkctl list
# networkctl reload
# networkctl up eth1
# networkctl status eth1
```

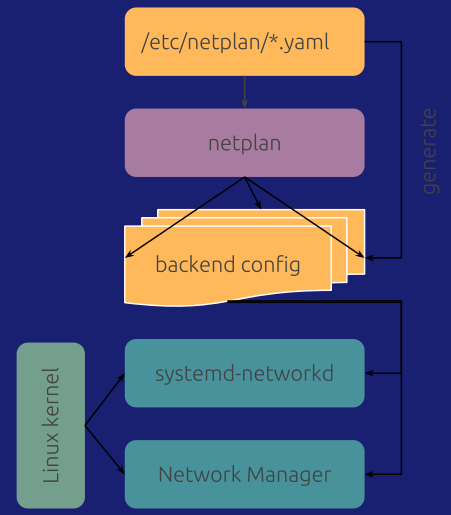
네트워크

Netplan

netplan

netplan은 "systemd-networkd" 및 "NetworkManager"를 백-앤드로 사용하는 컨트롤 플레인 도구다. 이를 통해서 앞서 이야기 하였던 두 개의 도구를 손쉽게 백-앤드로 구성해서 사용한다.

다만, 레드햇 계열 배포판에서는 **네트워크매니저가** 기본이며, **데비안 계열**에서는 "netplan"기반으로 **"네트워크 매니저"** 및 "systemd-network"를 지원한다.



netplan

```
# dnf install netplan -y

# vi /etc/netplan/eth1.yaml
```

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      addresses:
        - 10.10.10.2/24
      nameservers:
        search:
          - "mycompany.local"
        addresses:
          - 10.10.10.253
          - 8.8.8.8
```

netplan

```
# netplan apply
# netplan get
# cat /etc/resolv.conf
# ip a s eth0 | grep inet
```

netplan

레드햇 계열 배포판은 netplan를 기본으로 사용하지 않는다. 만약, 현재 사용중인 레드햇 계열 배포판에서 백엔드를 어떠한 도구로 사용해야 될지 고민이 된다면, "netplan"를 권장한다.

다만, 이 도구는 고급 기능, 예를 들어서 "teamd", "InfiniBand"와 같은 기능이 지원하지 않는다. 고급 기능이 필요한 경우, NetworkManager나 systemd-networkd를 직접 구성해서 사용을 권장한다.

네트워크

firewalld

nftables

nftables

기존에 사용하던 iptables는 대용량의 컨테이너 및 가상머신 운영에는 적절하지 않았다. 그래서 기존에 사용하던 {ip,ip6,arp,eb}tables명령어를 새로운 "in-kernel packet farmwork"로 변경하였다.

새로운 테이블(nftables) 프로그램은 기존에 사용하던 Netfilter의 정책도 호환이 가능하며, 인프라에 구성된 NAT와 그리고 사용자 영역의 큐잉(queuing)과 로그 기능을 하위 시스템으로 제공한다.

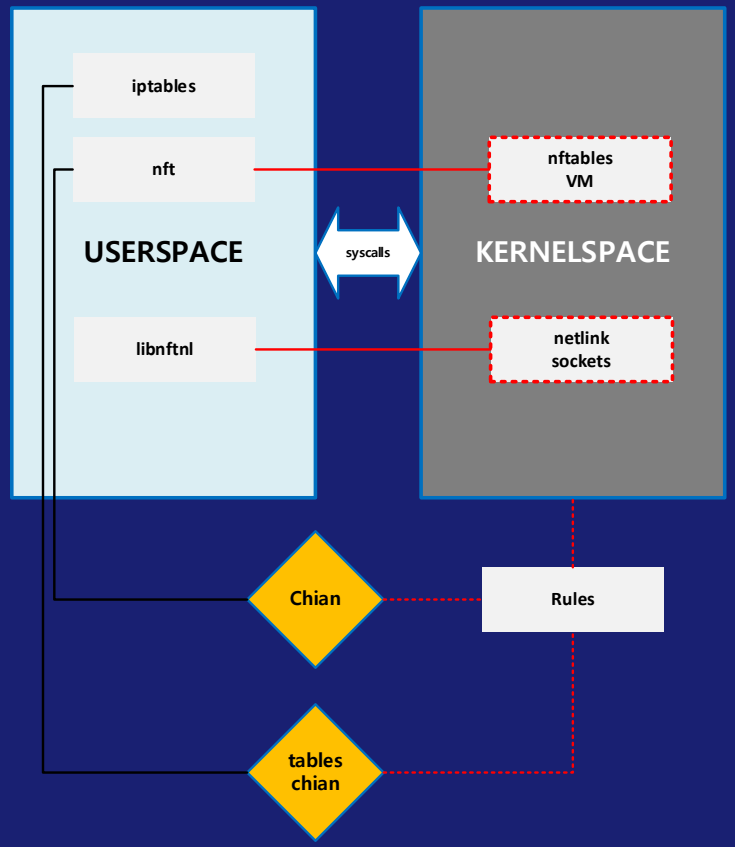
이전 iptables(text file)와 다른 부분은 nftables(JSON)기반으로 정책파일을 관리한다. 그래서 이전 테이블보다 빠르게 입출력 및 검색이 가능하다.

nftables

nftables는 VM(Virtual Machine)를 가지고 있다. 자바의 JVM과 마찬가지로 nft-vm를 가지고 있다. 사용자가 선언한 내용은 바이트 코드 형태로 컴파일이 되며, 이 기반으로 netlink를 구성 및 생성한다.

nft는 Netlink API를 사용하여, 이를 기반으로 커널에 작업을 수행한다. 결론적으로는, nft는 컴파일러 및 디 컴파일러가 있으며, 데이터는 JSON기반으로 되어 있다.

nftables


iptables
nft
USERSPACE
libnftnl
syscalls
nftables
VM
KERNELSPACE
netlink
sockets
Chian
Rules
tables
chian

nftables

여전히 호환성 모드로 iptables 사용은 가능하나 가급적이면 nftables기반으로 작업 권장.

```
# yum install nftables
# systemctl enable --now nftables.service
# dnf install iptables-services iptables-legacy
# systemctl start iptables
# systemctl is-active iptables nftables firewalld
➢ active
➢ inactive
➢ inactive
```

nftables

정책 출력은 아래 명령어로 가능하다.

```
# nft list tables ip
# nft list tables
# nft list counters
```

특정 아이피 드롭

```
# nft add rule ip filter OUTPUT ip daddr 1.2.3.4 drop
```

nftables

정책을 추가하는 명령어는 다음과 같이 사용한다. 예를 들어서 특정 아이피 드랍을 원하는 경우, 아래와 같이 사용한다.

```
# nft add rule ip filter OUTPUT ip daddr 1.2.3.4 drop
```

테이블 생성은 다음과 같이 한다. 같은 이름으로 테이블을 생성하지만, 사용하는 필터 위치가 다르기 때문에 중복이 되지 않는다.

```
# nft add table inet base_table
# nft add table arp base_table
```

목록 출력은 아래와 같은 명령어로 가능하다.

```
# nft list tables
# nft list tables inet
```

nftables

테이블 제거를 하기 위해서는 아래와 같이 명령어를 입력한다.

```
# nft delete table inet base_table
```

체인을 테이블에 추가하기 위해서 다음과 같이 명령어를 입력한다.

```
# nft add chain inet base_table input_filter "{type filter hook input priority 0;}"
# nft -a list table inet base_table
```

80/TCP포트를 막기 위해서 다음과 같이 명령어를 사용한다.

```
# nft add rule inet base_table input_filter tcp dport 80 drop
```

nftables

모든 rules내용을 확인하기 위해서는 다음과 같이 명령어를 사용한다.

```
# nft -a list ruleset
```

핸들러 번호로 제거하기 위해서는 다음과 같이 명령어를 사용한다.

```
# nft delete rule inet base_table input_filter handle 3
```

nftables rules

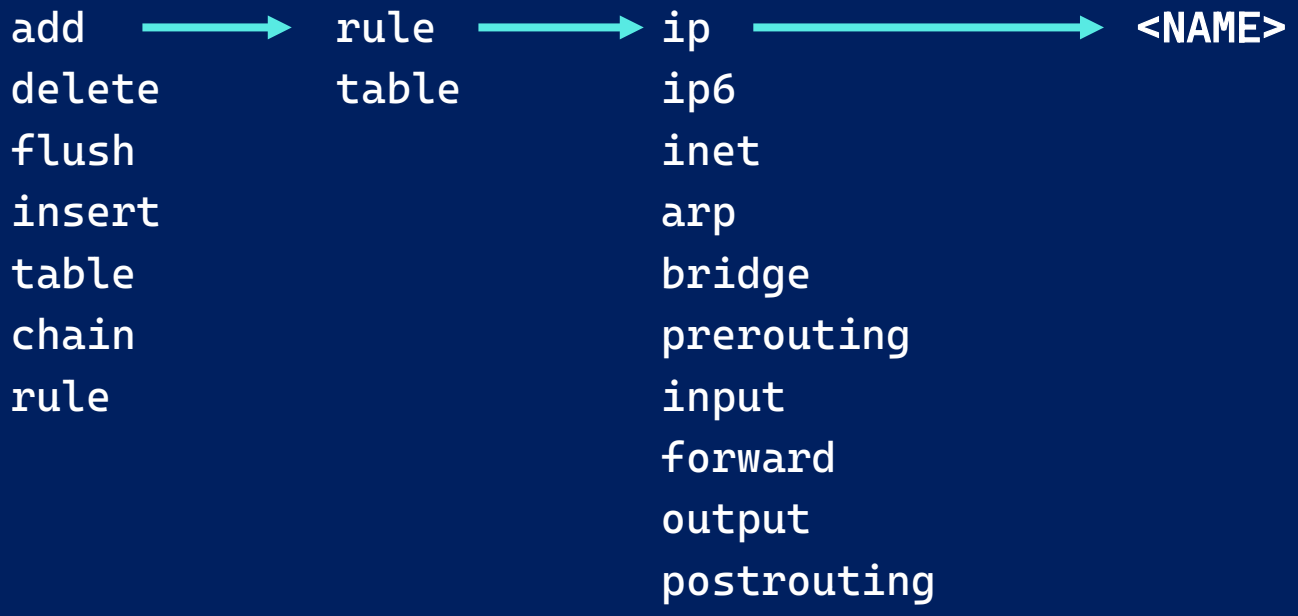
규칙(rule)은 일괄적으로 적용하기 위해서는 JSON형태로 파일을 작성 후, "nft"명령어로 밀어 넣으면 된다. "nft"는 "#!"를 지원하기 때문에 아래와 같이 작성 및 사용이 가능하다.

```
# vi httpd_service.sh
#! /sbin/nft -f
define http_ports = {80, 443}
flush ruleset
table inet local {
  chain input {
    type filter hook input priority 0; policy drop;
    tcp dport $http_ports counter accept comment "incoming http traffic";
  }
  chain output {
    type filter hook output priority 0; policy drop;
  }
}
#  nft -a list ruleset
```

nftables

명령어 사용 방법은 다음과 같다.

nft



nftables

대소문자 구별 합니다!

특정 아이피 드롭, 하지만 카운팅 모듈 사용.

```
# nft add rule ip filter OUTPUT ip daddr 1.2.3.4 counter drop
```

특정 아이피로 나가는 아이피 대역에 대한 카운팅.

```
# nft add rule ip filter OUTPUT ip daddr 192.168.1.0/24 counter
```

특정 포트번호에 대한 패킷 드랍.

```
# nft add rule ip filter INPUT tcp dport 80 drop
```

nftables

nftables의 family(protocol)은 다음과 같다.

ip	IPv4프로토콜 정책
arp	Address Resolution Protocol 제어 체인
ip6	IPv6프로토콜 정책
bridge	Linux Bridge 및 OVS Bridge
inet	일반적으로 많이 사용하는 애플리케이션 포트 정책
netdev	Container 및 VirtualMachine에서 사용하는 TAP(Test Access Point) 장치

자세한 설명은 https://lwn.net/Articles/631372/ 참고.

nftables

명령어 사용 예제. 아래 명령어가 제일 많이 사용하는 명령어 중 하나이다. 아래 명령어로 간단하게 시도한다.

ICMP 입력를 허용 합니다.

```
# nft add rule filter INPUT icmp type-echo-request accept
```

1.2.3.4로 나가는 트래픽에 대해서 거절 합니다.

```
# nft add rule ip filter OUTPUT ip protocol icmp ip daddr 1.2.3.4 counter drop
```

filte리스트의 테이블을 확인 합니다.

```
# nft -a list table filter
```

nftables

특정 rule를 제거한다. 번호를 꼭 넣어 주어야 한다.

```
# nft delete rule filter OUTPUT handle <NUMBER>
```

모든 rule를 메모리에서 비웁니다.

```
# nft flush table filter
```

filter테이블에 80/TCP포트를 허용 합니다.

```
# nft insert rule filter INPUT tcp dport 80 counter accept
```

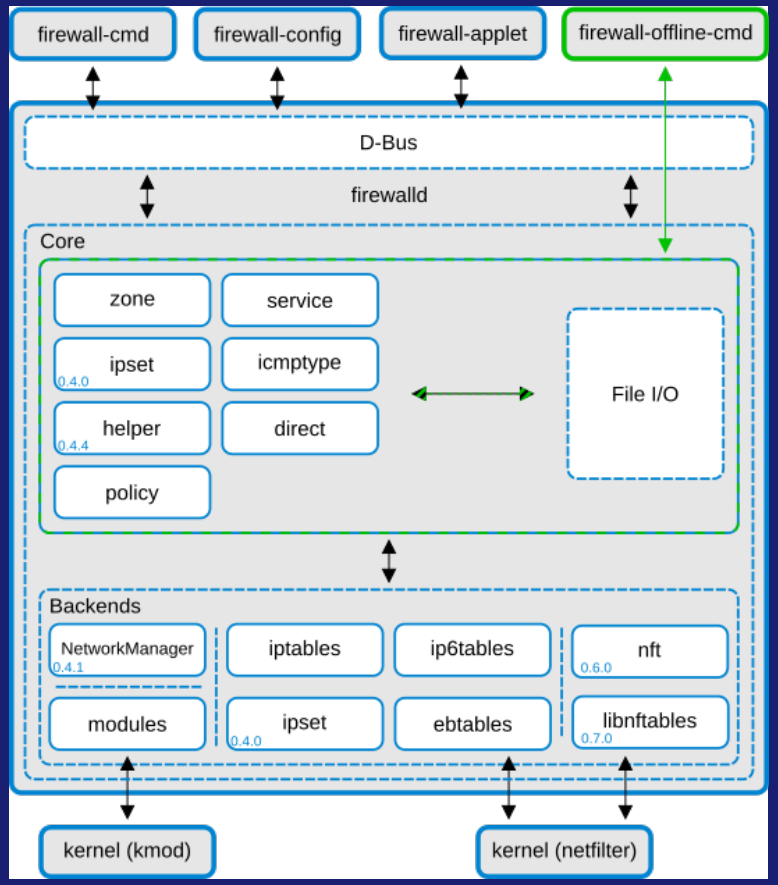
firewalld

기존에 사용하는 **iptables, nftables**위에 고급 계층을 올려서 사용자가 쉽게 사용할 수 있도록 한다. 현 레드햇 계열 배포판은 더 이상 iptables를 사용하지 않는다. 또한, **firewalld**도 **iptables**를 지원하지 않는다.

firewall-cmd명령어 기반으로 **영역(zone)**, **서비스(service)**, **포트(port)**와 같은 구성 요소를 XML기반으로 선언이 가능하다. 사용자가 원하는 경우, 추가적으로 XML파일을 만들어서 추가가 가능하다.

레드햇 계열은 **RHEL 9이후**부터 **nft/firewalld**만 사용이 가능하다.

firewalld


firewall-cmd
firewall-config
firewall-applet
firewall-offline-cmd
D-Bus
firewalld
Core
zone
service
ipset
0.4.0
icmptype
helper
0.4.4
direct
policy
File I/O
Backends
NetworkManager
0.4.1
iptables
ip6tables
nft
0.6.0
modules
ipset
0.4.0
ebtables
libnftables
0.7.0
kernel (kmod)
kernel (netfilter)

firewalld

```
# systemctl start firewalld
# ls -l /lib/firewalld
# cd /lib/firewalld/services
…
http.xml
…
```

firewalld

존 목록을 확인하는 방법.

```
# firewall-cmd --get-zones
# firewall-cmd --list-all --zone=
```

"특정 존"에 아이피/포트/서비스 및 기본 존 변경.

```
# firewall-cmd --add-source=10.10.10.0/24 --zone=block --permanent
# firewall-cmd --add-service=https --zone=drop
# firewall-cmd --add-service=http                ## public zone
# firewall-cmd --add-port=8899/tcp
# firewall-cmd --set-default-zone=block
# firewall-cmd --get-default-zone
```

패키지 관리 방법

dnf/module

repository mirror

dnf

"dnf"는 기존에 사용하던 "yum"명령어를 대체하는 새로운 패키지 관리자. dnf에서 제일 큰 차이점은 바로 modules라는 기능이 새로 도입이 되었음.

이를 통해서 특정 프로그램 설치 시 의존성이 필요한 경우 dnf module를 통해서 패키지 제공이 되며, 또한 이 기능은 이전에 사용하였던 "SCL(Software Collection Library)"와 비슷하게 호스트 영향 없이 확장이 가능하다.

MODULE

모듈 목록을 아래에서 확인이 가능하다.

https://gitlab.com/redhat/centos-stream/modules/

모듈을 확장하기 위해서 다음과 같이 명령어를 실행한다.

```
# dnf install https://rpms.remirepo.net/enterprise/remi-release-9.rpm -y
# dnf --enablerepo=remi-modular --disablerepo=appstream module list -y
```

module

```
# dnf module list

CentOS Stream 8 - AppStream

Name                Stream          Profiles Summary

container-tools     rhel8 [d][e]    common [ Most recent (rolling) versions of podman, buildah, skopeo, runc, conmon, runc, conmon, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and updatd]ed as frequently as every 12 weeks.

container-tools     1.0             common [ Stable versions of podman 1.0, buildah 1.5, skopeo 0.1, runc, conmon, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and supported for 24 monthd]s.
```

rollback

'yum', 'dnf'는 rollback기능을 제공한다.

```
dnf history          rollback
                     undo
                     redo
                     list
                     info
                     install
                     remove
                     update/upgrade
```

yum vs dnf-3

'yum'과 'dnf'의 제일 큰 차이점은 yum은 순수하게 파이선으로 작성이 되어 있다. 하지만, dnf는 libdnf3라는 C언어로 재-구성이 되었다. 이를 통해서 기존에 yum에서 불편했던 느린 반응 및 파이썬 라이브러리 문제 발생을 방지할 수 있다.

현재 대다수 RPM 및 레드햇 기반의 배포판은 "YUM"에서 "DNF-3/5"로 변경이 된 상태이다.

RPM commands

RPM명령어는 기존에는 'rpm'이라는 하나의 명령어만 사용하였다. RPM은 기능이 확장 되면서, 명령어 분리하기 시작. 아래처럼 분리가 되었다.

- ▣◀╖œ 기본 설치 명령어. 일반적으로 패키지 추가/삭제/제거 및 확인.
- ▣◀╖ł ▣Čżδ ĢĐœ ".rpm"패키지를 ".tgz"로 변경한다.
- ▣◀╖ Č◀ᵴ╝œ ".rpm"에서 cpio묶여 있는 파일을 푸는 명령어.
- ▣◀╖ăćœ RPM의 BerkelyDB 조회 시 사용하는 명령어.
- ▣◀╖⌐Đı ♂œ RPM에서 사용하는 공개키 관리.
- ▣◀╖▣▪ Đ▣ī œ RPM에서 사용하는 시그 키 관리.
- ▣◀╖ĢĐ▣ī δŢī œ RPM에 등록된 패키지의 자원들에 대한 의존성 검사.

RPM commands

일반적으로 'rpm'명령어를 통해서 패키지 관리 및 검사가 가능하다. 아래는 대표적으로 많이 사용하는 명령어이다.

```
rpm -ql <PACKAGE_NAME>
rpm -qa  --last <PACKAGE_NAME>
rpm -q <PACKAGE_NAME>
rpm -qf <FILENAME>
rpm -ivh <PACKAGE_NAME>
rpm -ev <PACKAGE_NAME>
rpm -qi <PACKAGE_NAME>
rpm -Vp <PACKAGE_NAME>
rpm -Va <PACKAGE_NAME>
rpm -qa gpg-pubkey*
rpm -q --scripts systemd
rpm -q --triggers systemd
```

yum/dnf

앞서 이야기 하였지만, YUM명령어는 DNF-3로 변경이 되었음. 저장소의 모든 RPM파일을 받고, 저장소를 구성하기 위해서는 다음과 같은 명령어로 구성이 가능함.

```
# dnf reposync
# dnf install createrepo_c
# createrepo_c .
```

위의 명령어로 구성이 완료가 되면, Apache, Nginx기반으로 저장소를 구성하면 됨. 예제는 아래와 같이 진행한다.

mirror

이전에 사용하던 미러링 기능도 dnf명령어로 통합이 되었다. 저장소 미러를 받기 위해서 다음과 같이 명령어를 수행한다.

```
# mkdir -p /mnt/reposdisk/rpms
# dnf reposync -p /mnt/repodisk/rpms
# createrepos_c /mnt/repodisk/rpms .

# dnf install httpd -y
# mount -obind /mnt/repodisk/rpms /var/www/html/
# vi /etc/yum.repos.d/mirros.repo
[kdn]
name=internal repo
baseurl=<IP>
gpgcheck=0
enable=1
```

웹 관리도구

COCKPIT

관리도구

프로세스

systemd

systemd기반에서는 **`'systemctl whoami'`**를 통해서 특정 프로세서가 어떠한 유닛에서 사용 중인지 확인이 가능하다. 이 부분에 대해서 위에서 언급하였기 때문에 직접 다루지는 않는다.

이외 나머지 부분은 유닛 제어 명령어를 통해서 제어가 관리하다. 이전 "System V"와 다르게 가급적이면 'systemctl'명령어로 관리를 권장한다.

ps

리눅스는 총 3가지 스타일로 제공한다.

1. BSD
2. UNIX
3. GNU

```
1   UNIX options, which may be grouped and must be preceded by a dash.
2   BSD options, which may be grouped and must not be used with a dash.
3   GNU long options, which are preceded by two dashes.
```

사람 마다 많이 다르기는 하는데 일반적으로 **유닉스/BSD** 옵션으로 많이 사용한다. GNU형식은 문자열이 상대적으로 길어서 잘 사용하지 않는다.

ps

'ps'명령어는 제일 기본적으로 사용하는 프로세스 확인 명령어. 제일 많이 유닉스 형태의 명령어는 다음과 같다.

```
# ps -ef
```

아래 명령어는 BSD 스타일로 많이 사용하는 옵션이다.

```
# ps aux
```

위의 명령어는 GNU 스타일의 명령어. 일반적으로 엔지니어들이 많이 사용하는 스타일은 "dash(-)"스타일이다.

```
# ps efww --pid 801
```

ps

"a"옵션은 BSD 스타일의 "only yourself" 옵션

"u"옵션은 EUID(effective user ID)

"x"옵션은 x BSD스타일의 "must have a tty" 옵션

 # ps aux or ps au

```
[root@localhost ~]# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.7 128144  6828 ?        Ss   Sep11   0:02 /usr/lib/systemd/syste
root           2  0.0  0.0      0     0 ?        S    Sep11   0:00 [kthreadd]
root           4  0.0  0.0      0     0 ?        S<   Sep11   0:00 [kworker/0:0H]
root           5  0.0  0.0      0     0 ?        S    Sep11   0:00 [kworker/u32:0]
root           6  0.0  0.0      0     0 ?        S    Sep11   0:00 [ksoftirqd/0]
root           7  0.0  0.0      0     0 ?        S    Sep11   0:00 [migration/0]
root           8  0.0  0.0      0     0 ?        S    Sep11   0:00 [rcu_bh]
root           9  0.0  0.0      0     0 ?        R    Sep11   0:36 [rcu_sched]
root          10  0.0  0.0      0     0 ?        S<   Sep11   0:00 [lru-add-drain]
root          11  0.0  0.0      0     0 ?        S    Sep11   0:00 [watchdog/0]
```

ps

"-f" 옵션은 "full-formatting list"

"-e" 옵션은 "all processes"

위의 명령어는 Unix(AIX)스타일의 명령어

```
# ps -ef
```

```
[root@localhost ~]# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Sep11 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --
root           2     0  0 Sep11 ?        00:00:00 [kthreadd]
root           4     2  0 Sep11 ?        00:00:00 [kworker/0:0H]
root           5     2  0 Sep11 ?        00:00:00 [kworker/u32:0]
root           6     2  0 Sep11 ?        00:00:00 [ksoftirqd/0]
root           7     2  0 Sep11 ?        00:00:00 [migration/0]
root           8     2  0 Sep11 ?        00:00:00 [rcu_bh]
root           9     2  0 Sep11 ?        00:00:36 [rcu_sched]
root          10     2  0 Sep11 ?        00:00:00 [lru-add-drain]
```

ps

```
# ps -x
```

```
[tang@www ~]$ ps -x
    PID TTY      STAT   TIME COMMAND
  69492 ?        Ss     0:00 /usr/lib/systemd/systemd --user
  69496 ?        S      0:00 (sd-pam)
  79289 ?        Ss     0:05 tmux
  79290 pts/15   Ss     0:00 -sh
  79320 pts/21   Ss     0:00 -sh
  79350 pts/22   Ss     0:00 -sh
  80825 pts/15   S+     0:00 ssh root@192.168.90.171
  80828 pts/15   S+     0:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
  80861 pts/21   S+     0:00 ssh root@192.168.90.3
  80864 pts/21   S+     0:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
  80865 pts/22   S+     0:00 ssh root@192.168.90.168
  80868 pts/22   S+     0:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.168
```

ps

```
# ps -fu or -fU <UID> <UNAME>
# ps -fU tang
```

```
[tang@www ~]$ ps -fU tang
UID          PID    PPID  C STIME TTY          TIME CMD
tang       69492       1  0 Sep06 ?        00:00:00 /usr/lib/systemd/systemd --user
tang       69496   69492  0 Sep06 ?        00:00:00 (sd-pam)
tang       79289       1  0 Sep06 ?        00:00:05 tmux
tang       79290   79289  0 Sep06 pts/15   00:00:00 -sh
tang       79320   79289  0 Sep06 pts/21   00:00:00 -sh
tang       79350   79289  0 Sep06 pts/22   00:00:00 -sh
tang       80825   79290  0 Sep07 pts/15   00:00:00 ssh root@192.168.90.171
tang       80828   80825  0 Sep07 pts/15   00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
tang       80861   79320  0 Sep07 pts/21   00:00:00 ssh root@192.168.90.3
tang       80864   80861  0 Sep07 pts/21   00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
```

ps

 # ps -U root -u root

"-u" 영향 받는 사용자 아이디(RUID)

"-U" 실제 사용자 아이디(EUID)

```
[tang@www ~]$ ps -u tang -U tang
    PID TTY          TIME CMD
  69492 ?        00:00:00 systemd
  69496 ?        00:00:00 (sd-pam)
  79289 ?        00:00:05 tmux: server
  79290 pts/15   00:00:00 sh
  79320 pts/21   00:00:00 sh
  79350 pts/22   00:00:00 sh
  80825 pts/15   00:00:00 ssh
  80828 pts/15   00:00:00 sss_ssh_knownho
```

ps

일반 포멧팅 및 확장 포멧팅을 활성화 하며, 그룹 이름 혹은 아이디로 조회한다.

```
# ps -Fg tang
# ps -fG 1000
```

```
[tang@www ~]$ ps -Fg tang
UID          PID    PPID  C    SZ   RSS PSR STIME TTY
tang       69492       1  0 22388  9528   4 Sep06 ?
tang       69496   69492  0 81902  3320   0 Sep06 ?
tang       79289       1  0  7246  4528   5 Sep06 ?
tang       79290   79289  0  6929  5140   3 Sep06 pts/15
tang       79320   79289  0  6929  5012   5 Sep06 pts/21
tang       79350   79289  0  6929  4988   3 Sep06 pts/22
tang       80825   79290  0 15448  6812   6 Sep07 pts/15
tang       80828   80825  0 24797  5588   6 Sep07 pts/15
tang       80861   79320  0 15448  6920   3 Sep07 pts/21
```

ps

명시된 특정 프로세서만 출력한다.

```
# ps -fp
```

```
[tang@www ~]$ sudo ps -fp 1
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Sep03 ?        00:01:30 /usr/li
```

ps

특정 부모 프로세서(parent process id)에 대해서 조회한다.

```
# ps -f --ppid 1
```

```
[tang@www ~]$ sudo ps -f --ppid 1
UID          PID    PPID  C STIME TTY          TIME CMD
root         751       1  0 Sep03 ?        00:00:11 /usr/lib/systemd/systemd-journald
root         799       1  0 Sep03 ?        00:00:06 /usr/lib/systemd/systemd-udevd
rpc          899       1  0 Sep03 ?        00:00:03 /usr/bin/rpcbind -w -f
root         908       1  0 Sep03 ?        00:00:04 /sbin/auditd
```

kill/killall/pkill

- 1, HUP, reload a process → systemctl reload sshd.service
- 9, KILL, kill a process without page down
- ‖ Dǿ˥ƒG∂ç, gracefully stop a process → systemctl stop

pkill/pgrep

pkill은 프로세스의 이름 혹은 다른 속성을 확인하여 종료 신호 전달.

```
$ pkill <signal> <process_name>

$ pgrep -u root sshd
$ pgrep -u tang,root,adaemon
```

pkill/pgrep

"ssh"프로세서 이름의 개수 확인.

```
$ pgrep -c ssh
10

$ pgrep -d "-" ssh
1363-30317-79439-79442-79459-79462-80023-80026-80047-80050-80051-80054-
80825
```

pkill/pgrep

사용자 "tang"이 사용중인 "ssh"으로 사용하는 프로세스 아이디가 출력된다.

```
$ pgrep -u tang ssh
80825
80828
80861
80864
80865
80868
131706
```

pkill/pgrep

특정 사용자가 사용하는 프로세서의 개수 및 프로세스 목록을 출력한다.

```
$ pgrep -u tang ssh -c
12

$ pgrep -l ssh
1363 sssd_ssh
30317 sshd
79439 ssh
79442 sss_ssh_knownho
79459 ssh
79462 sss_ssh_knownho
```

pkill/pgrep

특정 프로세스를 확인하기 위해서 'pgrep'명령어를 통해서 가능하다. 일반적으로 "ps -ef | grep"조합을 많이 사용하지만, 권장은 'pgrep'사용을 권장한다.

```
$ pgrep -a ssh
79439 ssh root@192.168.90.178
79442 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.178
79459 ssh root@192.168.90.187
79462 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.187
80023 ssh root@192.168.90.183
80026 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.183
131706 sshd: tang@pts/0
```

pkill/pgrep

ssh와 일치하지 않는 프로세스만 출력.

```
$ pgrep -v ssh
```

정확하게 sshd이름과 일치하는 프로세스만 출력.

```
$ pgrep -x sshd
```

최근에 생성된 ssh process만 출력.

```
$ pgrep -n ssh
```

이전에 생성된 ssh process만 출력.

```
$ pgrep -o ssh
```

pkill/pgrep

특정 사용자 혹은 프로그램의 PID정보를 확인하기 위해서 아래와 같이 명령어를 실행한다. 사용자 종료 경우에는 앞서 학습한 'loginctl'명령어를 사용해서 사용자 제어 및 관리한다.

```
$ pkill '^ssh$'
$ pkill -9 -f "ping 8.8.8.8"
$ pkill -U mark
$ pkill -U mark gnome -9
$ pkill -9 -n screen
```

절대로 "9"번 시그널은 사용중인 프로세서에서 사용하지 마세요!

사용자 관리

useradd

autofs

useradd, passwd

이미 알고 있는 명령어, 그리고 제일 많이 사용하는 명령어. 리눅스에는 두 가지 형식의 명령어가 존재한다.

- ð ł ă ă

GNU스타일 사용자 추가 명령어. 레드햇 및 대다수 리눅스 배포판은 useradd명령어 기반으로 사용자 추가한다.

ł ă ă - ð

이 명령어는 **데비안 계열 및 BSD**에서 많이 사용하는 명령어. 레드햇 계열 배포판은 useradd, adduser 둘 다 가지고 있지만 실제로는 useradd를 사용한다.

useradd

이 부분에서는 일반적으로 사용하는 사용자 및 암호 설정 과정을 다른 방식으로 다루어 본다.

```
# useradd -p $(openssl passwd -1 <PASSWORD>) <USERNAME>
# useradd test
# echo "<USERNAME>:<PASSWORD>" | chpasswd
# echo <PASSWORD> | passwd --stdin <USERNAME>
```

아래와 같이 사용자 암호생성 및 등록을 동시에 수행이 가능하다.

```
# adduser --password $(echo "helloworld" | mkpasswd --stdin) test1
# dnf install mkpasswd zsh
# mkpasswd -m sha512crypt test
# adduser -p $(mkpasswd -m sha512crypt test) -s zsh test2
```

user list

이전에 사용자 목록은 "/etc/passwd"파일에서 수동으로 확인하였다. 현재 리눅스에서는 사용자 목록 확인은 'getent'명령어를 통해서 확인을 권장하고 있다.

```
# grep ^test2 /etc/passwd
# cat /etc/passwd | grep test2
# getent passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
```

nfs server

사용자 홈 디렉터리를 일괄적으로 제공하기 위해서 "autofs"사용을 권장한다. 하지만, 이 방법은 오래된 방법이며, 레드햇 및 데비안 계열 배포판에서 기본값으로 지원하지 않는다.

아래는 "autofs"설치 및 설정 과정이다. 테스트를 하기 위해서 NFS서버 구성이 필요하다.

```
# dnf install nfs-utils
# systemctl enable --now nfs-server
# mkdir -p /srv/indirect/autofs-user1 autofs-user2
# mkdir -p /srv/direct/
# vi /etc/exports
/srv/indirect *(rw,sync)
/srv/direct/autofs-user3  *(rw,sync)
/srv/direct/autofs-user4  *(rw,sync)
# exportfs -avrs
```

4/1/2024

systemd(automount)

이전에 사용하였던 "autofs"는 복잡하고 사용하기가 어려웠다. "systemd"로 변경이 되면서, "autofs"는 "automount"라는 자원으로 통합 및 변경이 되었다. "automount"는 "mount"유닛과 의존성이 있다.

사용 방법은 앞에서 사용하였던 "mount"와 거의 동일하다. 파일 이름에 꼭 대시(-)로 디렉터리 구별을 해야 한다.

```
# vi home-testuser.automount
[Unit]
Description=automount for testuser
ConditionPathExists=/home/testuser

[Automount]
Where=/home/testuser
TimeoutIdleSec=10

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

systemd(automount)

오토 마운트가 사용할 일반 "mount"자원도 생성한다.

```
# vi home-testuser.mount
[Unit]
Description=mount for testuser

[Mount]
What=$YOUR_SERVER:/$YOUR_SHARE
Where=/home/testuser
Type=nfs
Options=_netdev,auto

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

autofs server/client

systemd의 automount와 비교하기 위해서 autofs서버를 설치 및 구성한다. 필요하지 않는 경우, 굳이 진행하지 않아도 된다. 아래와 같은 방식 더 이상 사용을 권장하지 않는다.

```
# dnf search autofs
# dnf install autofs
# systemctl enable --now autofs.service
# vi /etc/auto.master.d/direct.conf
/-          /etc/auto.direct

# vi /etc/auto.direct
/direct    -fstype=auto,rw,async node1.example.com:/direct

# vi /etc/auto.master.d/indirect.conf
/home/ /etc/auto.indirect

# vi /etc/auto.indirect
*       -rw,soft,async      node1.example.com:/indirect/&
```

autofs client

연결이 완료가 되면, 아래와 같이 확인이 가능하다.

```
# showmount -e node1.example.com
# ssh testuser@node1.example.com
# pwd
# df -h
```

가상머신 및 컨테이너

podman

libvirtd

컨테이너

포드만

리눅스 컨테이너

Podman은 기존에 사용하던 Docker대안으로 사용한다. 오픈소스 쪽에서는 Podman기반으로 자체적인 클러스터 및 Pod기반의 컨테이너 서비스 구성이 가능하다. Podman은 다음과 같은 서비스가 있다.

- podman.service
- io.podman.service

리눅스에서는 기본적으로 다음과 같은 컨테이너 런타임을 사용한다.

- runc/crun
- conmon

포드만

포드만은 오픈소스 표준 컨테이너 엔진이다. 이 컨테이너 엔진은 기존에 도커 엔진에서 제공하는 기능을 그대로 구현하지만, OCI, CRI와 같은 사양을 따르기 때문에 완벽한 표준 컨테이너 엔진이다.

현재 오픈소스 컨테이너에서는 표준 개발 도구로 사용하고 있으며, 쿠버네티스와 같이 사용이 가능하도록 자원을 제공 및 지원하고 있다.

포드만 다음과 같은 도구를 오픈소스 개발자 및 사용자에게 제공하고 있다.

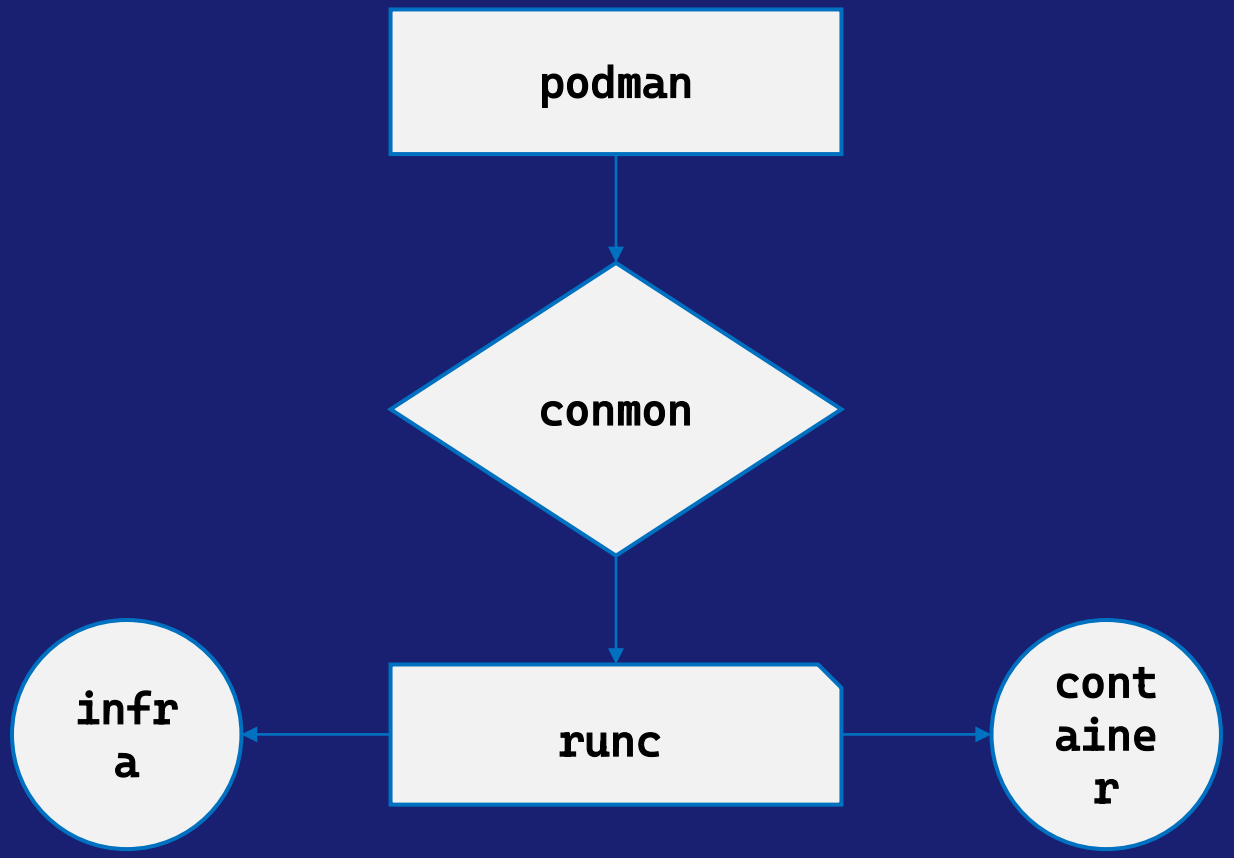
- Podman Desktop
- Podman Engine
- Buildah
- Skopeo

포드만

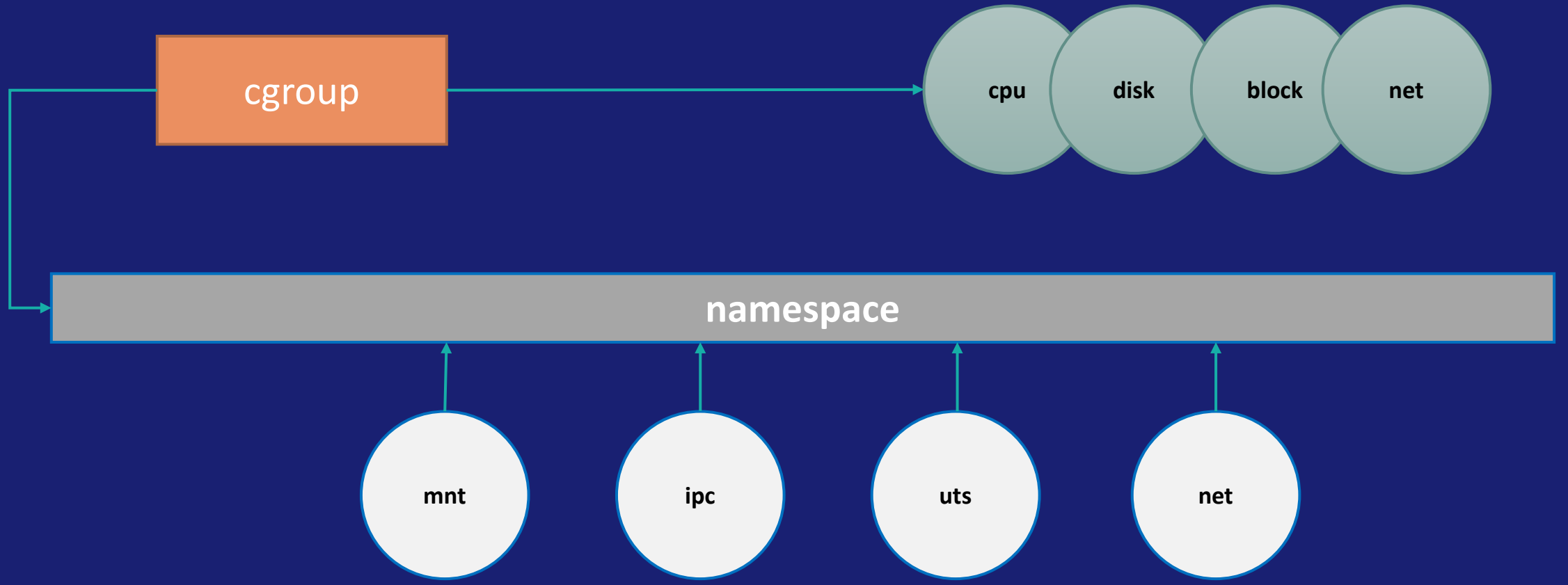
배포판 별로 다르지만 보통은 "podman.service" 혹은 "io.podman.service"라는 이름으로 운영이 된다. 레드햇 계열은 설치 시 다음과 같은 명령어로 설치한다.

```
# dnf install podman
# dnf module list containers
# dnf epel-release
# dnf search docker
> podman-docker
> podman-compose
```

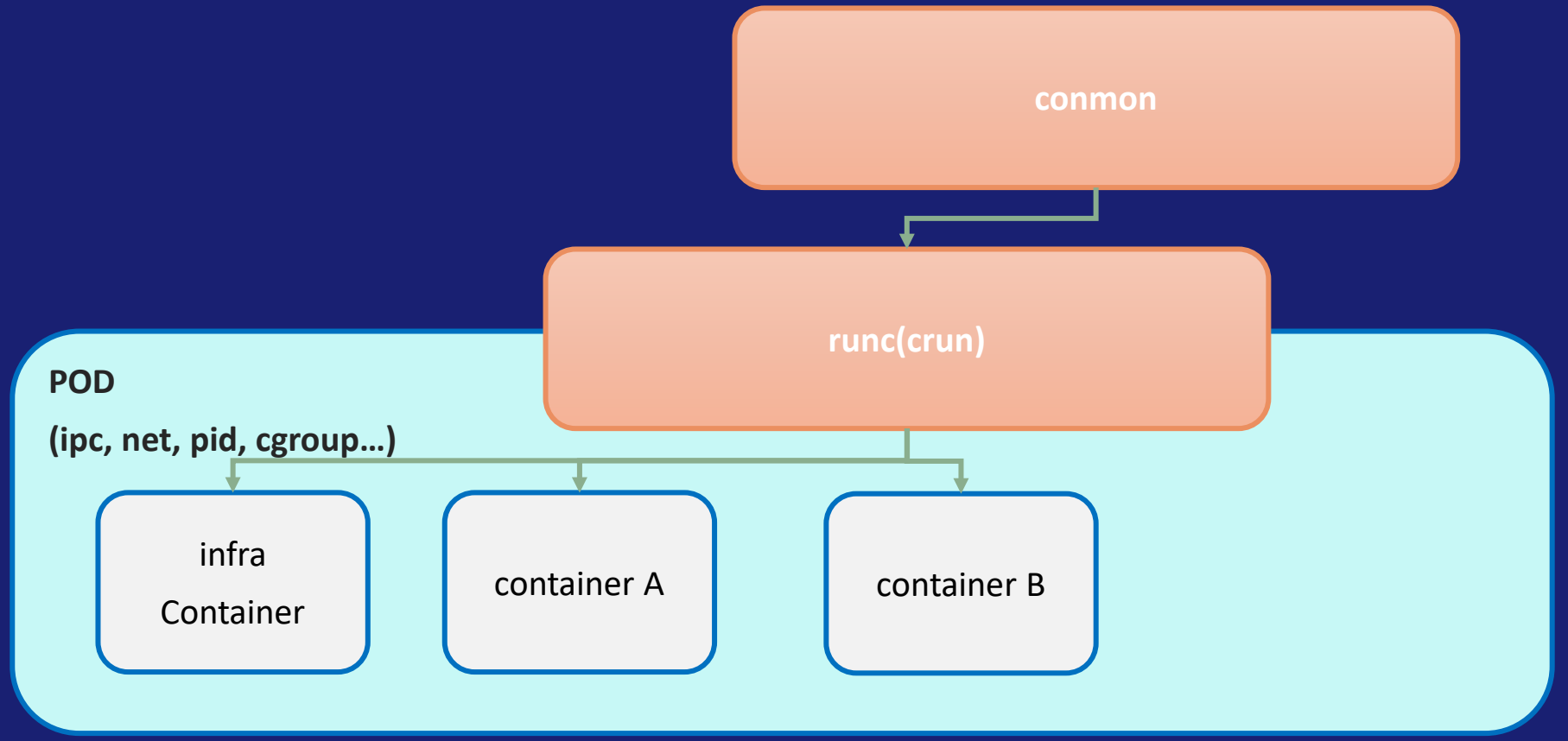
포드만



포드만



가상머신 및 컨테이너 구성 및 구현



이미지 내려받기

이미지를 원격 서버에서 내려받기 합니다.

```
# podman pull <IMAGE>
```

내려받은 이미지를 확인 합니다.

```
# podman images
```

이미지를 제거 합니다.

```
# podman rmi
```

이미지를 검사 합니다.

```
# podman inspect <IMAGE>
```

컨테이너 조회

현재 실행중인 컨테이너를 확인 합니다.

```
# podman container ls
# podman ps
```

실행중인 컨테이너를 제거 및 검사 합니다.

```
# podman rm <CONTAINER_ID>
# podman inspect <CONTAINER_ID>
```

컨테이너 중지 및 시작

```
# podman stop/start
```

pod 생성

컨테이너 생성과 비슷하게 POD구성 시 아래와 같이 명령어를 사용한다.

```
# podman pod ls
# podman pod create
# podman pod ls
# podman pod create --name POD_http
```

container 생성

생성된 Pod 및 컨테이너를 아래와 같은 명령어로 연결 혹은 새로 구성하면서 실행이 가능하다. 일단 Pod는 생성하지 않고, 컨테이너만 생성하여 동작이 올바르게 되는지 확인한다.

```
# podman container create --name httpd quay.io/centos7/httpd-24-centos7:centos7
# podman container ls
```

pod+container

"Pod", "Container"기반으로 컨테이너 인프라를 구현하려는 경우, 다음처럼 명령어를 실행한다. 포드(Pod)기반으로 컨테이너를 구현하는 경우, 볼륨, 네트워크 기능을 컨테이너 별로 분리 및 격리 운영이 가능하기 때문에 안전하게 사용이 가능하다.

또한, 쿠버네티스로 마이그레이션을 준비하는 경우, 포드 기반으로 생성하여 테스트 후, 손쉽게 마이그레이션 자원을 생성 할 수 있다.

```
# podman container create --name container_http --rm --pod new:pod_http quay.io/centos7/httpd-24-centos7:centos7
# podman start container_httpd
# podman container ls
# podman generate kube pod_http --filename pod_http.yaml --service --replicas 1 --type deployment
# ls pod_http.yaml
# kubectl apply -f pod_http.yaml
```

컨테이너 정보

∫ Ģł ◙∫ ┌δ ć∫ Č╝ ╒nł δ ╒Đ◙♂∫

컨테이너에서 사용하는 이미지 및 컨테이너 정보가 저장되는 위치. 임시적인 정보 및 메타정보가 저장이 된다.

∫ ◙▪ ╒∫ Č╝ ╒nł δ ╒Đ◙♂∫

동작 시 사용되는 임시 정보만 저장이 된다. 리부팅이 되거나 재시작이 되면 해당 정보들은 삭제가 된다.

∫ Đ'nČ∫ Č╝ ╒nł δ ╒Đ◙♂∫

OCI기반의 컨테이너 런타임 정보들은 위의 디렉터리에서 설정 및 구성이 된다. 예를 들어서 이미지를 받아오기 위한 정보인 레지스트리 서버도 이 디렉터리에서 구성이 된다.

∫ Đ'nČ∫ Č ╒δ∫

컨테이너에서 사용하는 컨테이너 네트워크 설정이다. 사용하는 런타임 혹은 오케스트레이션 프로그램 별로 구성 및 저장이 된다.

컨테이너 서비스화

리눅스에서 사용하는 **flatpak, snap**같은 패키지 서비스는 호스트에 설치 되는 방식이 아니다. 컨테이너 기반으로 애플리케이션 설치 및 실행을 한다. 이와 같은 방식으로 사용자가 만든 컨테이너를 애플리케이션 형태로 사용 및 동작이 가능하다.

앞에서 사용하였던, **"systemd-nspwan"**과는 다르다. **"systemd-nspwan"**는 **"boot-full"**컨테이너를 구성하며, "systemd-container"경우에는 systemd에서 서비스 형태로 구성한다.

컨테이너 서비스화

아래와 같이 간단하게 컨테이너를 포드만에서 생성한다. 아래는 간단하게 웹 서비스를 포드만으로 생성하며, 외부 포트 및 디렉터리를 볼륨으로 바인딩한다.

```
# mkdir /root/htdocs
# echo "Hello shell training" > /root/htdocs/index.html
# podman run -d --rm -p 8080:8080 -v /root/htdocs/:/var/www/html/ --name
apache quay.io/centos7/httpd-24-centos7:centos7
# podman ps
# curl http://localhost:8080
```

컨테이너 서비스화

생성된 서비스를 systemd의 서비스로 구성한다.

```
# podman generate systemd --restart-policy=always -t 1 apache
# podman generate systemd --new --files --name apache
# mkdir -p $HOME/.config/systemd/user/
# cp container-apache.service $HOME/.config/systemd/user/
# systemctl daemon-reload --user
# systemctl --user enable --now apache.service
# systemctl -t service --user
#/$ loginctl enable-linger <USERNAME>
```

buildah

포드만에서 이미지 빌드 부분이 독립이 되어서 나온 도구가 "buildah"이다. 컨테이너 이미지 빌드는 여전히 'docker build', 'podman build'사용이 가능하다.

표준 컨테이너에서는 CI/CD를 통해서 이미지 빌드 시, 위의 도구보다는 가급적이면 "buildah"사용을 권장한다. 현재 쿠버네티스는 이미지 생성 시, "Tekton" + "buildah"기반으로 권장한다.

buildah

이미지 생성 방식은 다음과 같다.

1. Dockerfile
2. Containerfile
3. from scratch

"Dockerfile"은 여전히 사용이 가능하지만, 오픈소스에서는 "Containerfile"으로 사용을 권고하고 있다.
"Containerfile"는 기존의 "Dockerfile"과 동일한 명령어 구조를 가지고 있다.

```
FROM fedora:latest
RUN echo "Installing httpd"; yum -y install httpd
EXPOSE 80
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

buildah

혹은 명령어 방식으로 구성이 가능하다. 이 방식은 CD에서 종종 사용하기도 한다.

```
# buildah from centos
# buildah run centos-working-container yum install httpd -y
# echo "Hello from the blackcat" > index.html
# buildah copy centos-working-container index.html /var/www/html/index.html
# buildah config --entrypoint "/usr/sbin/httpd -DFOREGROUND" centos-
working-container
# buildah commit centos-working-container centos-website
```

buildah

이미지를 처음부터 빌드하는 경우, 아래처럼 명령어를 실행한다.

```
# newcontainer=$(buildah from scratch)
# buildah containers
# scratchmnt=$(buildah mount $newcontainer)
# echo $scratchmnt
# dnf install -y --releasever=8 --installroot=$scratchmnt redhat-release --nogpgcheck
# yum install -y --setopt=reposdir=/etc/yum.repos.d --installroot=$scratchmnt --
setopt=cachedir=/var/cache/dnf httpd --nogpgcheck
# mkdir -p $scratchmnt/var/www/html
# echo "Your httpd container from scratch works!" > ?$scratchmnt/var/www/html/index.html
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

skopeo

이미지 레지스트리에 있는 이미지를 검색하거나 혹은 복사 시, 많은 제약이 있다. 특히, 도커 및 포드만 경우에는 상황에 따라서 프로토콜 버전에 따라서 검색이 잘 되지 않는다. 이러한 문제로 'search'부분을 분리하여 만든 도구가 "skopeo"이다.

사용방법은 간단하게 다음과 같다.

```
# skopeo copy docker://quay.io/skopeo/stable:latest
docker://registry.example.com/skopeo:latest
# mkdir -p /var/lib/images/busybox
# skopeo copy docker://busybox:latest dir:/var/lib/images/busybox
# skopeo copy docker://busybox:latest docker-archive:archive-
file.tar:busybox:latest
# skopeo sync --src docker --dest dir registry.example.com/busybox
/media/usb
# skopeo list-tags docker://docker.io/fedora
```

리눅스 가상화

libvirtd

리눅스 가상화

현래 리눅스에서 표준적으로 두 가지 가상화 형식을 제공하고 있다.

1. 하이퍼바이저 형식
2. 커널 기반 하이퍼바이저 형식

리눅스에서 많이 사용하는 QEMU/KVM는 **커널 기반 하이퍼바이저** 형식이다. 다만, QEMU/KVM기반으로만 가상머신을 구현하는 경우 관리가 어렵기 때문에 다음과 같은 도구를 통해서 가상머신을 관리한다.

1. libvirtd
2. libguestfs
3. virt-*(install, manager, p2v, top…)

위와 같은 도구를 통해서 가상머신 생성 및 관리를 한다.

가상화 소프트웨어

가상화 소프트웨어를 설치하기 위해서 기본적으로 다음과 같은 **그룹 패키지 설치를 권장**한다.

```
# dnf group list
# dnf group install "Virtualization Host" -y
```

설치가 완료가 되면 가상머신 생성 및 관리하는 라이브러리 대몬(libvirtd)을 실행한다.

```
# systemctl enable --now libvirtd
```

서비스가 올바르게 실행이 되면, 다음 명령어로 가상화 자원을 확인한다.

```
# virsh list
# virsh net-list
# virsh pool-list
```

libvirtd

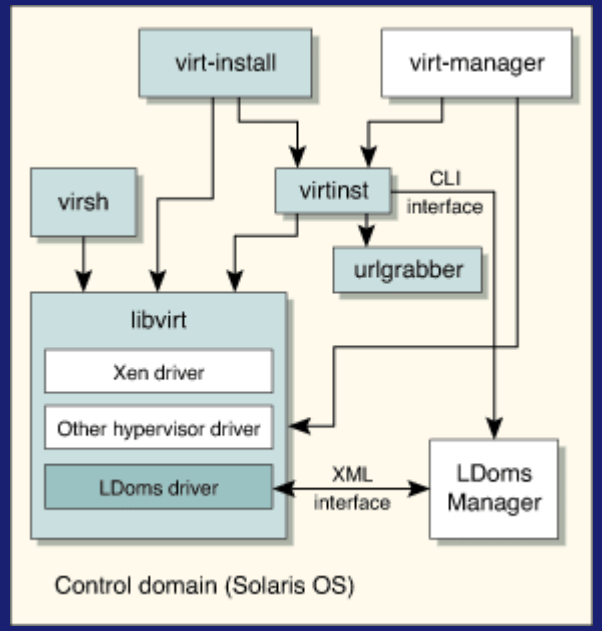
libvirtd에서 사용하는 디렉터리는 다음과 같다.

1. /var/lib/libvirt/
2. /etc/libvirt/

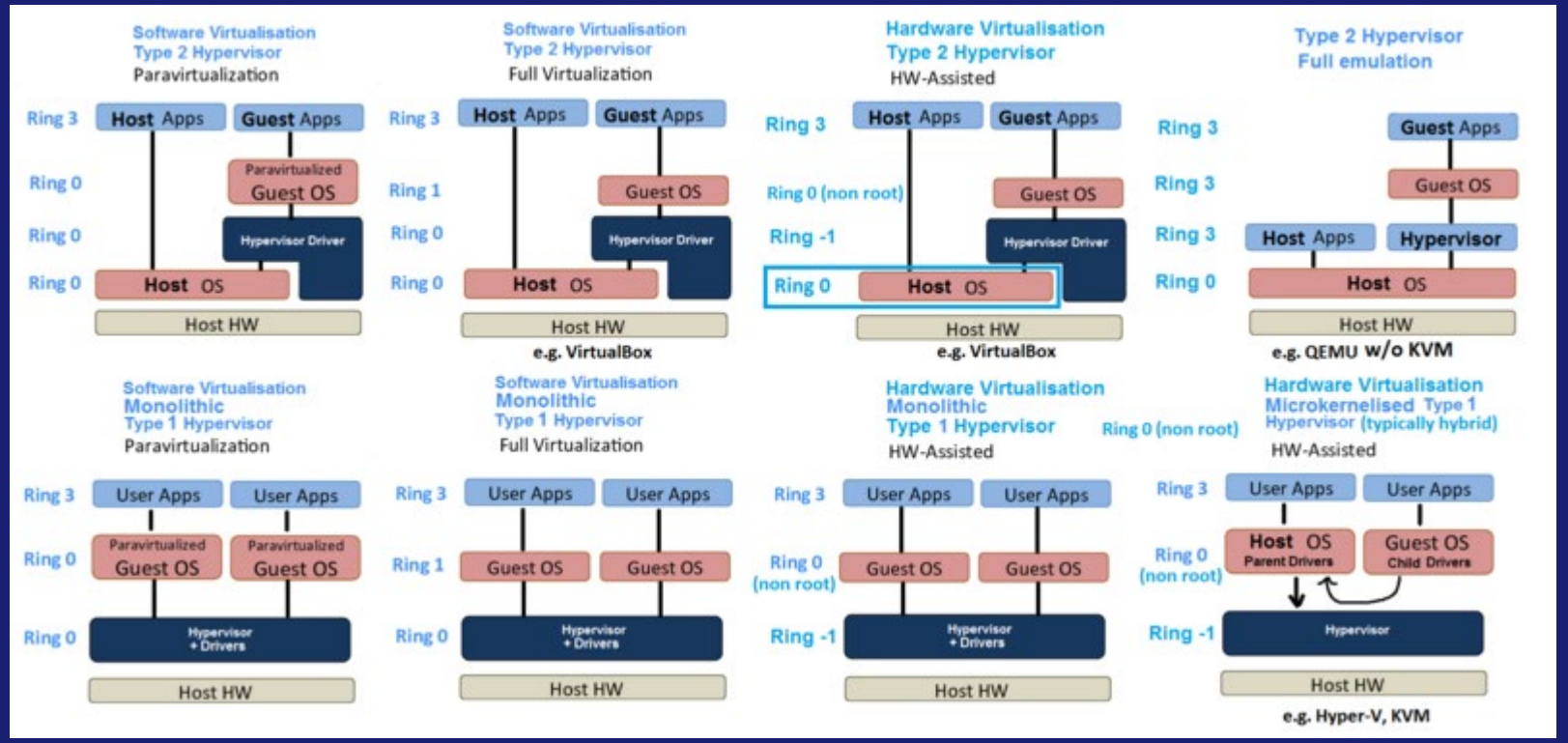
위의 두개의 디렉터리는 가상머신에서 사용하는 네트워크 정보 및 가상머신의 XML파일 그리고, 가상머신 이미지를 가지고 있다. 설정 파일은 "/etc/libvirt"를 사용하며, 가상머신 디스크는 일반적으로 "/var/lib/libvirt"를 사용한다.

libvirtd는 리눅스 가상화 솔루션, 예를 들어서 오픈스택, oVirt, VMware, Xen와 같은 하이퍼바이저들은 드라이버로 구성이 되어 동작한다.

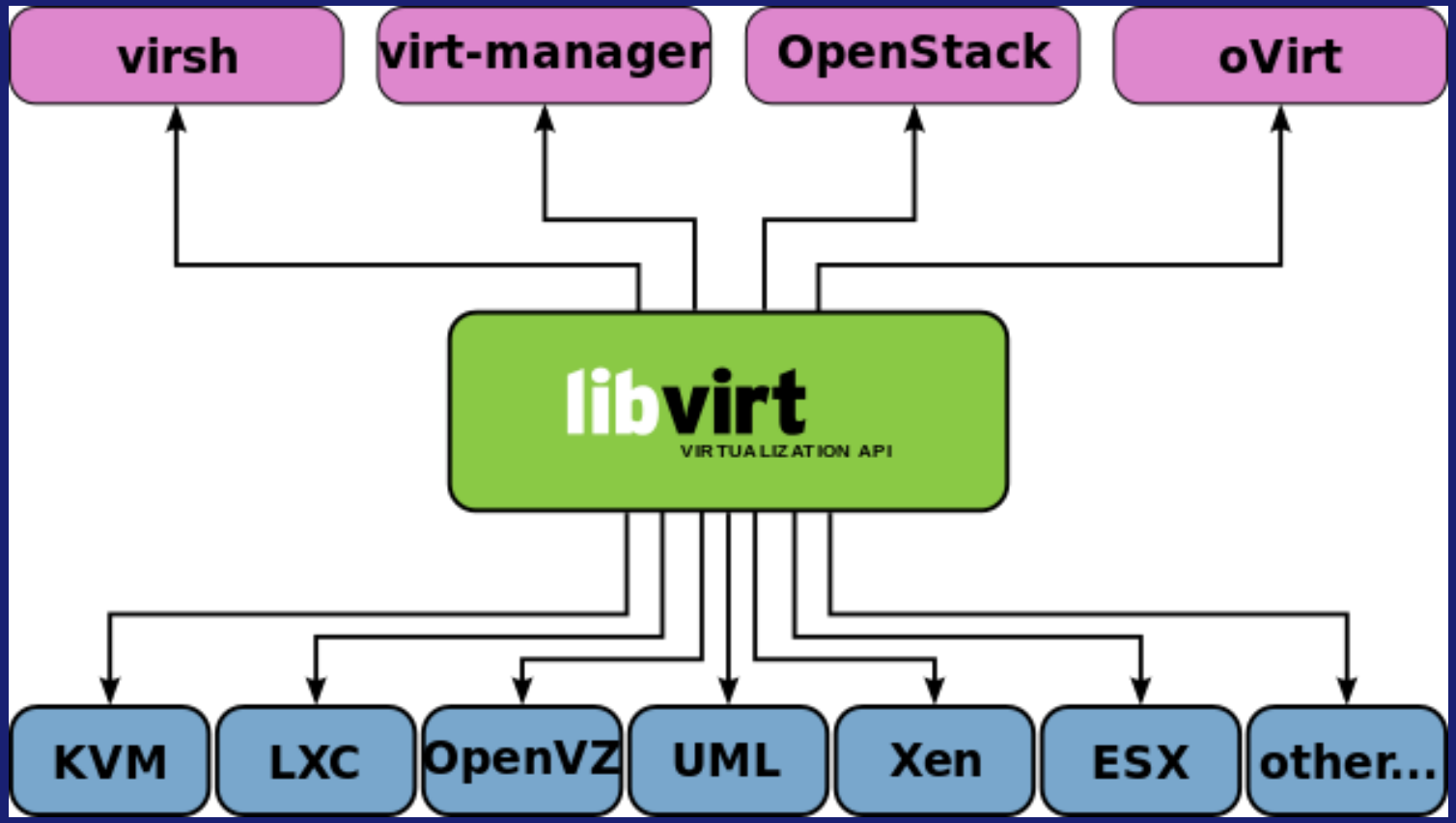
libvirtd


virt-install
virt-manager
virsh
virtinst
CLI
interface
urlgrabber
libvirt
Xen driver
Other hypervisor driver
LDoms driver
XML
interface
LDoms
Manager
Control domain (Solaris OS)

libvirtd


Software Virtualisation
Type 2 Hypervisor
Paravirtualization
Ring 3
Host Apps
Guest Apps
Ring 0
Paravirtualized
Guest OS
Ring 0
Hypervisor Driver
Ring 0
Host OS
Host HW
Software Virtualisation
Type 2 Hypervisor
Full Virtualization
Ring 3
Host Apps
Guest Apps
Ring 1
Guest OS
Ring 0
Hypervisor Driver
Ring 0
Host OS
Host HW
e.g. VirtualBox
Hardware Virtualisation
Type 2 Hypervisor
HW-Assisted
Ring 3
Host Apps
Guest Apps
Ring 0 (non root)
Guest OS
Ring -1
Hypervisor Driver
Ring 0
Host OS
Host HW
e.g. VirtualBox
Type 2 Hypervisor
Full emulation
Ring 3
Guest Apps
Ring 3
Guest OS
Ring 3
Host Apps
Hypervisor
Ring 0
Host OS
Host HW
e.g. QEMU w/o KVM
Software Virtualisation
Monolithic
Type 1 Hypervisor
Paravirtualization
Ring 3
User Apps
User Apps
Ring 0
Paravirtualized
Guest OS
Paravirtualized
Guest OS
Ring 0
Hypervisor
+ Drivers
Host HW
Software Virtualisation
Monolithic
Type 1 Hypervisor
Full Virtualization
Ring 3
User Apps
User Apps
Ring 1
Guest OS
Guest OS
Ring 0
Hypervisor
+ Drivers
Host HW
Hardware Virtualisation
Monolithic
Type 1 Hypervisor
HW-Assisted
Ring 0 (non root)
Ring 3
User Apps
User Apps
Ring 0
(non root)
Guest OS
Guest OS
Ring -1
Hypervisor
+ Drivers
Host HW
Hardware Virtualisation
Microkernelised Type 1
Hypervisor (typically hybrid)
HW-Assisted
Ring 3
User Apps
User Apps
Ring 0
(non root)
Host OS
Parent Drivers
Guest OS
Child Drivers
Ring -1
Hypervisor
Host HW
e.g. Hyper-V, KVM

libvirtd


virsh
virt-manager
OpenStack
oVirt
libvirt
VIRTUALIZATION API
KVM
LXC
OpenVZ
UML
Xen
ESX
other...

명령어

가상머신 이미지 빌드 생성은 여기서 다루지는 않으며, 가상머신에서 사용하는 이미지는 'virt-builder'통해서 내려 받기 한다. 'virt-builder'는 별도의 설정이 없으면 "libguestfs"에 제공해주는 디스크 이미지를 사용한다.

별도로 웹 서버 기반으로 **내부용 이미지 서버**(virtual disk image registry server)를 구성해서 배포용으로 사용이 가능하다. 아래 명령어로 이미지 구성한다.

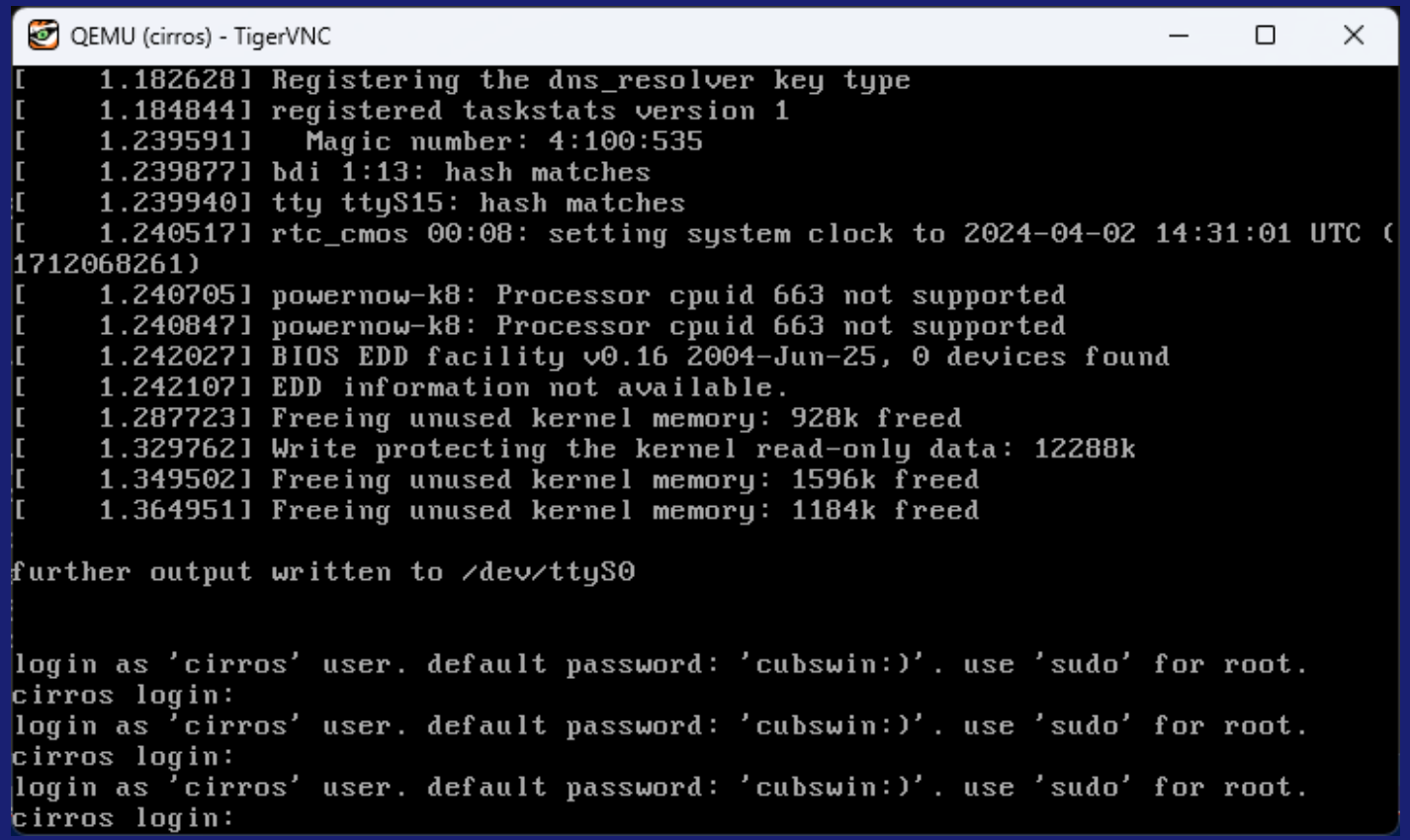
```
# virt-builder --list
# virt-builder cirros-0.3.5 --output /var/lib/libvirt/images/cirros.raw
# virt-builder --root-password password:centos --size 10G --format qcow2
centosstream-9 --output /var/lib/libvirt/images/centosstream-9.qcow2
# qemu-img info /var/lib/libvirtd/images/centosstream-9.qcow2
```

명령어

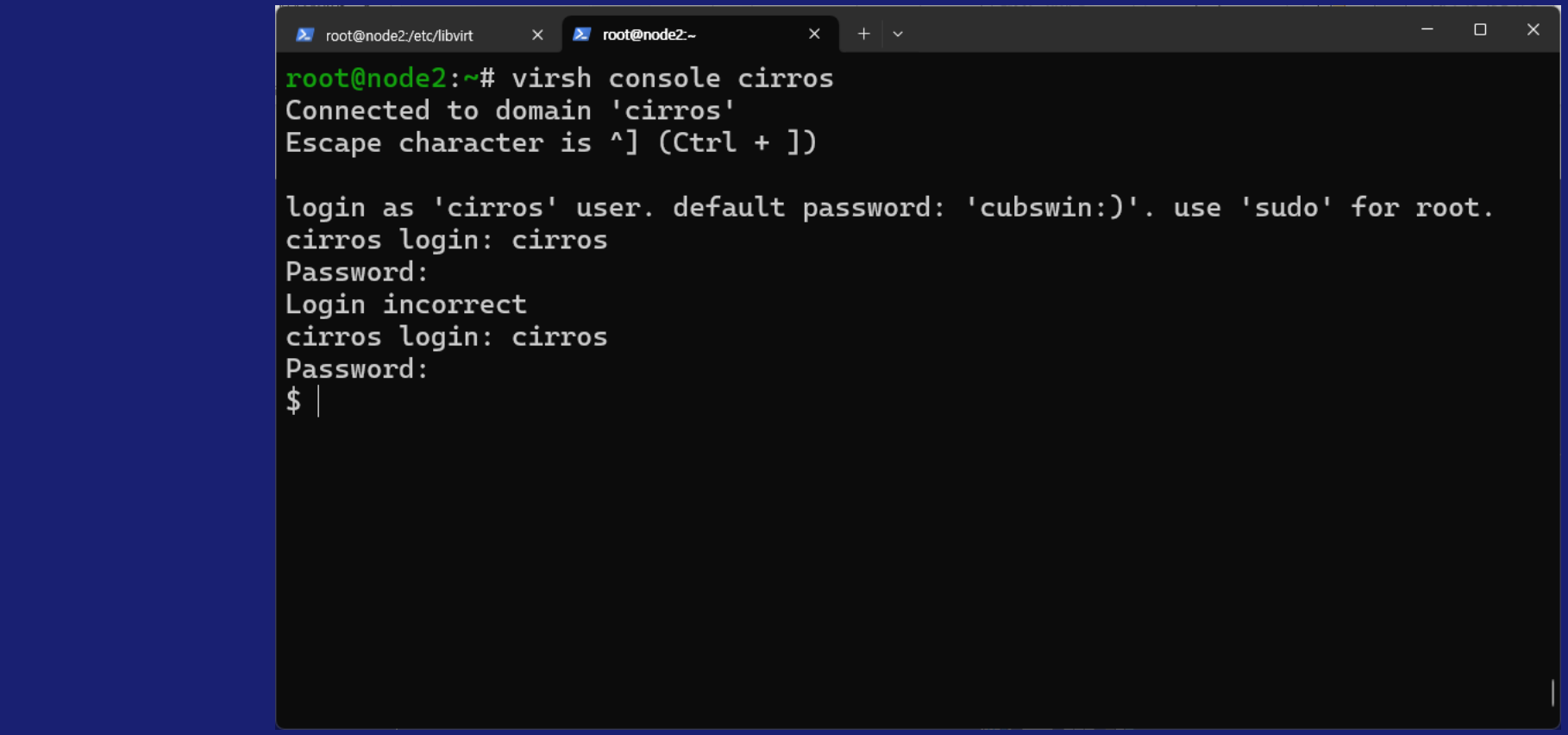
가상머신에서 사용할 패키지 및 가상머신을 생성한다. 가상머신 이미지가 문제 없이 구성이 되면서, 가상머신을 하나씩 올리도록 한다.

```
# dnf install virt-install -y
# virt-install --osinfo list | grep -e centos -e cirros
# virt-install --name cirros --vcpu 2 --memory 512 --
disk=path=/var/lib/libvirt/images/cirros.raw --osinfo=cirros0.3.0 --import --
noautoconsole --network=network=default --graphics vnc,port=5901,listen=0.0.0.0 --
destroy-on-exit
# virt-install --name centosstream-9 --vcpu 2 --memory 1536 --
disk=path=/var/lib/libvirt/images/centosstream-9.qcow2 --osinfo=centos-stream9 --
import --noautoconsole --network=network=default --graphics
vnc,port=5902,listen=0.0.0.0 --destroy-on-exit
```

console(vnc)


QEMU (cirros) - TigerVNC
[1.182628] Registering the dns_resolver key type
[1.184844] registered taskstats version 1
[1.239591] Magic number: 4:100:535
[1.239877] bdi 1:13: hash matches
[1.239940] tty ttyS15: hash matches
[1.240517] rtc_cmos 00:08: setting system clock to 2024-04-02 14:31:01 UTC (
1712068261)
[1.240705] powernow-k8: Processor cpuid 663 not supported
[1.240847] powernow-k8: Processor cpuid 663 not supported
[1.242027] BIOS EDD facility v0.16 2004-Jun-25, 0 devices found
[1.242107] EDD information not available.
[1.287723] Freeing unused kernel memory: 928k freed
[1.329762] Write protecting the kernel read-only data: 12288k
[1.349502] Freeing unused kernel memory: 1596k freed
[1.364951] Freeing unused kernel memory: 1184k freed
further output written to /dev/ttyS0
login as 'cirros' user. default password: 'cubswin:)'. use 'sudo' for root.
cirros login:
login as 'cirros' user. default password: 'cubswin:)'. use 'sudo' for root.
cirros login:
login as 'cirros' user. default password: 'cubswin:)'. use 'sudo' for root.
cirros login:

console(serial)



자동화

앤서블 맛보기

앤서블 소개

앤서블은 본래 Ansible Community, Company에서 제작하였고 현재, 레드햇이 인수하였음. 앤서블은 총 두 가지 릴리즈를 유지하고 있음.

1. Ansible Core
2. Ansible Engine

앤서블 코어

코어는 앤서블 앤서블 핵심 모듈로 구성이 되어 있으며, 그 이외 확장으로 **posix, collection, community**로 통해서 확장이 가능하다. 기본적으로 많이 사용하는 모듈은 core/posix 그 이외 나머지 기능들은 collection, community로 확장.

조금 혼돈 스럽기는 하지만 앤서블 **코어 == 엔진(engine)**이라고 표현하기도 한다. **core, engine** 둘 다 같은 기능이다. 현재 앤서블은 앤서블 **AAP(Automation Application Platform)**이라는 이름으로 프로젝트를 진행하고 있다.

앤서블 소개

또한 앤서블 코어는 두 가지 릴리즈 방식이 있다.

- ansible-core(another name is ansible-base)

  앤서블 코어는 앤서블 인터프리터 + 코어 모듈
- ansible-project

  앤서블 코어 + 추가적인 컬렉션 구성
- ansible-navigator

  앤서블 플레이북 실행 및 관리하는 통합 런처 도구

앤서블 소개

- The ansible==4.0.0 package on PyPI will depend on ansible-core>=2.11
- ansible==3.0.0 that was released today depends on ansible-base>=2.10.5,<2.11.
- ansible-core doesn't become 4.0.0, the next version will be 2.12.

앤서블 소개

앤서블 사용하기 전에 준비를 해야 될 부분은 다음과 같다.

- YAML 작성 시 사용할 에디터(아무거나 좋다! 정말로!)
- YAML 문법
- "ansible.cfg" 및 directory work frame
- ansible, ansible-playbook명령어 사용 방법

vim editor

혹은 centos 8버전 이후를 사용한다면?

```
$ sudo yum install vim-ansible vim -y
$ touch .vimrc
```

리눅스 콘솔에서 작성 시 사용하는 대표적인 에디터는 vi/vim에디터가 있다. vi, vim를 사용하기 위해서는 아래와 같이 설정을 한다. 설정하지 않는 경우, 에디터 사용이 어려울 수 있다.

```
$ vi .vimrc
autocmd FileType yaml setlocal et ts=2 ai sw=2 nu sts=0
set cursorline
```

YAML 문법

```yaml
%YAML 1.2
---
YAML: YAML Ain't Markup Language™

What It Is:
  YAML is a human-friendly data serialization
  language for all programming languages.

YAML Resources:
  YAML Specifications:
  - YAML 1.2:
    - Revision 1.2.2      # Oct 1, 2021 *New*
    - Revision 1.2.1      # Oct 1, 2009
    - Revision 1.2.0      # Jul 21, 2009
  - YAML 1.1
  - YAML 1.0

  YAML Matrix Chat:   '#chat:yaml.io'     # Our New Group Chat Room!
  YAML IRC Channel:   libera.chat#yaml    # The old chat
  YAML News:          twitter.com/yamlnews
  YAML Mailing List:  yaml-core           # Obsolete, but historical

  YAML on GitHub:                         # github.com/yaml/
    YAML Specs:         yaml-spec/
    YAML 1.2 Grammar:   yaml-grammar/
    YAML Test Suite:    yaml-test-suite/
    YAML Issues:        issues/
```

SSH

앤서블은 기본적으로 두 가지 접근 방식을 제공한다.

1. SSH 비공개/공개키 접근 방법
2. SSH기반으로 사용자 아이디 및 비밀번호 접근

이 둘 중 하나를 사용하면 된다. 앤서블은 키 기반으로 호스트 접근 및 관리를 권장한다. 2번은 일반적으로 권장하지 않는다.

SSH

```
$ ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
$ ssh-copy-id <ID>@<HOST>
```

YAML

앤서블에서 사용하는 문법을 작성하기 위해서는 다음과 같은 조건을 만족해야 한다.

- 최소 한 칸 이상의 띄어쓰기(권장은 2칸)
- 탭 사용시 반드시 빈 공간으로 전환
- 블록 구별은 -(대시)로 반드시 처리

YAML

```
- name: simple playbook
  hosts: all
  become: true

  tasks:
  - module:
      args1:
      args2:
```

YAML

앤서블 블록 구별을 보통 "-"로 구별한다. 시작 블록은 보통 다음과 같은 형식으로 많이 사용한다.

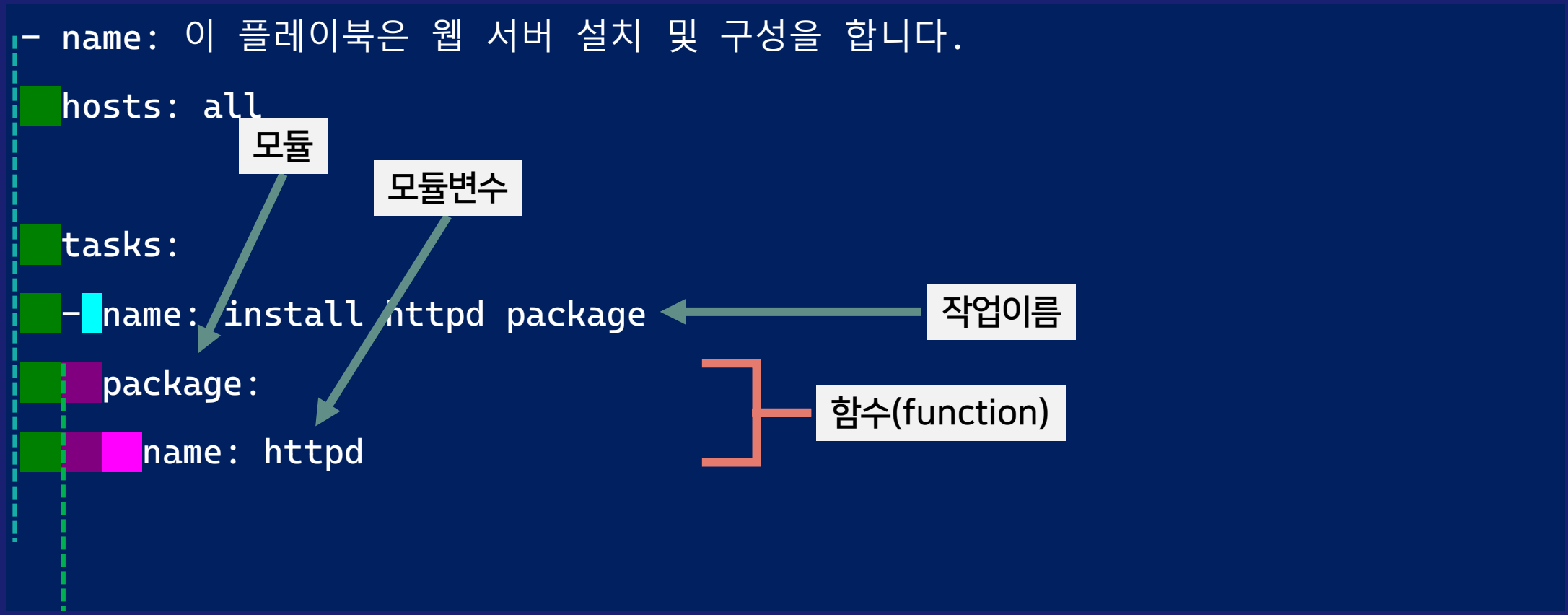
```
- name:
  module:
    args: value

- name:
  module:
    args: value
```

YAML

앤서블 맨 위의 상단은 아래처럼 구성한다.

```
- name: <작업이 수행 시 출력되는 이름>
  hosts: all                    ## 대상서버 이름.
  become: true                  ## 앤서블 내장 키워드
```

YAML


- name: 이 플레이북은 웹 서버 설치 및 구성을 합니다.
hosts: all
tasks:
- name: install httpd package
package:
name: httpd
모듈
모듈변수
작업이름
함수(function)

기본 명령어 키워드

"Hosts"키워드는 다음과 같은 미리 예약된 옵션이 있다.

- **localhost**: 127.0.0.1와 같은 자기 자신 루프 백(loopback)
- **all**: 인벤토리에(inventory)등록된 모든 호스트
- **[group]**: 특정 그룹에만 적용하는 명령어 키워드
- **inventory, group** 이런 부분은 너무 깊게 들어가지 말기!

기본 명령어 키워드

맨 상단에 있는 키워드 및 옵션은 보통 전역 키워드(global keyword)라고 생각하면 된다.

여기에서 적용된 옵션 및 명령어는 모든 플레이북에 적용이 되며 기존에 적용이 되어 있는 "ansible.cfg"의 내용을 오버라이드(override)가 된다.

```
become:
remote_user:
```

기본 명령어 키워드

모든 작업이 시작되는 구간. tasks 구간에는 여러 **모듈(module)**이 모여서 하나의 작업 **워크플로우(workflow)**를 구성한다.

여러 개의 워크 플로우가 구성이 되면 이걸 플레이북 혹은 **플레이북 작업(playbook tasking)**이라고 부른다.

앤서블 문법

그래서 권장하는 방법은 작성시 각각 모듈에 "name:"키워드를 사용하여 작성 및 구성을 권장한다.

```
- name: this is the first module task
  ping:
```

위와 같은 방법으로 명시한 모듈에 어떻게 사용할 것인지 명시한다.

앤서블 모듈 및 패키지

제일 많이 사용하는 모듈 "copy"로, **파일복사 기능**을 구현하면 다음과 같다.

```
- name: copy an issue file to remote server
  copy:
    src: /tmp/issue
    dest: /etc/issue
```

앤서블 모듈 및 패키지

모듈에 대한 자세한 옵션을 보기 위해서는 다음과 같은 명령어로 실행한다.

```
$ ansible-doc <MODULE NAME>
```

사용 가능한 모듈 목록을 확인하기 위해서는 아래 명령어로 목록 확인이 가능하다.

```
$ ansible-doc -l
```

앤서블 명령어

앤서블은 YAML형태 말고 ad-hoc방식이 있다. 이 방식은 마치 쉘 스크립트 실행하는 방식과 비슷하게 **모듈+인자 형태**로 구성이 되어 있다.

아래는 간단한 ad-hoc사용 방식이다.

```
$ ansible <host>, -m <module> -a "arg1=<value> arg2=<value>:
```

앤서블 명령어

애드훅은 쉘 스크립트에서 같이 사용하거나 혹은 몇몇 쉘 스크립트 기능을 표준화 모듈 기반으로 사용하기 위해서 사용한다.

자주 사용하지는 않지만 애드훅 기반으로 구성하는 경우 아래와 같은 방식으로 구성을 한다.

앤서블 문법

```
$ ansible localhost, -m ping

$ ansible localhost, -m copy -a "src=/etc/hostname dest=/root/hostname
remote_src=yes"

$ ansible localhost, -m package -a "name=vsftpd state=present"
```

앤서블 인벤토리

인벤토리

4/1/2024

앤서블 인벤토리

앤서블 인벤토리는 다음과 같은 형식을 가지고 있다.

[인벤토리 이름]

<호스트>

[인벤토리 이름:children]

<그룹이름>

앤서블 인벤토리

인벤토리는 위의 내용을 기준으로 다음과 같은 내용을 가지고 있다.

- 호스트 이름
- 아이피 주소
- 호스트에서 사용하는 변수

앤서블 인벤토리

인벤토리 파일은 일반적으로 "inventory"혹은 "hosts" 파일이름으로 구성함. 다른 이름으로 변경을 원하는 경우 "ansible.cfg"에서 변경이 가능함. "-i"으로 임의적으로 선택 가능.

```
# mkdir ansible
# cd ansible
# nano hosts
[web]
localhost

[db]
localhost
```

```
$ ansible-playbook <PLAYBOOK>
$ ansible-playbook -i  <PLAYBOOK>
```

앤서블 설정

앤서블 설정은 보통 아래와 같이 한다. 앤서블 설정 파일은 다음과 같은 명령어로 생성이 가능하다.

```
$ cat ansible.cfg
[defaults]
inventory = hosts
# ansible-config init --disabled -t all > ansible.cfg
```

앤서블 플레이북

앤서블에서 서버 상태를 확인하는 핑 플레이북을 만들어보자.

```
# vim ping.yaml
---
- hosts: all
  tasks :
  - name: ping to all hosts
    ping:
```

앤서블 플레이북

위에서 학습했던 내용으로 다음과 같이 응용 및 활용한다.

```
# vim copy.yaml
- hosts: all
  tasks :
  - name: copy to /etc/hostsname on the remote /root/hostname.2
    copy:
      src: /etc/hostname
      dest: /root/hostname.2
    remote_src: yes
# ansible-playbook copy.yaml
# ls -l /root/hostname.2
```

앤서블 플레이북

패키지 설치 플레이북은 다음과 같다.

```
# vi package.yaml
- hosts: all
  tasks :
  - name: install the postfix package
    yum:
      name:
        - postfix
        - httpd
        - mariadb-server
      state: present
# ansible-playbook package.yaml
# rpm -qa | grep -e postfix -e httpd -e mariadb-server
```

앤서블 플레이북

```
# vi html.yaml
- hosts: all
  tasks :
  - name: make a text file for httpd service
    copy:
      content: "Hello World My Ansible Service"
      dest: /var/www/html/index.html
# ansible-playbook html.yaml
# ls -l /var/www/html/index.html
```

앤서블 플레이북

```
# vi service.yaml
- hosts: all
  tasks :
  - name: start and enable to a httpd, mariadb-server and postfix service
    systemd:
      name: "{{ item }}"
      state: started
      enabled: yes
    loop:
      - httpd
      - mariadb
      - postfix
# ansible-playbook service.yaml
# systemctl is-active httpd
```

앤서블 플레이북

```
# vi hostname.yaml
- hosts: all
  tasks :
  - name: change to a hostname
    hostname:
      name: www.example.com
# ansible-playbook hostname.yaml
# hostnamectl
```