

# STANDARD LINUX

최국현

[tang@linux.com](mailto:tang@linux.com)

03/25/2023

# 소개

랩 및 교육대상



# 교육대상

- 리눅스를 사용하였으나, 오랫동안 리눅스 배포판 및 기술에 대해서 갱신하지 않는 사용자
- 전반적인 오픈소스 기반 운영체제에 대한 동향을 알고 싶은 사용자

# 강사소개

이름: 최 국현

- [tang@linux.com](mailto:tang@linux.com)

:)

# 목차



# 랩

가상머신 두 대 기반으로 진행을 권장 합니다. 한 대만 사용하셔도 크게 문제가 없습니다.

## 리눅스 배포판

- 가급적이면 **CentOS-9-Stream, Rocky 9**으로 권장 합니다.
- 우분투 사용 하셔도 상관 없습니다. 다만 몇몇 명령어 및 패키지는 호환이 되지 않을 수 있습니다.

## 가상 머신 사양

---

**vCPU:** 2개

**vMEM:** 4096

**vDISK:**

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

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

# 오픈소스

리눅스와 오픈소스 관계



# 오픈소스

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

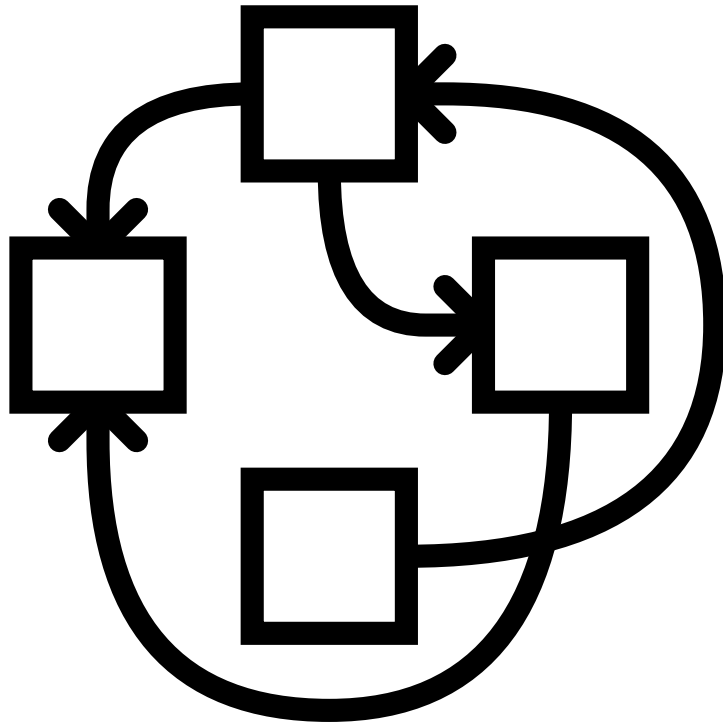
선진적인 디자인의 **마이크로 커널 디자인(Micro Kernel)**를 사용하여서, 당시 성능으로는 구현 및 사용하기가 어려운 부분이 많았음.

- [GNU/Linux Kernel](#)
- [GNU/Hurd Kernel](#)

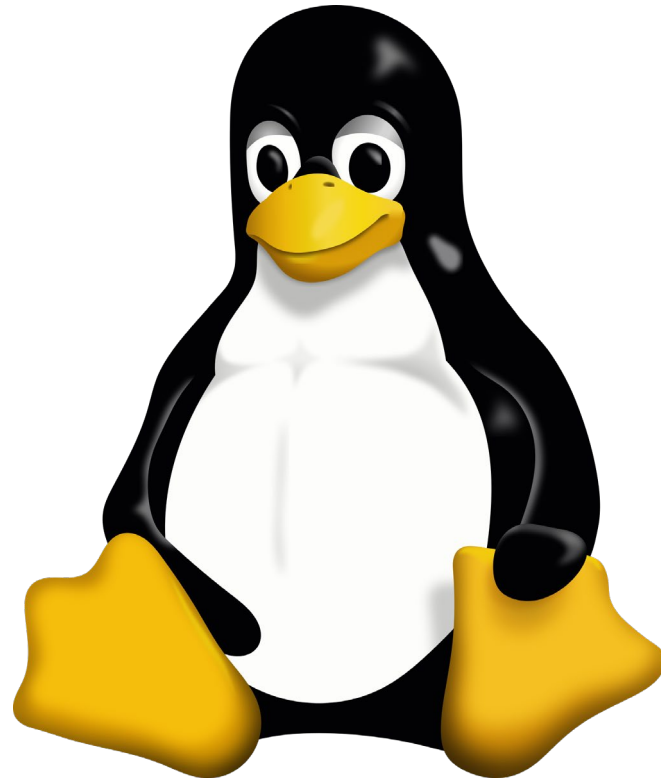
현재 허드 커널을 사용하는 배포판은 데비안 리눅스 배포판 밖에 없다.



# 하드커널



# 리눅스 커널



# GNU/LINUX

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 토발즈**가 만든 리눅스 커널을 지칭을 하며, 리눅스 커널은 GNU소프트웨어하고 같이 사용하기 때문에 보통 **GNU/Linux**라고 부른다.

# 리눅스

리눅스 커널은 현재 안드로이드 및 IoT 그리고 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 토발즈는 거절 하였다.

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.

# 오픈소스 라이선스

현재 리눅스는 다음과 아래와 같이 소프트웨어 구성이 되어 있다. 배포판마다 많이 다르지만, 대다수 상용 리눅스 배포판은 라이선스에 매우 민감하다.

## 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 구성되어 있다.

## MIT Software

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

# 리눅스 커널/셸

USERSPACE/KERNELSPACE



# 커널

커널 아키텍처 및 모듈



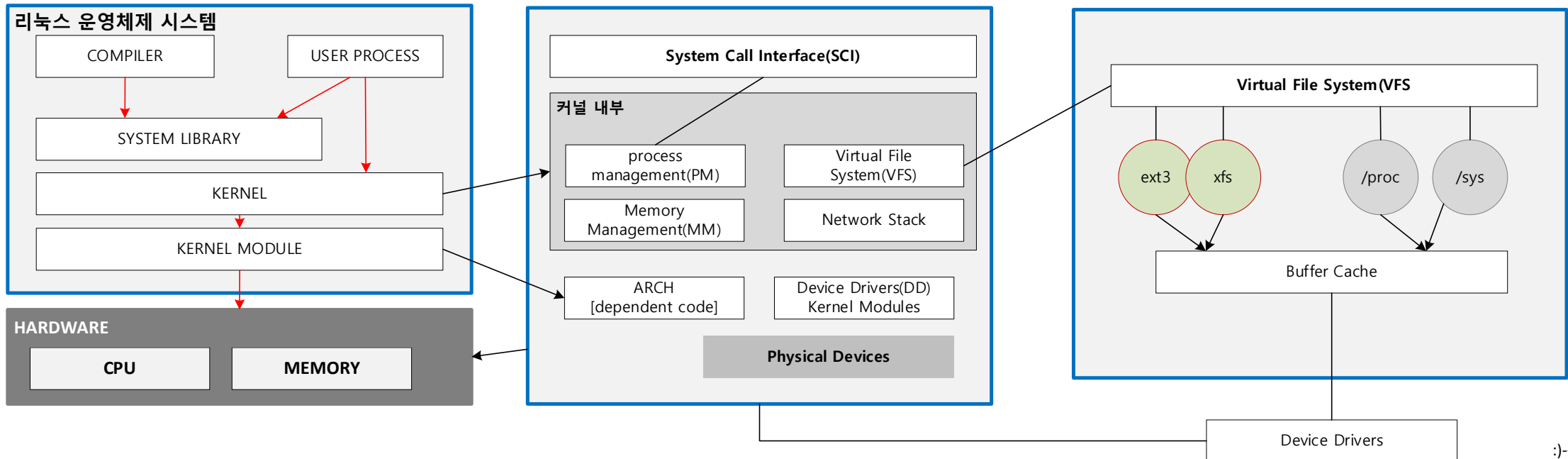
# 커널 아키텍트

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(Monolithic Kernel)** 커널 구조 기반으로 작성이 되었다. 현재는 모듈기능 지원 및 커널 핫 패치(Hot-Patch)를 지원하고 있기 때문에, 모놀릭 커널의 단점을 많이 제거하였다.



# 커널 버전 방식

리눅스 커널은 두 가지 형태로 현재 나누어져 있다.

## LTS Release

LTS버전은 Long Term Support 버전이다. 해당 버전의 소스코드는 5년에서 최장 10년 이상 지원하기도 하다. 해당 LTS기간 부분은 리눅스 토발즈나 혹은 커뮤니티 선택에 따라서 달라진다. (거의 대다수는 리눅스 토발즈 결정에 따르지만...)

## 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.

## Mainline Release

# 커널 버전 방식

**w** = Kernel Version = 4

**xx** = Major Revision = 4

**y** = Minor Revision = 0

**zzz** = Patch number = 45

mainline: <b>6.4-rc7</b>	2023-06-18	<a href="#">[tarball]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a>	<a href="#">[browse]</a>
stable: <b>6.3.9</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>6.1.35</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>5.15.118</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>5.10.185</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>5.4.248</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>4.19.287</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
longterm: <b>4.14.319</b>	2023-06-21	<a href="#">[tarball]</a>	<a href="#">[pgp]</a>	<a href="#">[patch]</a>	<a href="#">[inc. patch]</a>	<a href="#">[view diff]</a> <a href="#">[browse]</a> <a href="#">[changelog]</a>
linux-next: <b>next-20230623</b>	2023-06-23					<a href="#">[browse]</a>

# 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 안들어서"

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’”

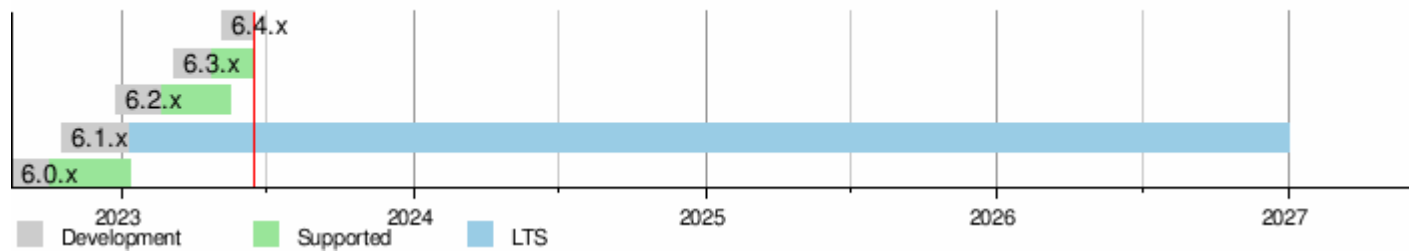
아래 링크에서 넘버링 변경에 대한 이유를 확인 할 수 있다.

[왜 리누즈 토발즈는 넘버링을 변경했는가?](#)

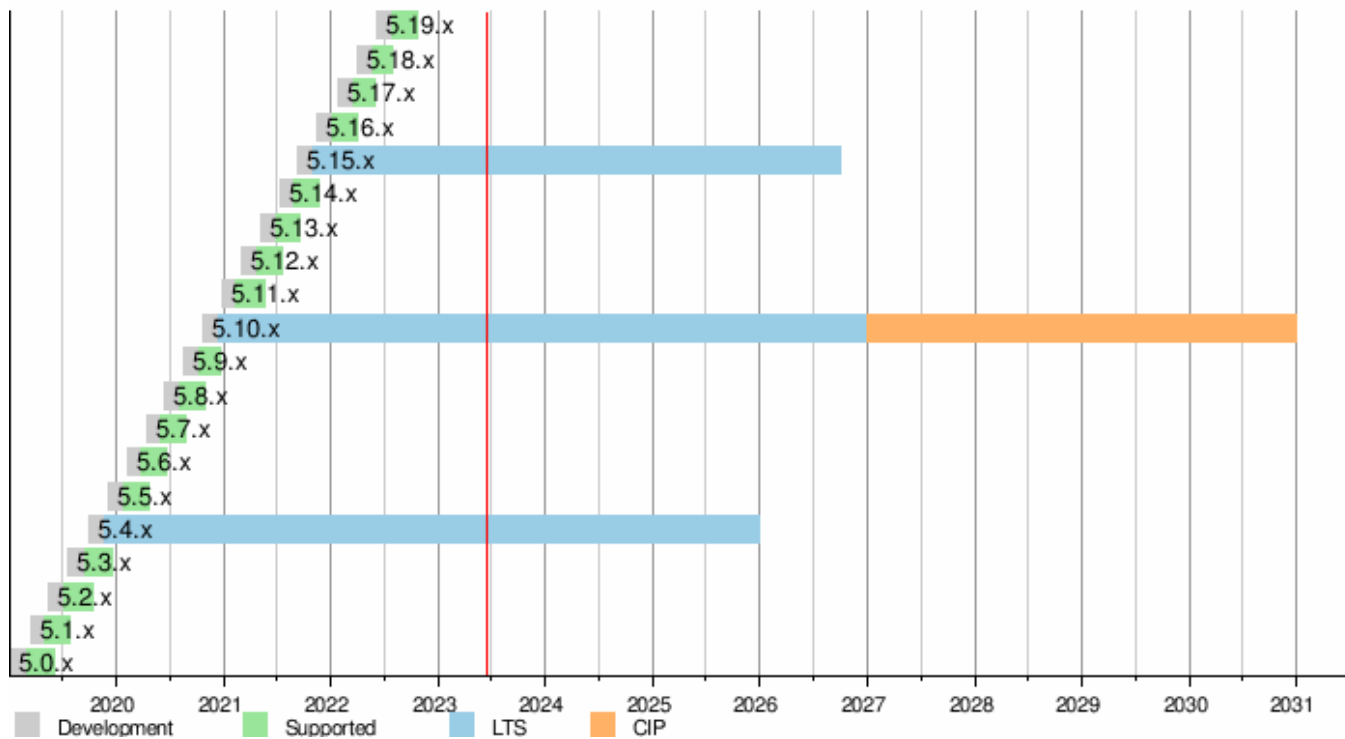
# 커널 버전 방식

- 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.

# 커널 버전 6.x

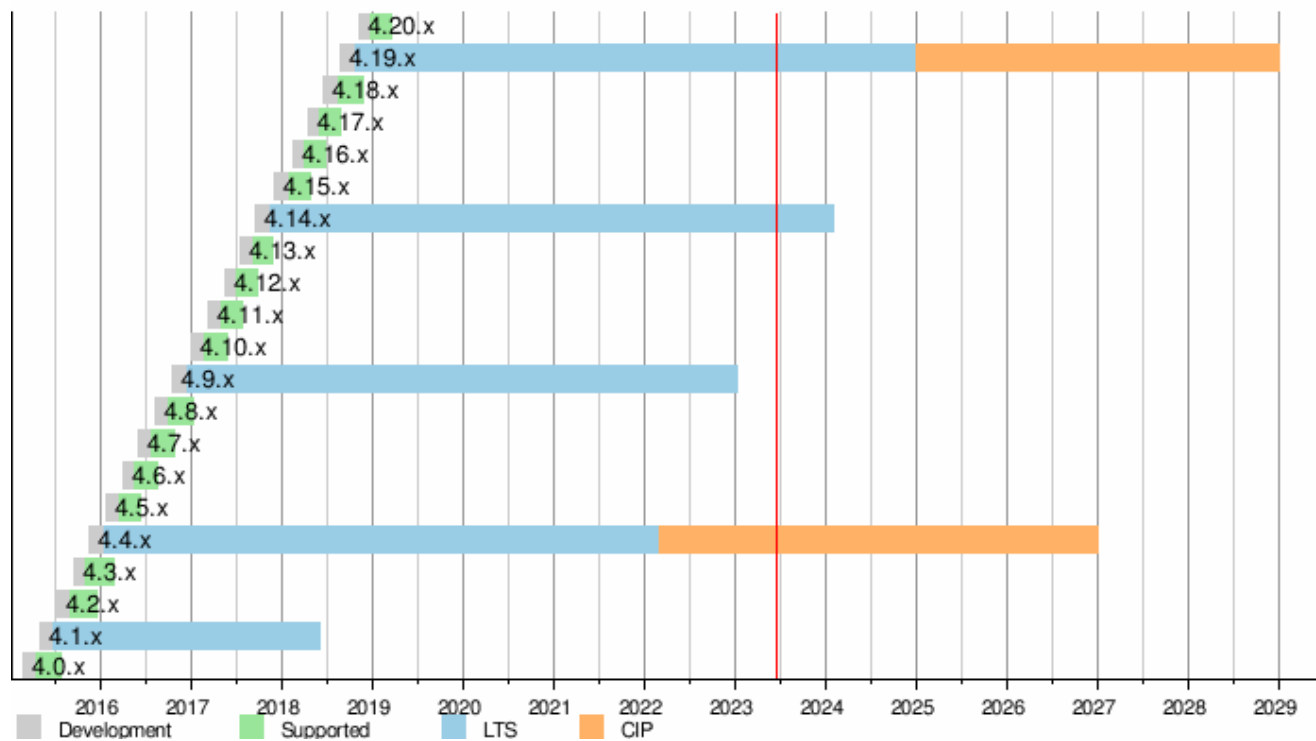


# 커널 버전 5.x

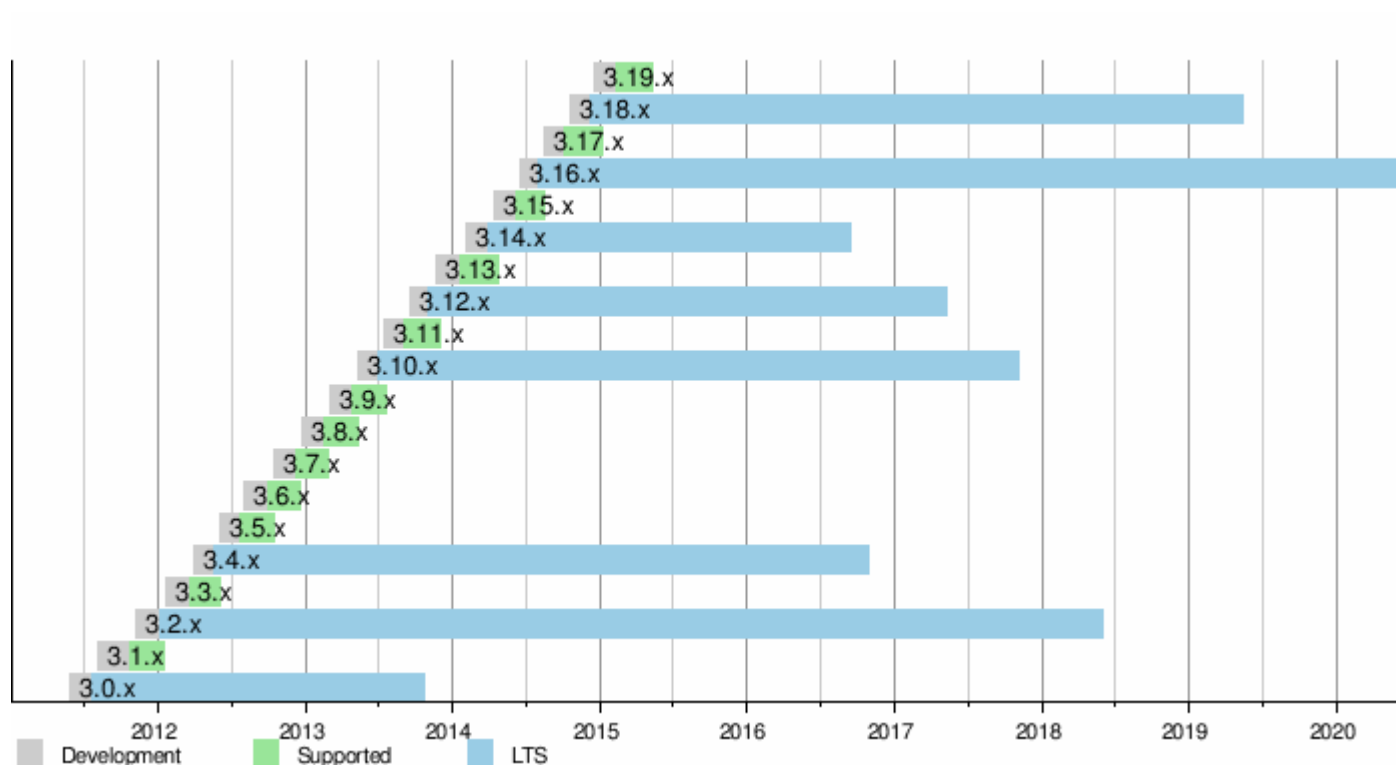




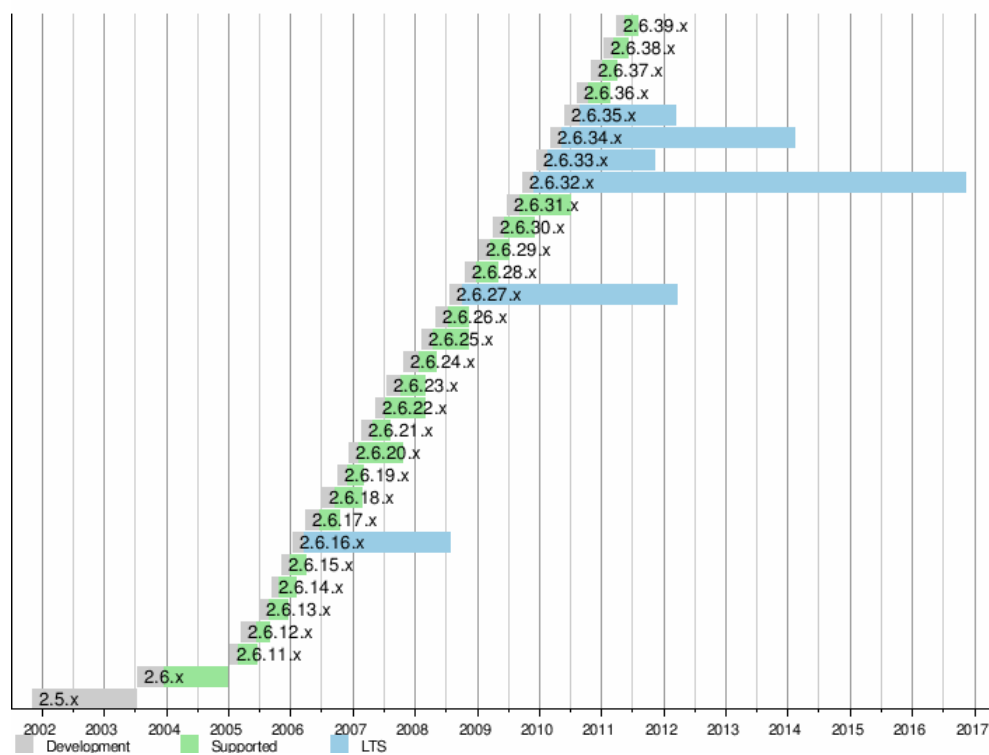
# 커널 버전 4.x



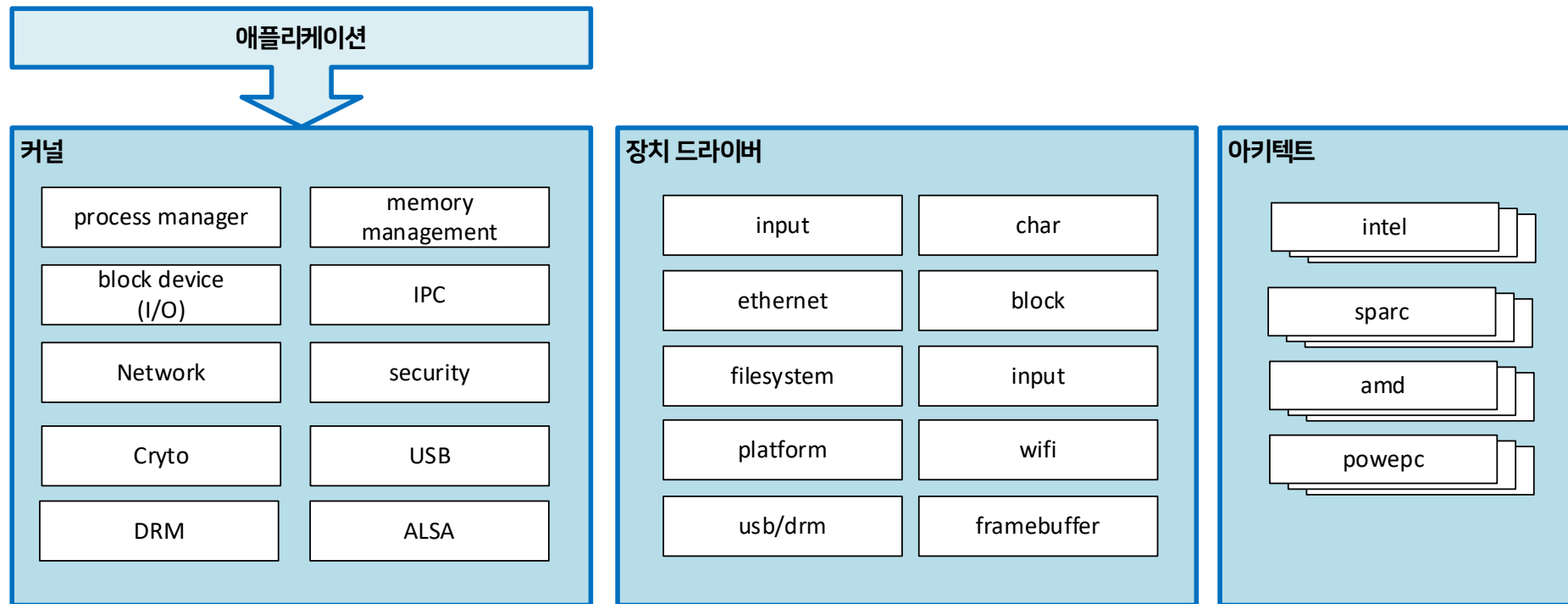
# 커널 버전 3.x



# 커널 버전 2.x

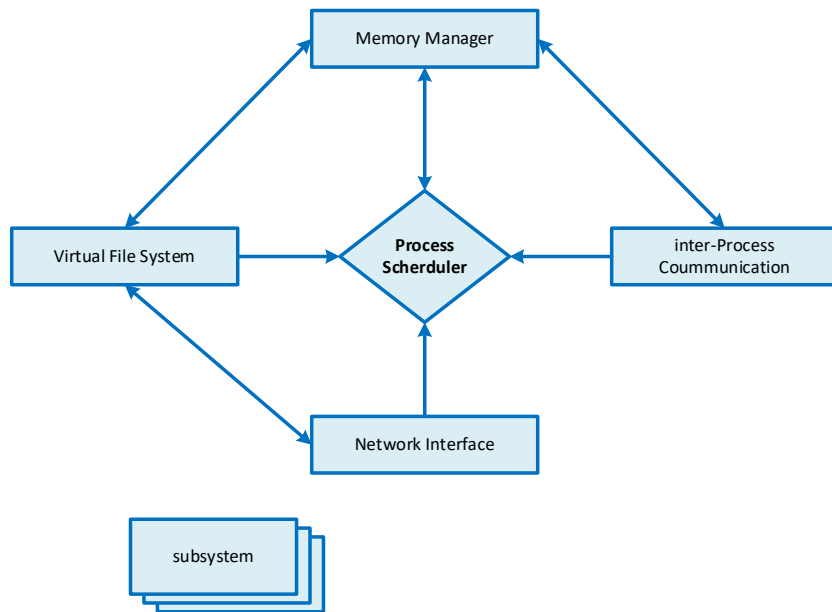


# 커널 아키텍트

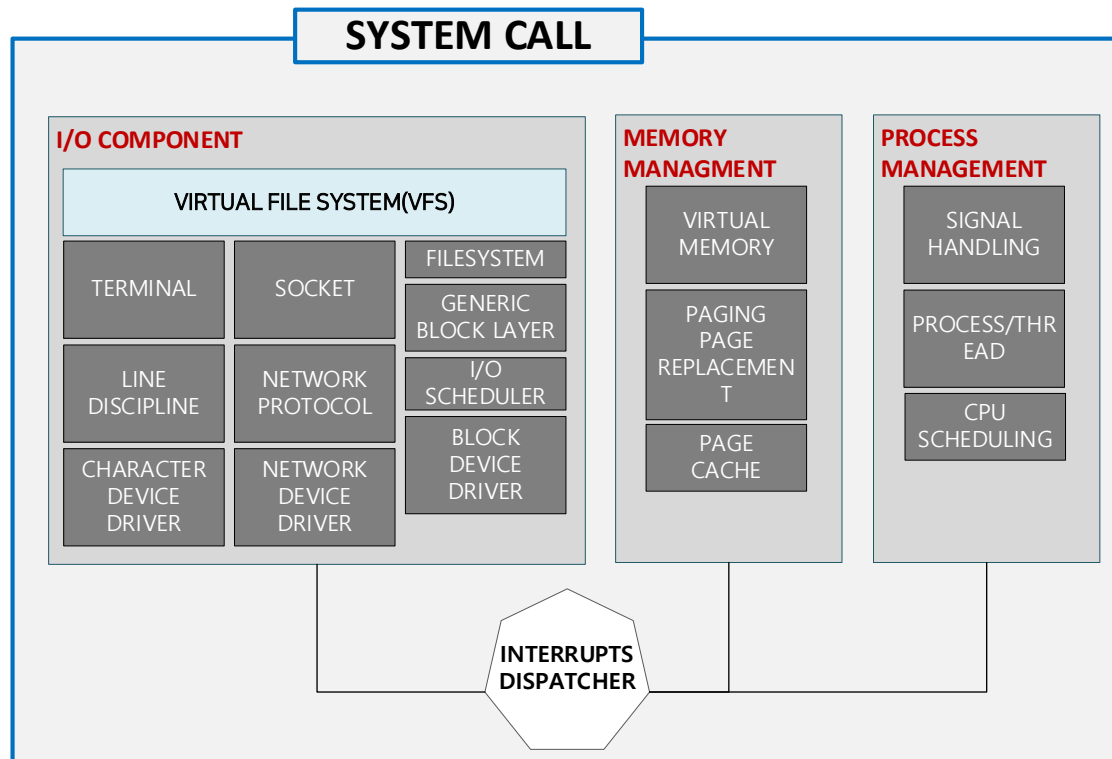


# 커널 아키텍트

리눅스 커널은 모놀리식(Monolithic Kernel) 커널 구조이다. 마이크로 커널 혹은 하이브리드 커널과 달리, 서브 시스템 기반으로 각 자원을 관리 및 사용한다.



# 커널 아키텍트

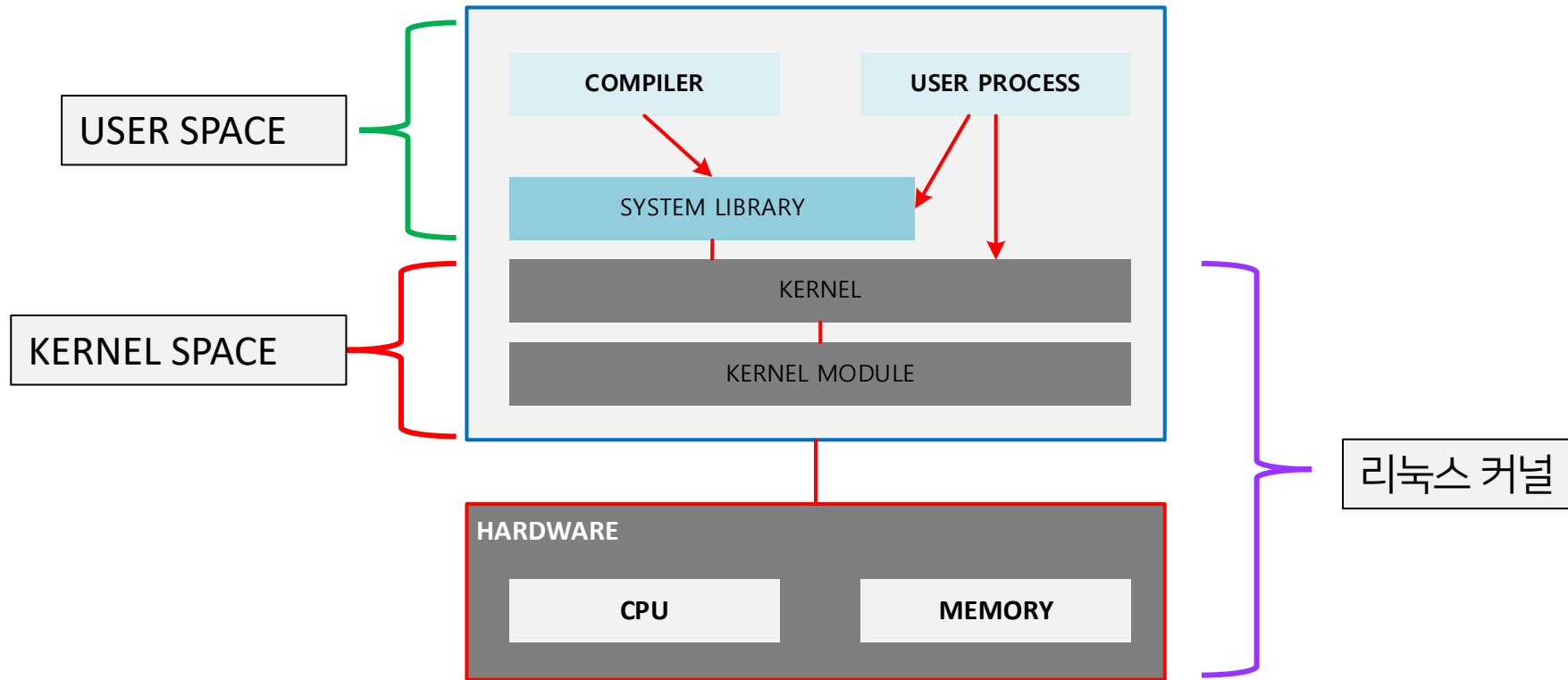


리눅스 시스템 콜은 다음과 같다.

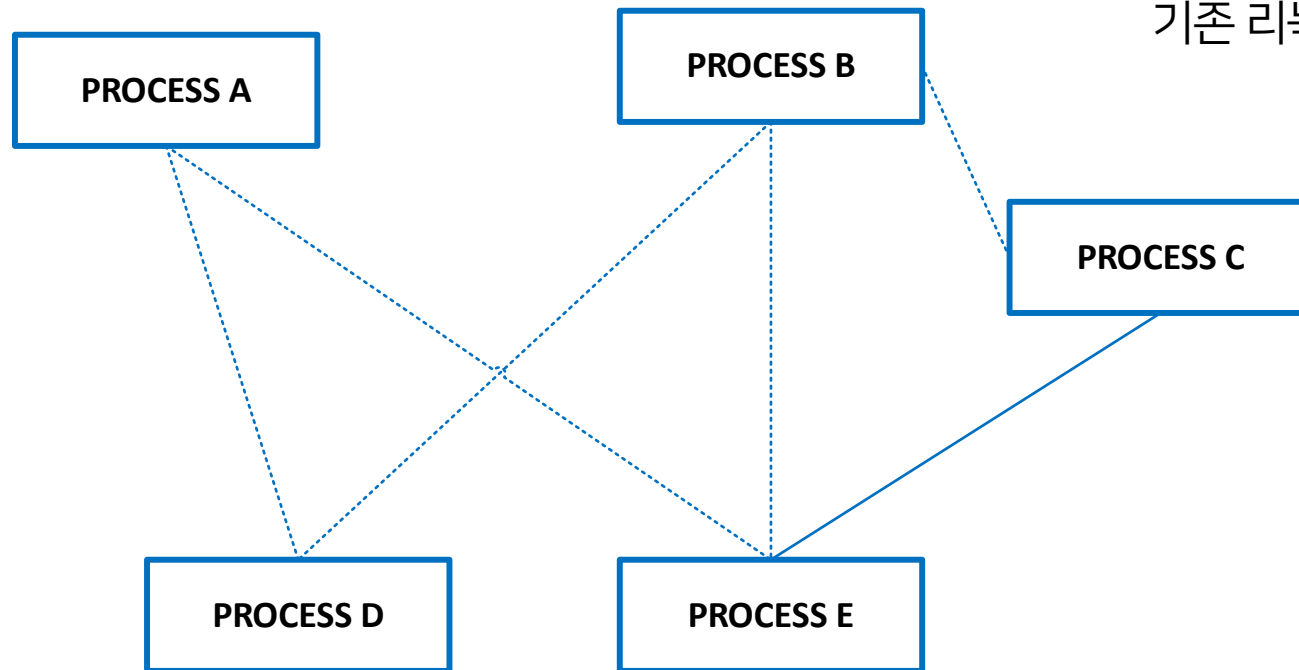
- I/O
- Memory
- Process

각 서버 시스템은 스케줄러가 있으며, 커널은 스케줄러를 통해서 작업을 할당 및 처리한다.

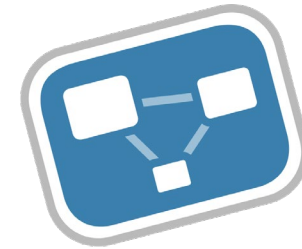
# 커널 아키텍트



# dbus



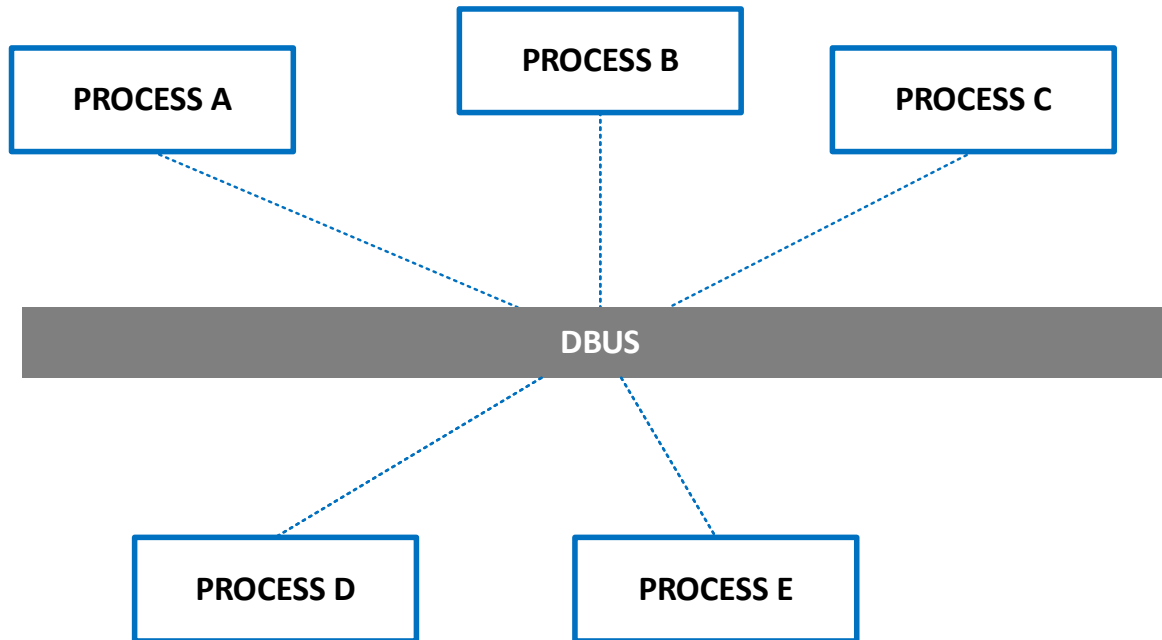
기존 리눅스에서 프로세스끼리 통신하는 방법



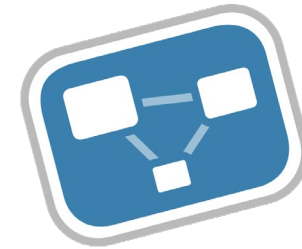
**freedesktop.org**



# dbus

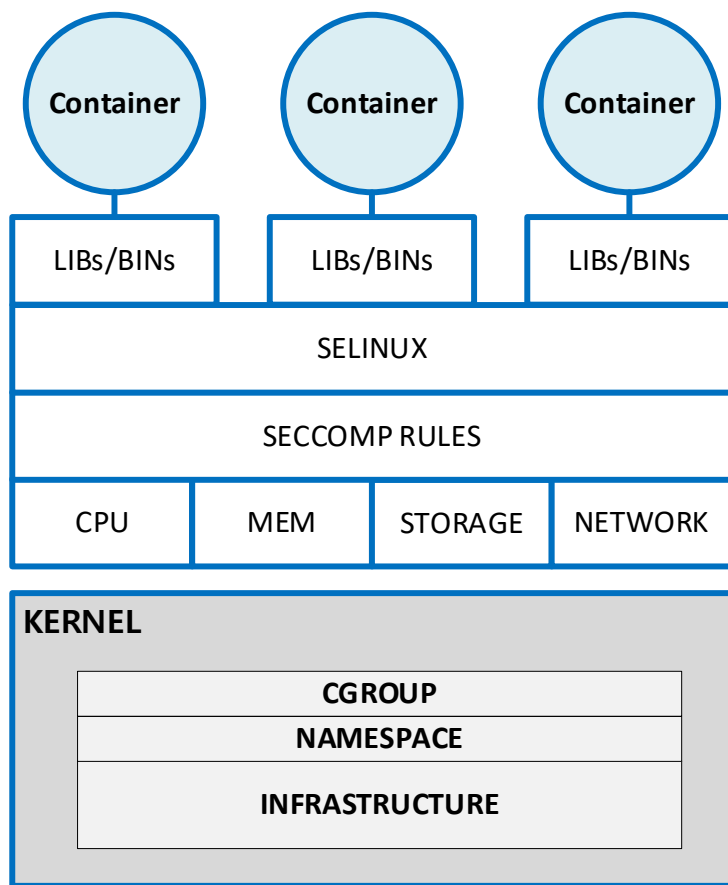


dbus기반으로 프로세스 통신하는 방법. 대표적인 프로그램은 firewalld, gdisk, NetworkManager같은 도구가 있다.

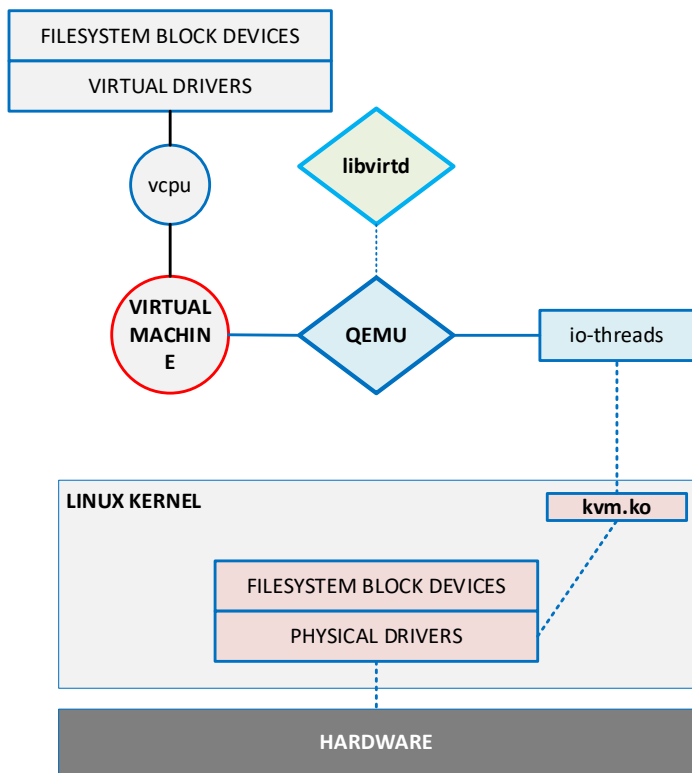


**freedesktop.org**

# 컨테이너



# 가상화



# 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 MIT라이선스 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

일반적인 리눅스 커널 모듈은 전부 "**akmod**"모듈 기반으로 구성이 되어 있다. **akmod**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

**akmods, dkms**는 모듈 다이나믹 커널 모듈이라고 이야기 한다. 커널 컴파일을 별도로 요구하지 않는 부분이 장점이며, 모듈만 컴파일 혹은 바이너리 설치하여 바로 모듈로 사용이 가능하다. 보통 **dkms, akmod**라는 이름으로 배포판에서 제공한다. 다만, 사용하기 위해서 라이브러리 링크 컴파일이 종종 필요하기 때문에 컴파일러 도구가 필요하다.

**장점:** kmod의 보다 사용하기 편한다.

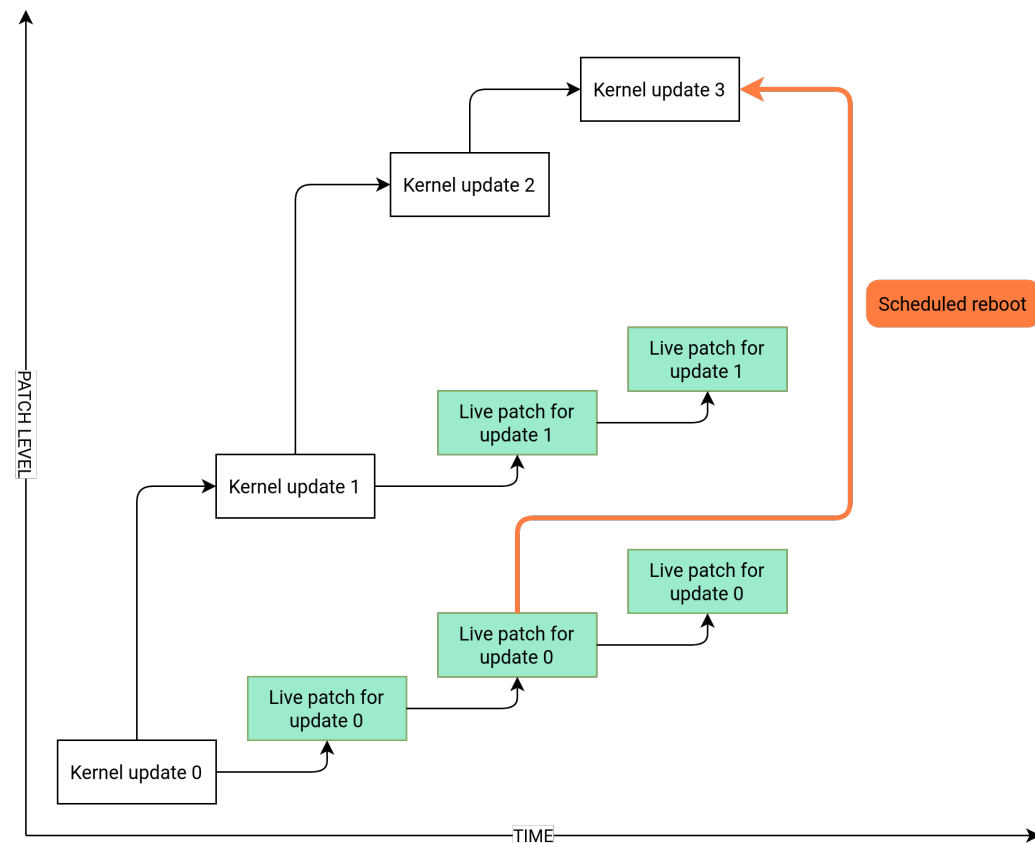
**단점:** 낮은 수준의 드라이버에서는 종종 호환성이나 혹은 사용이 불가능 하다.

# 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 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.

# 셸

사용자 영역

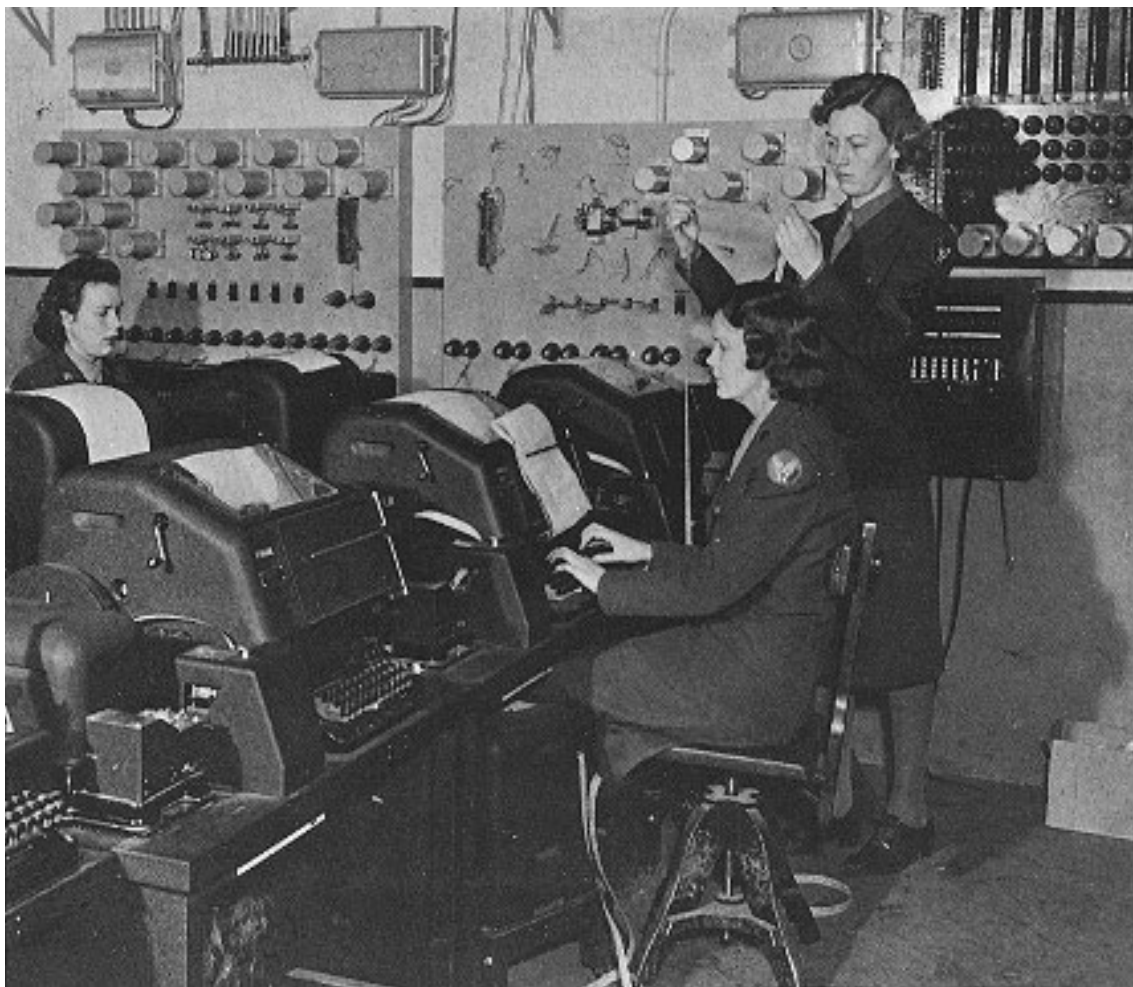
# BASH 무엇인가?

셸은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == SHELL라는 오해를 가지고 있음.  
**PuTTY/iTerm/Xterm**와 같은 터미널 도구가 아님. 터미널에 접근하기 위한 에뮬레이터 도구임.

Bash는 일반적으로 사용하는 CLI 도구와는 다름. BASH는 COMMAND LINE INTERFACE와 비슷하기는 하지만, 대화형 형태로 사용한다. 그래서, BASH셸에서 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

**TTY**는 "Teletypewriter"의 약자. 초기 유닉스 시스템은 로컬만 지원하였으며, BSD의 유닉스 프로젝트를 통해서 TCP/IP를 도입 후, 원격에서 시스템 접근이 필요.

"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."

사진에 있는 시스템은, 과거 텔레 타이퍼 시스템이며, 현재 유닉스 및 리눅스에서 사용하고 있는 시스템도 위와 동일한 개념을 채용하고 있다.

이를 통해서 원격에서 리모트에 있는 시스템에 타이핑이 가능해짐.

# 현재 리눅스에서 제공하는 쉘 목록

현재 리눅스 배포판에서 제공하는 쉘은 다음과 같다. RPM저장소 기반으로 지원하는 목록이며, 다른 배포판은 더 많은 쉘을 제공할 수 있다.

- bash(sh)
- tcsh(csh)
- zsh
- fish
- ksh
- dash
- mksh
- powershell(선택사항)

# 모던 리눅스에서 사용하는 셸

현재 대다수 리눅스 배포판은 더 이상 bash, tcsh, ksh를 사용하지 않는다. 대다수 사용자는 현재 다음과 같은 셸로 환경을 변경하고 있다.

1. zsh
2. fish
3. powershell

powershell, fish는 기존 bash셸과 호환성이 떨어진다. 즉, 두 개의 셸은 POSIX사양을 완전히 따르지 않기 때문에 기존 스크립트 사용이 어려울 수 있다. 하지만 zsh은 POSIX를 지원하기 때문에 기존 셸에서 사용하던 스크립트 사용이 가능하다. 모던 셸 사용을 원하면 zsh기반으로 사용을 권장한다.

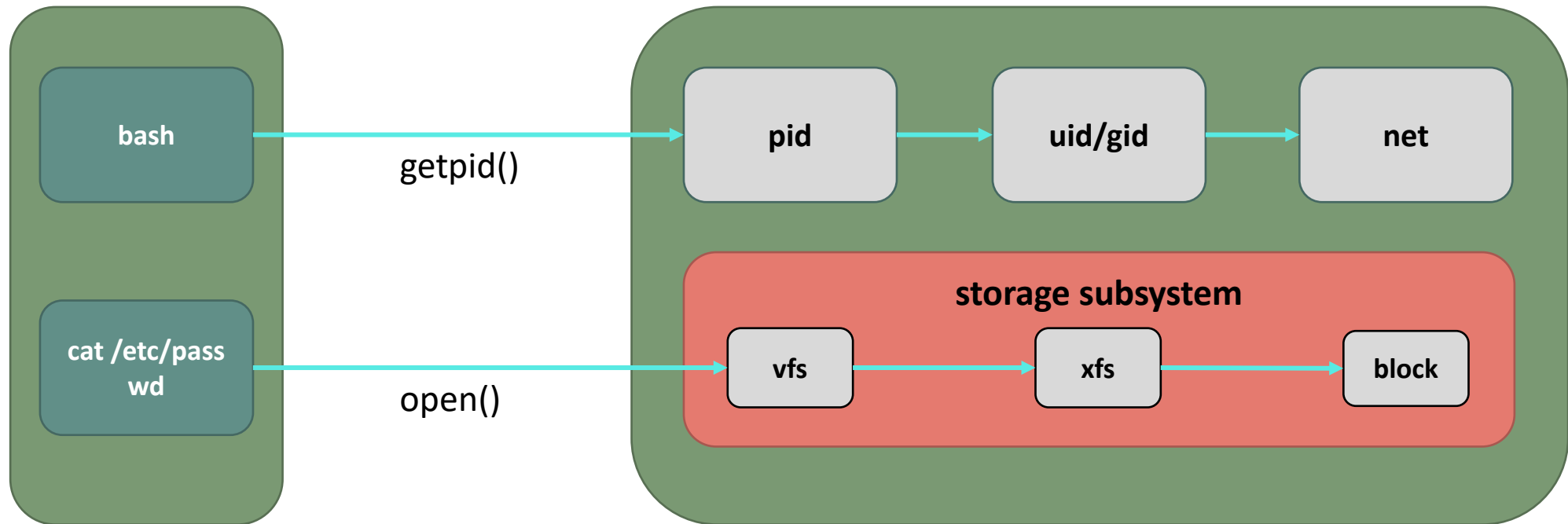
# 모던 리눅스에서 사용하는 셸

```
# dnf install zsh -y
```

```
# chsh -s /bin/zsh
```

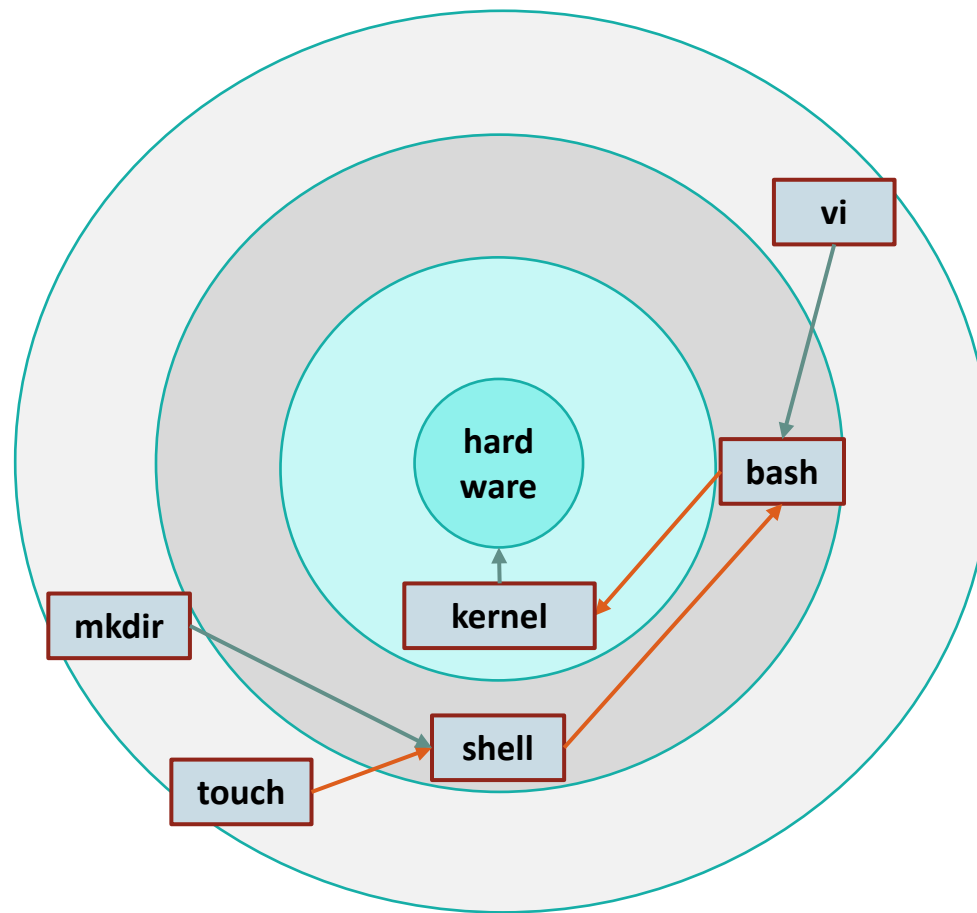
```
# sh -c "$(curl -fsSL  
https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

# BASH 무엇인가?

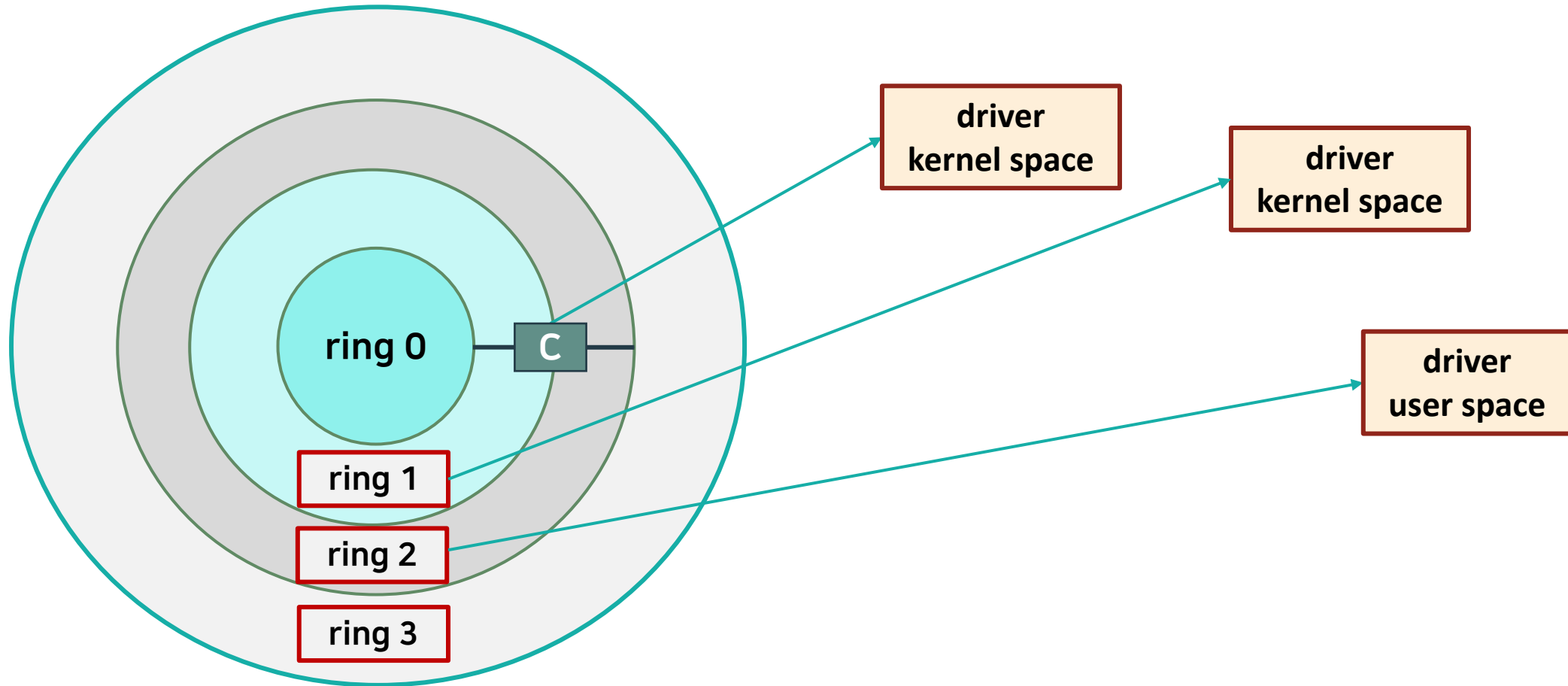




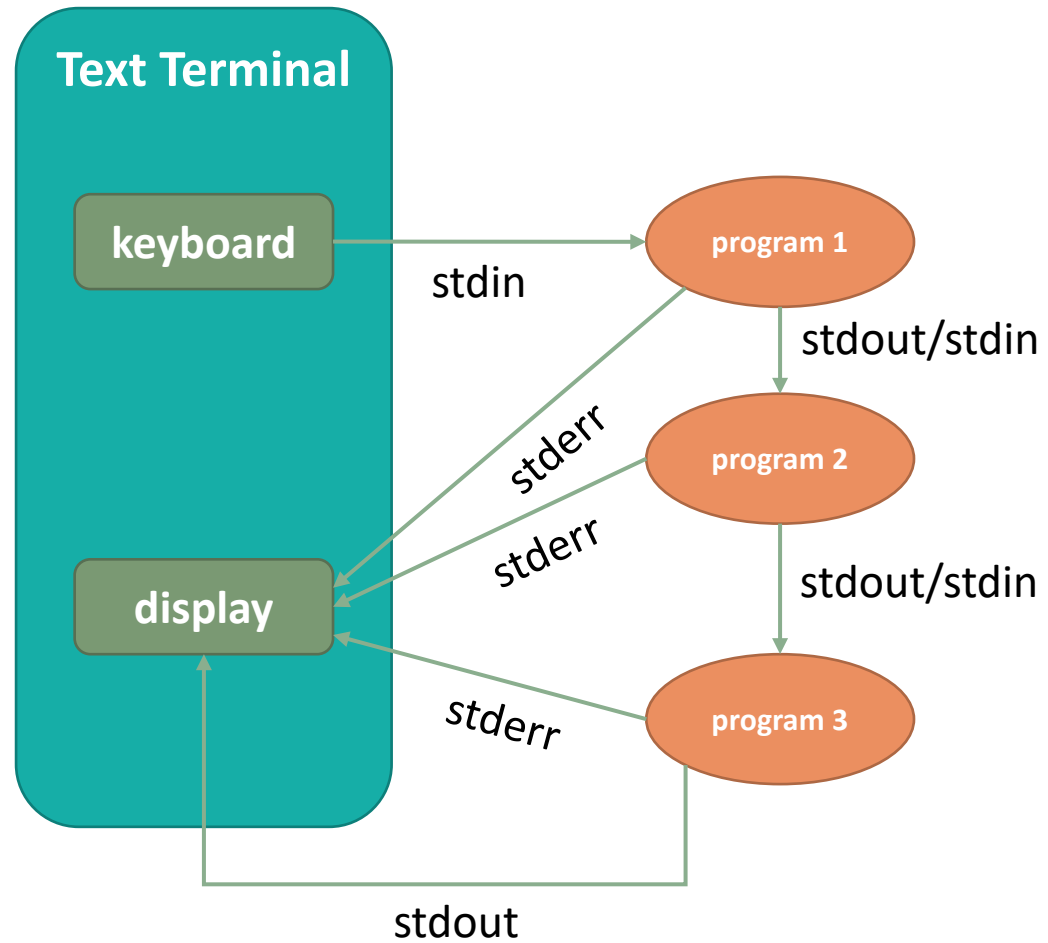
# BASH 무엇인가?



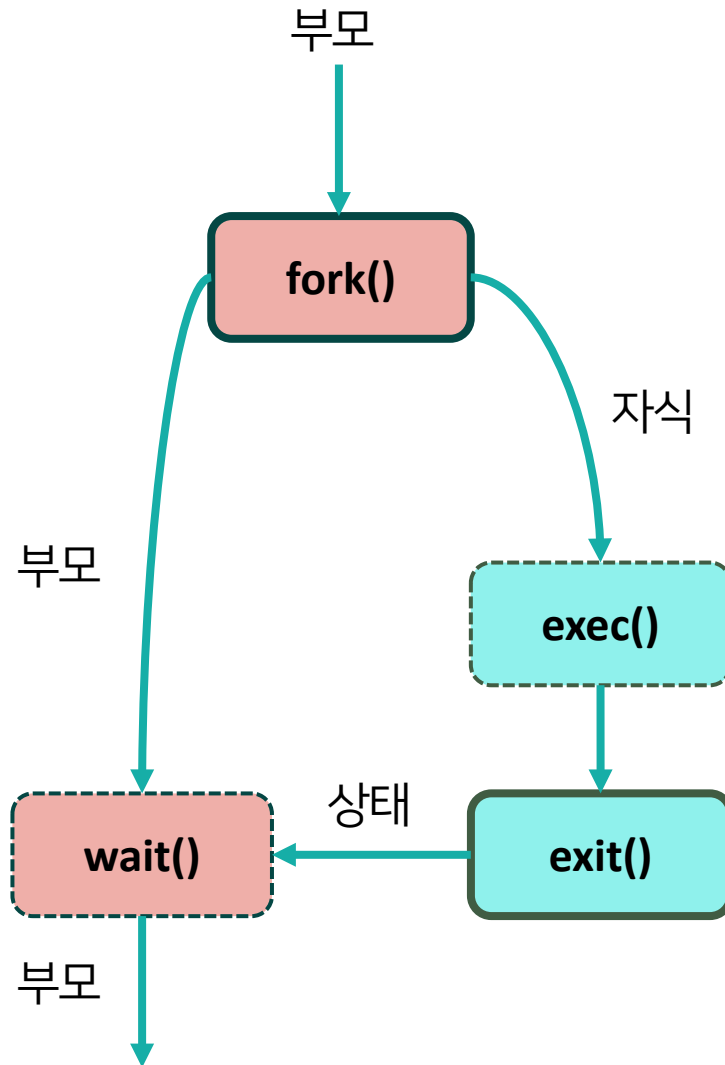
# BASH 무엇인가?



# BASH 무엇인가?

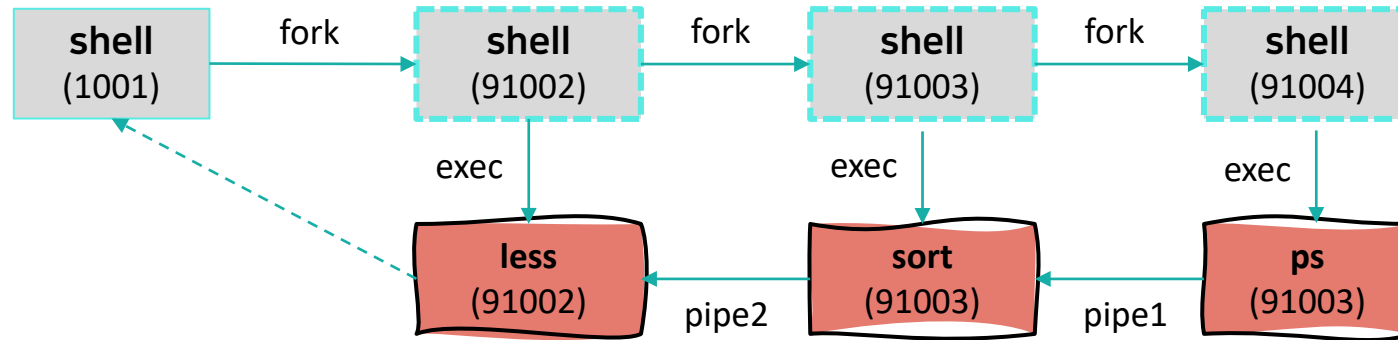


# BASH 무엇인가?



# BASH 무엇인가?

\$ ps | sort | less



# 각 셸의 차이점

거의 대다수의 리눅스는 bash가 설치되어 있음. 다른 말로 GNU/Linux는 BASH가 표준으로 사용하고 있다.

예를 들어서 **FreeBSD/NetBSD/OpenBSD** 그 외 다른 유닉스 들에서는 bash가 아닌 csh, tsh를 사용하기도 한다.

하지만 bash사용이 가능하며, 기존에 사용하였던 csh, tcsh에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 변수	x=3	set x=3	echo \$x	3	3	
	y="\$x 4"	set y="\$x 4"	echo \$y	3 4	3 4	
환경변수	export z=5	setenv z 5	echo \$z	5	5	'setenv z = 5' ?!
	export q="\$z 6"	setenv q "\$z 6"	echo \$q	5 6	5 6	

# 각 쉘의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
Shell & environment variables	z=7	set z = 7	echo \$z; env grep z	7 z=7	7 z=5	Tcsh는 다른 환경에서 같은 이름으로 다른 값을 가질 수 있다.



# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 경로 변수	export PATH=/a: /b	set path=(/a /b)	echo \$path; echo \$PATH; env grep path	- /a:/b -	/a /b /a:/b -	TCHS: path   PATH ?
별칭(별명)	alias ls="ls -l"	alias ls "ls -l"	ls	(same as ls -l)	(same as ls -l)	
명령 프롬프트	PS1=abc-	set prompt=abc-	[ENTER]	abc-	abc-	
재출력	prog > ofile 2> efile	(prog > ofile) >& efile	[ENTER]	(stdout data in ofile; stderr data in efile)	(stdout data in ofile; stderr data in efile)	BASH는 재출력 파일 사용 가능

# 각 셸의 차이점

	BASH	TCSH	출력	BASH	TCSH	비고
셸 로그인 시 사용하는 환경 파일	/etc/profile ~/.bash_profile ~/.bash_login ~/.profile	/etc/csh.cshrc /etc/csh.login ~/.tcshrc ~/.cshrc ~/.login				
셸 시작파일	~/.bashrc	/etc/csh.cshrc ~/.tcshrc ~/.cshrc				
로그아웃 파일	~/.bash_logout	/etc/csh.logout ~/.logout				
스크립팅						Bash혹은 sh스크립트가 제일 사용하기 편함.

# 프로토타입 및 오토메이션

## 프로토타입

bash기반으로 간단하게 스크립트 프로그래밍 가능하다. 목표 중 하나가 **“hack together”**

## 대화형 관리 시스템

bash는 borne-style셸 스타일로 리눅스 및 유닉스 에서 손쉽게 명령어를 사용할 수 있도록 해준다. 양쪽에서 서로 사용이 가능하기 때문에 손쉽게 이식이 가능하다.

## 오토메이션

자주 사용하는 명령어를 스크립트 기반으로 작성하여 간단하게 실행 및 수행이 가능한 방법.

# 프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 한다.

파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다. 이를 보통 파이핑(Piping)라고 한다.

# BASH가 할 수 있는 것.

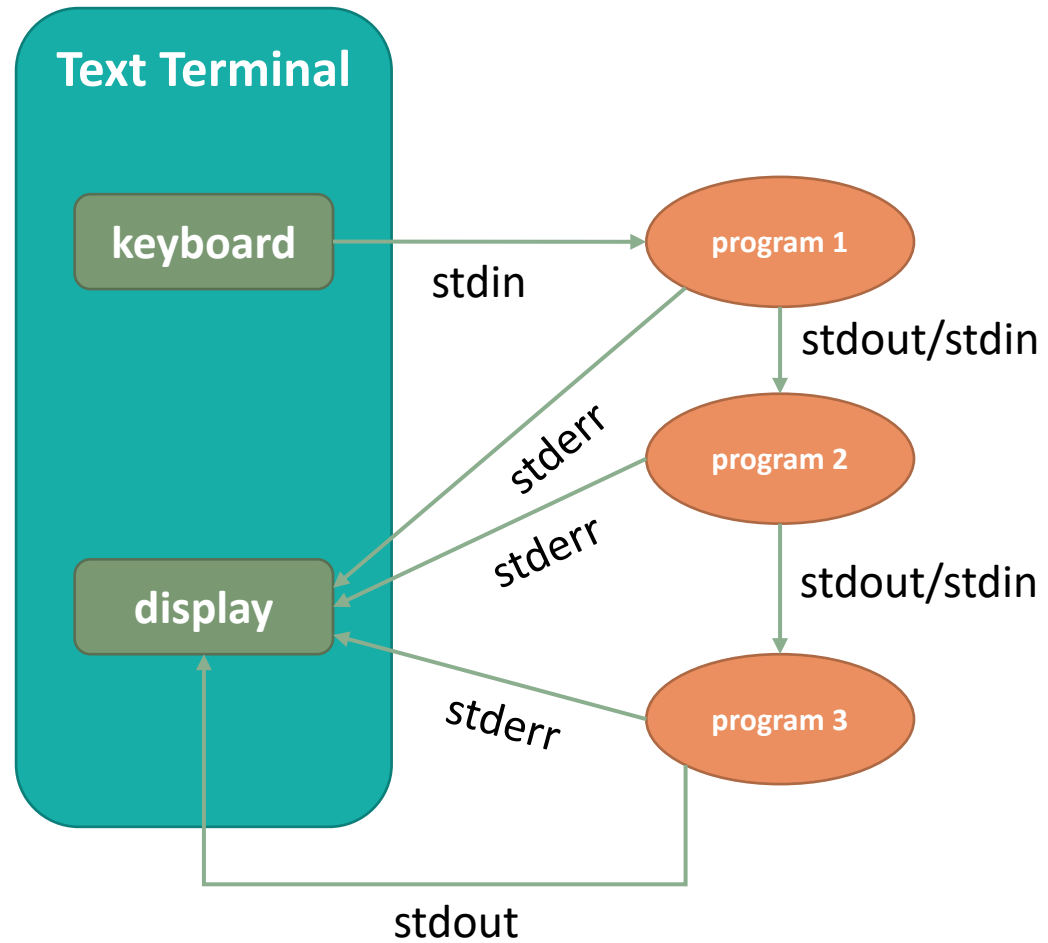
어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다. 이러한 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다. 예를 들어서 **'awk'**, **'sed'**같은 프로그램들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.

# BASH 이동성

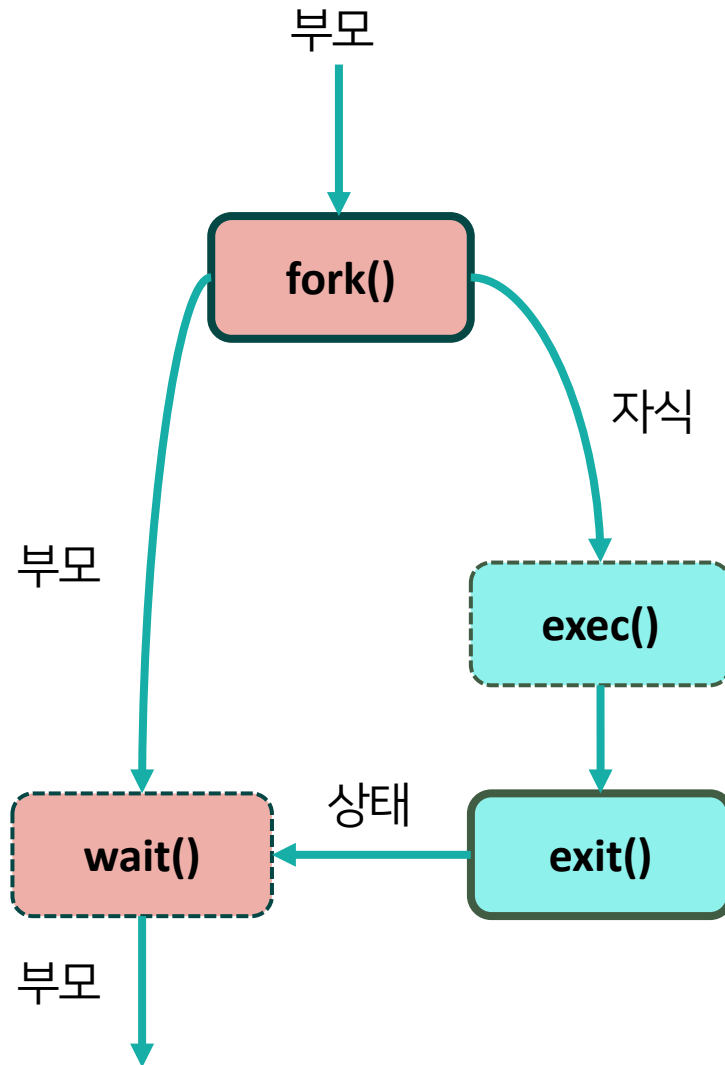
bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다.

이러한 이유로 POSIX셸 스크립트를 표준적으로 지원하며, 각기 다른 런타임 환경에서 사용하는 경우 bash는 강력한 도구가 될 수 있다. 마치, 자바처럼 쉘 프로그램이 이식 되어 있으면 손쉽게 사용이 가능하다.

# BASH 무엇인가?



# BASH 무엇인가?





# 도움말

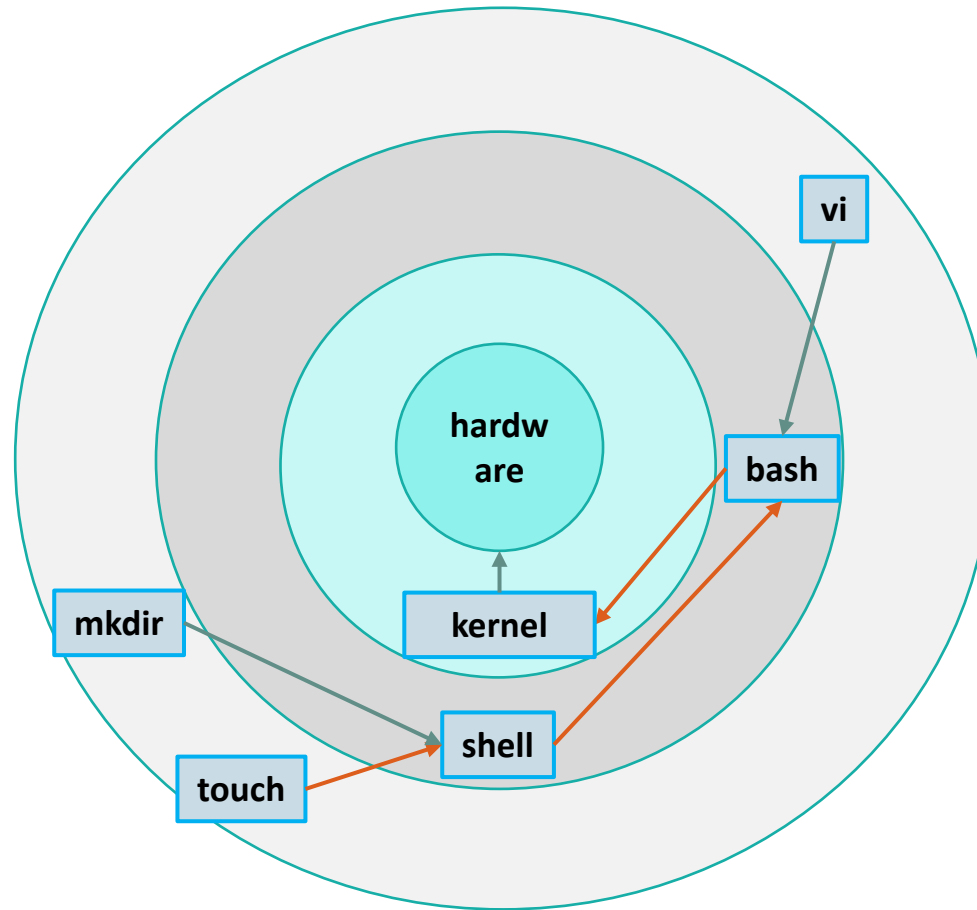
bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다. 내부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

```
bash$ help printf
```

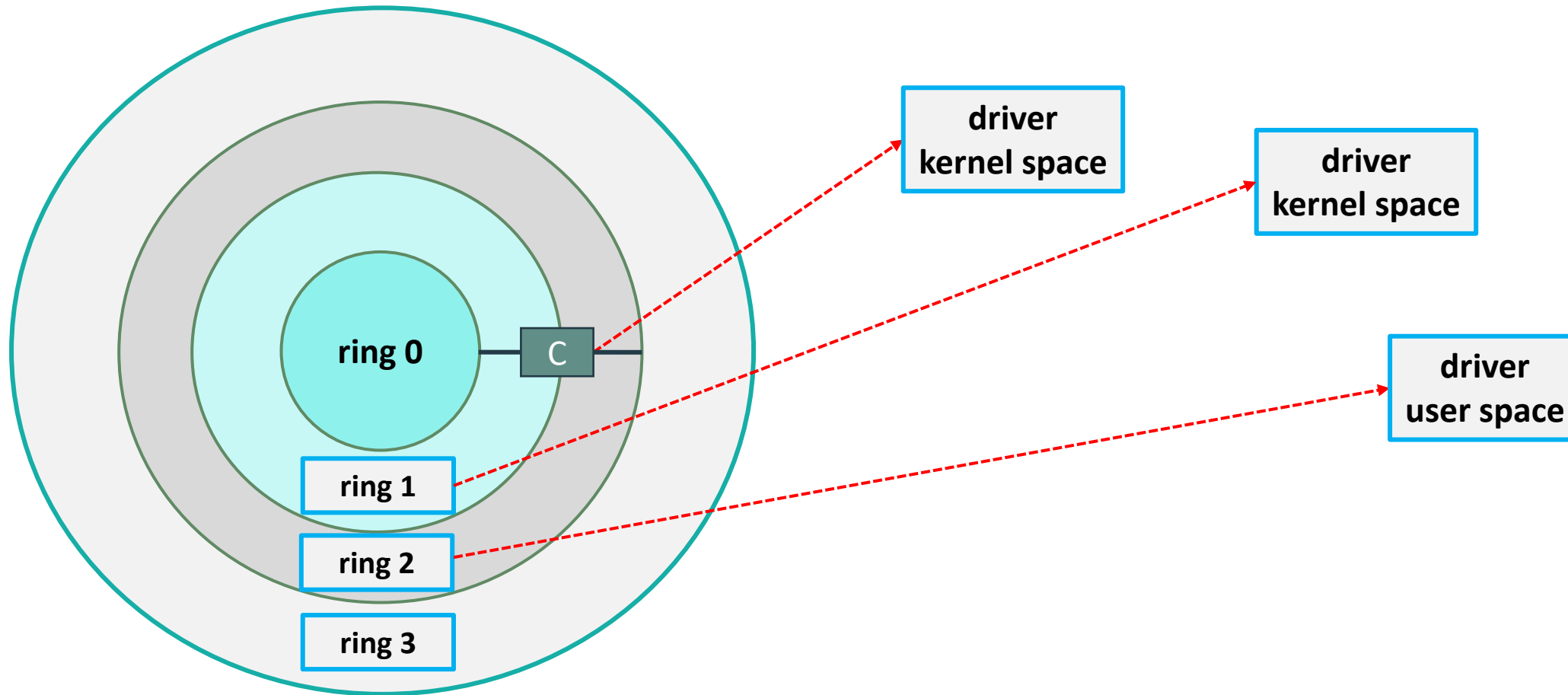
외부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

```
bash$ man printf
```

# Ring Structure



# Ring Structure



# systemD

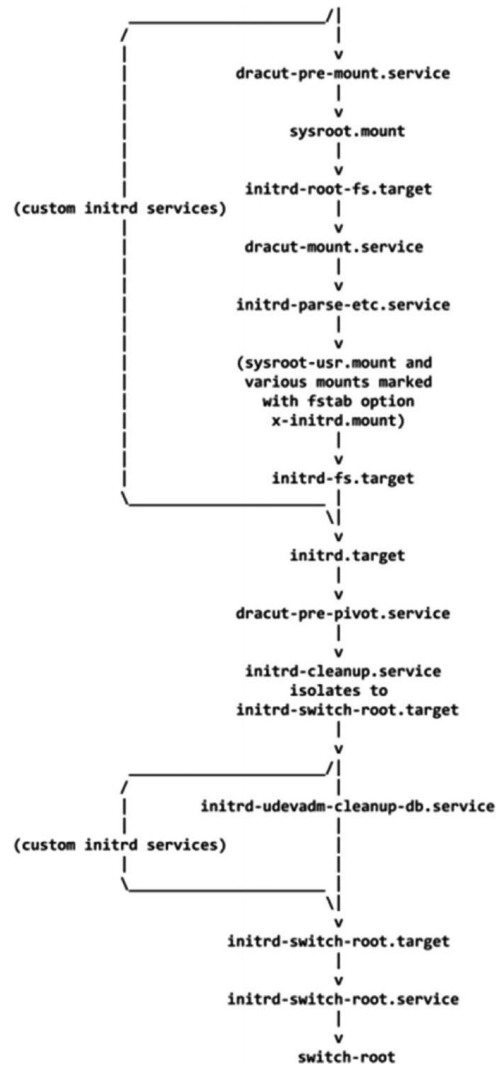
SYSTEM BLOCK



# systemd

[ ● ◀ ] systemd

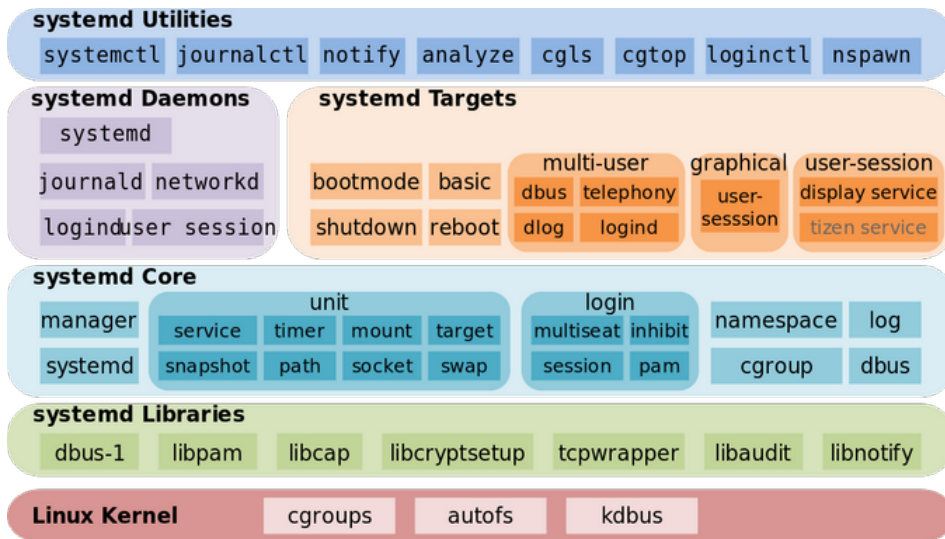
# systemd



systemd는 다음과 같은 구조로 부팅이 된다. target은 일종의 서비스 묶음, 그리고 이전의 런 레벨과 비슷한 역할을 한다.

# systemd

**systemd**는 기존에 사용하던 **system-v**, **up-start**를 대신한다. 현재 사용중인 **LSB(Linux Standard Base)**에서는 **systemd**기반으로 구성이 되어있다. 레드햇 계열의 리눅스는 **systemd**를 채용하고 있다.



# systemd

**systemd**는 **PID 1**번을 대체함. 기존에 사용하던 SystemV3 init, up-start는 더 이상 사용하지 않는다. 기존에 사용하던 **init**들은 서비스들을 스크립트 기반으로 관리하였기 때문에 작업 관리나(여기서 말하는 작업은 프로그램 시작, 마운트 같은 부분) 혹은 자원 분류가 모호하게 되었다.

예를 들어서, 기존에 사용하던 쉘 스크립트는 어떤 서비스가 어떤 파일이나 혹은 디렉터리는 생성/삭제/제거하는지 알 수 없다.



# systemd

현재 systemd는 다음과 같은 서비스들이, 시스템 블록으로 통합 되고 있다.

## hostname

호스트 이름은 **hostnamectl**로 통합이 되고 있다. **hostnamectl**은 **systemd**의 일부분이다. 호스트 이름 이외, 컴퓨터가 사용하는 머신 아이디 및 폼-팩터(Form Factor) 정보를 가지고 있다.

## timedate

호스트의 시간 및 날짜 그리고, NTP를 담당하는 서비스. 앞으로 NTP관련된 부분은 **systemd-timedated.service** 으로 변경될 예정이다.

## crond(timer)

기존 crond 서비스와 비슷하게, 특정 시간에 명령어 실행을 도와주는 서비스 유닛. 기존의 crond데몬은 조만간 **.timer**로 대체가 될 예정이다.

# Systemd 런레벨 정리

- multi-user -> init 3
- graphical -> init 5

**emergency.target, rescue.target** 경우에는 **init**에서 **single** 혹은 **s**와 비슷함. 단, 응급복구 모드인 **rd.break(ramdisk break)** 경우, 이전 ramdisk 기반 복구 기능과 조금 다르다. 런-레벨 확인을 하기 위해서 아래와 같은 명령어로 조회가 가능하다.

```
# systemctl -t target list-units -all
```

```
# systemctl get-defaults
```

```
# systemctl list-dependencies <TARGET_NAME>
```

# systemd

진행 전, **bash completion**이나 혹은 **fish**, **zsh** 셸 기반으로 작업을 권장. bash completion를 사용하기 위해서는 다음과 같이 명령어 실행.

```
# 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
```

# TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
default.target		graphical.target multi-user.target	기본적으로 multi-user.target이나 혹은 graphical.target으로 링크가 되어 있음.
graphical.target	5	runlevel5.target	multi-user에서 GUI로 전환

# TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
	4	runlevel4.target	사 용 하 지 않 음 . 본 래 런 레벨에서는 4 번은 사용하지 않았음. 서비스를 등록하는 경우 <b>multi-user.target</b> 를 수정하지 않는 선에서 사용이 가능하다.
<b>multi-user.target</b>	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-user 기 반 . 하 지 만 , CLI기반으로 부팅이 되며, NFS 및 네트워크 서비스는 시작이 되지 않음.
rescue.target	1	runlevel1.target	읽 기 전 용 상 태 로 싱 글 모 드 (single mode) 로 디스크의 루트 파일 시스템을 램 디스크에 마운트.

# TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
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

# TARGET

systemd 대상이름	시스템 런 레벨	링크 대상	설명
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 명령어 사용법

아래와 같이 명령어를 사용한다.

```
# systemctl status sshd
```

```
# systemctl start sshd
```

```
# systemctl restart sshd
```

# systemctl 명령어

**systemD**에서 관리하는 대다수 자원은 systemctl명령어 기반으로 대다수 유닛을 제어 및 관리한다. systemctl명령어는 **.servic**를 주로 다루지만, 다른 자원 **.timer**, **.mount**등에서도 사용이 가능하다. 자주 사용하는 명령어는 다음과 같다.

**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.

**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.

# journal

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. **rsyslogd**과 프로토콜 호환이 되며, 중앙로그 서버 기능도 제공하고 있다.

```
# systemctl status systemd-journal
```

```
# vi /etc/systemd/journal.conf
```

```
Storage=persistent
```

```
# cp -a /run/log/journal/ /var/log/
```

```
# journalctl -b
```

# journald remote

중앙서버 기능을 사용하기 위해서는 아래 패키지를 설치해야 한다.

```
# dnf install systemd-journal-remote
```

```
# mkdir -p /var/log/journal
```

```
# systemd-tmpfiles --create --prefix /var/log/journal
```

# journalctl 명령어

**systemD**기반에서는 더 이상 **rsyslog**, **syslog**를 사용하지 않는다. 여전히 로그는 **syslog**에도 남기고 있지만, 앞으로 **systemd**기반에서는 **journald**서비스로 로그 기록을 바이너리 데이터베이스로 저장한다. 이를 사용하기 위해서는 **journalctl**명령어로 데이터베이스를 조회하여 유닛 및 커널 관련된 메시지 확인이 가능하다.

제일 큰 장점은 기존에 어려웠던 **메시지 우선순위**를 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.
- \_SYSTEMD\_\***: systemD키워드 명령어를 통해서 자원을 조회한다. 직접 데이터베이스 필드를 선택한다.

# journalctl

**# journalctl --list-boots**

**# journalctl -u httpd -l -f**

**# journalctl -b -1**

**# journalctl -b <BOOT\_ID>**

**# journalctl --since "2023-04-17 12:00:00" --until "2023-04-18 12:00:00"**

**# journalctl --since yesterday**

**# journalctl --since 09:00 --until "1 hour ago"**

# journalctl

```
# journalctl -u httpd.service -u nginx.service --since today
```

```
# journalctl __PID=8080
```

```
# id -u www-data
```

```
33
```

```
# journalctl _UID=33 --since today
```

# journalctl

커널 메시지 확인하기 위해서 아래와 같이 명령어를 사용한다.

**# journalctl -k**

**# journalctl -k -b -5**      **# 5번째 커널 로그**



# journald 메시지 우선순위

**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
```

출력 방법은 변경하고 싶으면 다음과 같이 실행한다.

```
# journalctl -b -u httpd -o json
```

```
# journalctl -b -u nginx -o json-pretty
```

# 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
```

# notify

다른 유닛의 작업이 완료가 될 때까지 대기 후, 해당 작업이 완료가 되면 나머지 작업을 수행한다. 일반적으로 사용자가 notify를 작성하는 경우는 드물다.

```
# 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.

**제일 느린 서비스 순서대로 보여주는 명령어**

**# systemd-analyze blame**

**느린 유닛들 위주로 트리 형태로 출력(target)**

**# systemd-analyze critical-chain**

**전체 부팅 시간 확인은 다음과 같은 명령어**

**# systemd-analyze time**

**이미지 형태로 확인**

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

# cgls/cgtop

**systemd**에서 관리하는 유닛들의 **CPU, Memory, Disk**상태를 추적한다. systemd로 변경이 된 후, 모든 자원들은 cgroup를 통해서 추적 및 감시를 한다. 이전에 사용하던 **cgls, cgset**명령어는 사용이 가능하지만, 가급적이면 systemd기반으로 사용 및 관리 운영을 매우 권장한다.

# systemd-cgls

# systemd-cgtop

# nspawn

**systemd**에서 네임스페이스 및 cgroup기반으로 컨테이너 생성하는 명령어. 현재 대다수 배포판은 flatpak, snap같은 비설치형 애플리케이션 관리자를 제공한다. 이러한 애플리케이션 관리자는 systemd-nspawn기반으로 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
```



# systemD

systemd-timesyncd, ntpd

systemd-timers, crond

# 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
```

```
sourcedir /run/chrony-dhcp
```

```
driftfile /var/lib/chrony/drift
```

```
makestep 1.0 3
```

```
rtcsync
```

```
keyfile /etc/chrony.keys
```

```
ntsdumpdir /var/lib/chrony
```

```
leapsectz right/UTC
```

```
logdir /var/log/chrony
```

NTP서버 정보를 변경 한다.

- pool
- server

이 둘 중 하나를 사용한다. **iburst** 옵션은 네트워크를 통해서 초기화가 가능하면 빠르게 동기화를 시도한다.

# systemd-timesyncd, ntpd

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. CPU클럭이 불안정한 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 ntpd를 사용하지 않음

# systemd-timesyncd, ntpd

**systemd**로 오면서 **systemd-timesyncd**를 통해서 동기화가 가능하다. 파일 위치는 다음 중 둘 중 하나를 사용해도 된다.

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

# systemd-timesyncd, ntpd

```
# dnf install systemd-timesyncd -y
```

```
# vi /etc/systemd/timesyncd.conf
```

```
[Time]
```

```
NTP=0.arch.pool.ntp.org 1.arch.pool.ntp.org 2.arch.pool.ntp.org 3.arch.pool.ntp.org
```

```
FallbackNTP=0.pool.ntp.org 1.pool.ntp.org 0.fr.pool.ntp.org
```

```
ServerName=kr.pool.ntp.org
```

```
RootDistanceMaxUsec=5s
```

```
PollIntervalMinUsec=32s
```

```
PollIntervalMaxUsec=34min 8s
```

```
PollIntervalUsec=1min 4s
```

# crond

시스템 작업 예약은 **at**, **crond**도구를 통해서 가능하다. **at**는 일시적인 서비스 그리고, 오래된 명령어라서 더 이상 사용하지 않는다. 리눅스에서는 **crond**, **anacron**가 통합이 되면서 현재는 **crond**기반으로 사용한다. **crond**를 사용하기 위해서 다음과 같은 명령어로 조회 및 사용이 가능하다.

**# crontab -l**

**# crontab -e -u**

생성된 작업 리소스는 **/var/spool/cron/**에 사용자 이름으로 작업이 생성이 된다. 이 생성된 내용을 **crond**프로세스가 모니터링하여, 매 시간마다 작업을 수행한다.

# timer

레드햇 리눅스 기준, 앞으로 **RHEL 8,9**부터는 **crond**대신 **systemd-timer**기반으로 자원 구성을 권장한다. 현재, 이전에 사용하였던 예약된 시스템 작업들은 대다수가 **systemd**의 **timer**기반으로 이전이 되었다.

**# systemctl list-timers**

Mon 2023-05-15 09:47:35 KST 59min left Mon 2023-05-15 08:16:41 KST 31min ago dnf-makecache.timer dnf-makecac>

Tue 2023-05-16 00:00:00 KST 15h left Mon 2023-05-15 00:59:36 KST 7h ago logrotate.timer logrotate.s>

Tue 2023-05-16 07:42:31 KST 22h left Mon 2023-05-15 07:42:31 KST 1h 5min ago systemd-tmpfiles-clean.timer systemd-tmp>

# .timer example

타이머 기반으로 예약 작업을 구성하면 아래와 같이 설정 파일을 작성한다.

```
# vi /etc/systemd/system/test.timer
```

```
[Unit]
```

```
Description=Run foo weekly and on boot
```

```
[Timer]
```

```
OnBootSec=15min
```

```
OnUnitActiveSec=1w
```

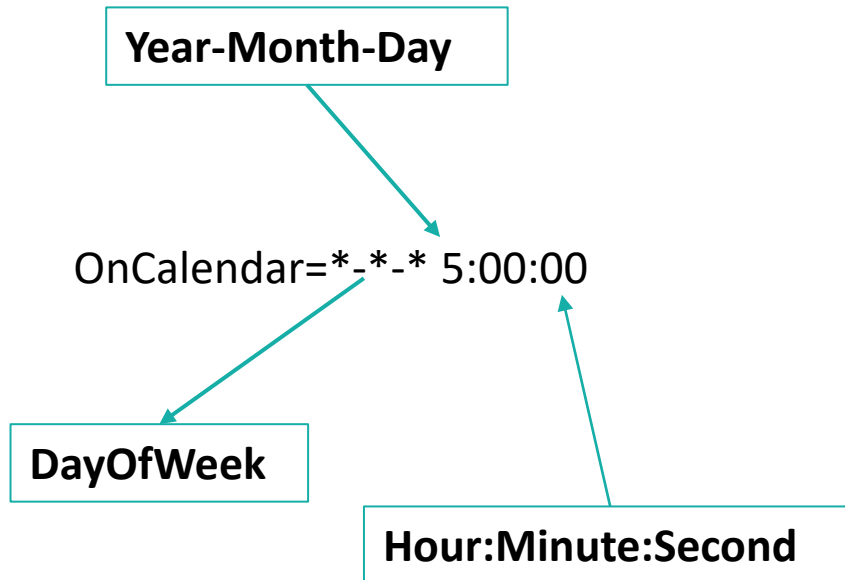
```
[Install]
```

```
WantedBy=timers.target
```



# systemd-timers, crond

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



# systemd-timers, crond

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

```
# systemd-run --on-active=30 /bin/touch /tmp/ihatesystemd
```

```
# systemd-run --on-active="30m" --unit wildfly.service
```

# 시스템

SELinux

네트워크

프로세스 관리

디스크 관리

네트워크

사용자 관리

파일 관리

# SELINUX

시스템

# SELINUX

**SELinux**는 미국 **NSA**에서 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.

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

# SELINUX

## getenforce

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 수정한다.

**/etc/selinux/config**

SELINUX=enforcing

SELINUXTYPE=targeted

**enforcing:** 강제로 SELinux 정책 적용.

**permissive:** 감사만 하며, 정책은 적용하지 않음.

**targeted:** 프로세스 중심으로 정책 적용.

**mls:** 다중 계층 보안으로, 각각 등급별로 접근하는 영역을 다르게 한다.

**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
```



# SEMANAGE

일반적으로 보통 **-l** 옵션은 "**list**"이다. **-C** 옵션은 "**Customized**" 옵션이다. 사용자가 수정하거나 혹은 변경한 부분만 출력한다. **-a**는 "**add**" **selinux policy**파일에 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.

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

```
# setsebool xdm_write_home=1
```

변경된 내용에 대해서 확인하기 위해서는 다음과 같이 실행한다.

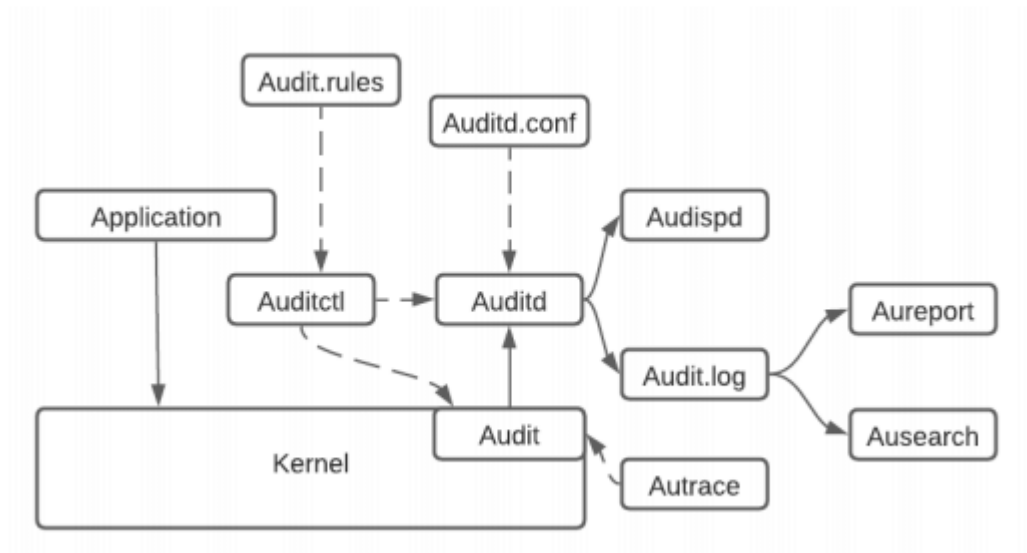
```
# semanage boolean -lC
```

-l: list

-C: Customized

# AUDIT

Audit는 시스템에서 발생하는 시스템 콜, 사용자, 파일 및 디렉터리 같은 자원에 대해서 감사한다.



# AUDIT

특정 프로그램에서 실행하는 모든 콜 확인

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

특정사용자가 파일 접근(open)확인

```
# auditctl -a exit,always -S open -F auid=510
```

성공하지 않는 콜 확인하기

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

# 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
```

# 디스크 관리

기본 명령어 및 개념

LVM2

STRATIS

VDO

# block command

## lsblk

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

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

## blkid

슈퍼 블록에서 생성된 UUID 정보를 확인한다.

## fdisk

MBR 형태로 디스크에 파티션을 생성한다. 프라이머리는 최대 4개까지 생성, 확장 파티션을 운영체제 별로 다르기는 하지만 보통 32개까지 가능하다.



# block command

## gdisk

EFI(GPT)형태로 디스크에 파티션을 생성한다. 최대 128개까지 생성이 가능하다.

## partprobe

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

# block command

## kpartx

단일 디스크 혹은 파티션을 추가 및 삭제하는 명령어

## parted

CLI기반으로 디스크의 파티션 관리를 한다.

# block command

## sfdisk

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

## cdisk

TUI기반으로 사용이 가능한 텍스트 에디터.

## hwinfo

하드웨어 정보를 확인하는 명령어. 블록 장치를 확인 시 사용하는 명령어는 다음과 같다.

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

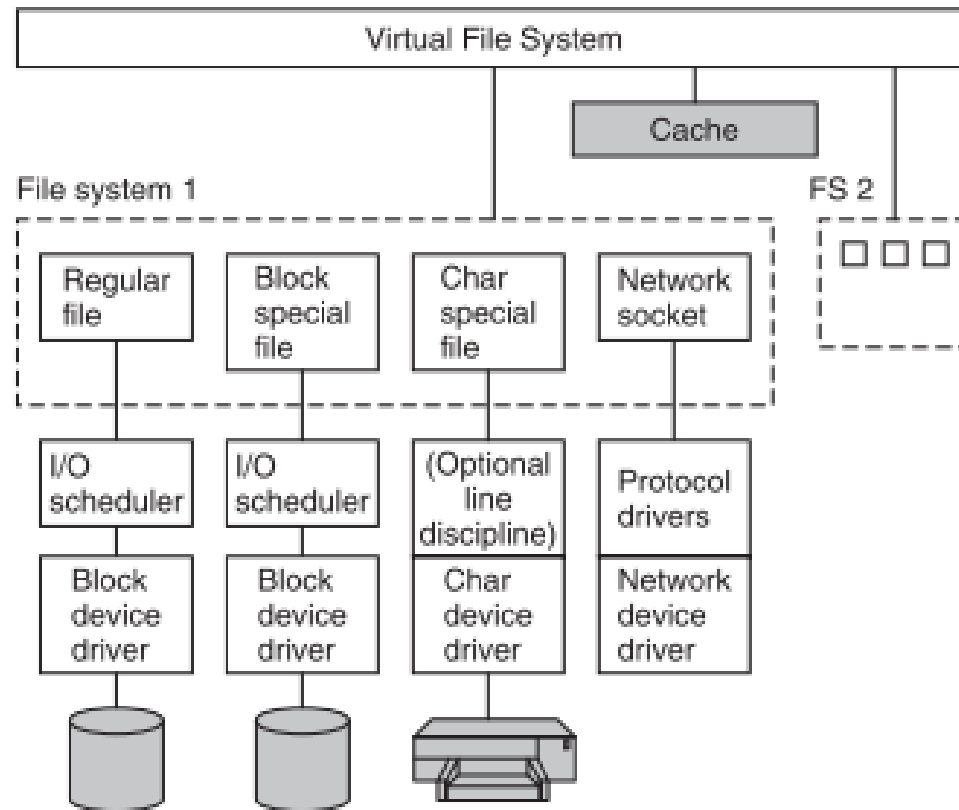
# 디스크 관리를 위한 명령어

# 디스크 관리를 위한 명령어

디스크 관리를 위해서 많이 사용하는 명령어는 다음과 같다.

- lsblk
- blkid
- fdisk/gdisk
- (k)partx, partprobe

# block



**Figure 10-22.** The Linux I/O system showing one file system in detail.

# lsblk

블록 장치를 확인하기 위해서는 **lsblk** 명령어로 확인한다. lsblk는 커널에서 인식이 되어 있는 블록장치를 화면에 출력한다.

```
# lsblk
```

블록장치 즉, "vda", "sda", "had" 같은 장치들은 사용이 가능한 이름으로 출력이 된다.

# blkid

# blkid

/dev/mapper/cs-swap: UUID="4b8975a5-a743-42cd-994f-8d3a53947ac3" TYPE="swap"

/dev/mapper/cs-home: UUID="29e33d2d-3fcb-462b-a3f0-0792fd30e444" TYPE="xfs"

/dev/mapper/cs-root: UUID="f0905739-d81a-4125-a979-3899db37bb59" TYPE="xfs"

/dev/sda2: UUID="5ba71bf0-9077-48d5-873d-0deea8f9ddcd" TYPE="xfs" PARTUUID="018ef963-eff2-4a29-bc02-31413bc4a961"

/dev/sda3: UUID="xgsInx-UnKy-EdwC-ox4g-u1o6-fGDs-bNt51T" TYPE="LVM2\_member"  
PARTUUID="f48a7760-a99d-4f8f-bd5f-a9c964db06b8"

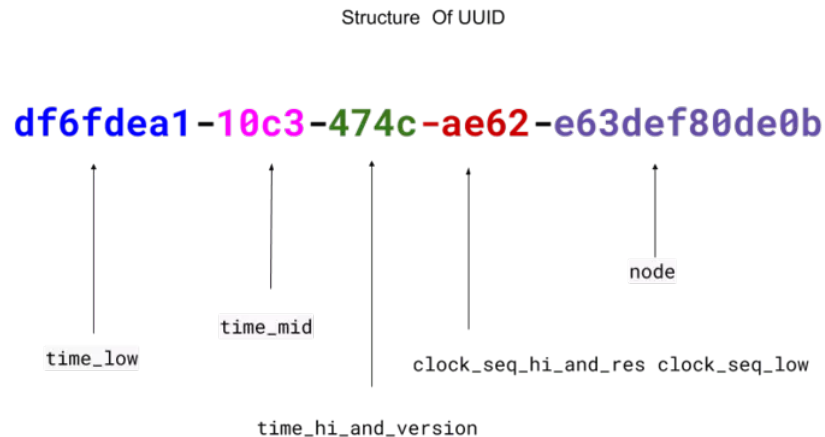
/dev/sda1: UUID="A9BD-E0B5" TYPE="vfat" PARTLABEL="EFI System Partition"  
PARTUUID="55f6c500-0ffa-47e7-af6a-fd7df90ec4c5"



# 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 유래가 되었다. 현재 사용중인 UUID는 버전 5이다.



# blkid

## PARTUUID

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

## DISKUUID

disk-uuid는 일반적으로 /etc/fstab에 등록된 내용. 보통 특정 파티션에 링크가 되어 있다.

위의 심볼릭 링크는 /dev/disk에서 확인이 가능하다.

# lsblk

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPPOINTS
sda	8:0	0	12G	0	disk	
sda1	8:1	0	1G	0	part	/boot
sda2	8:2	0	11G	0	part	
cs-root	253:0	0	9.8G	0	lvm	/
cs-swap	253:1	0	1.2G	0	lvm	[SWAP]
sdb	8:16	0	10G	0	disk	
sdc	8:32	0	10G	0	disk	
sr0	11:0	1	1024M	0	rom	
sr1	11:1	1	374K	0	rom	

# 디스크 파티셔닝

gdisk/fdisk/cfdisk/parted

parted, partprobe

# 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?

파티션은 보통 다음과 같은 네이밍 규칙을 가지고 있다.

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

리눅스 SCSI는 최대 16개의 마이너 숫자(minor 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 명명

리눅스 배포판별로 블록장치 생성 개수는 다르지만, 일반적으로 128개 정도의 장치를 생성한다. 수세나 혹은 다른 배포판은 16개 혹은 30개 정도 생성하며, mknod, MAKEDEV같은 명령어로 추가적으로 생성이 가능하다. 커널에서 인식이 된 블록 장치들은 /proc/partitions, **/dev/disk/**에서 장치 정보들이 생성 및 공유가 된다.

디스크에 관련된 커널 파라미터는 **/sys/class/scsi\_host/**에서 조정이 가능하다.

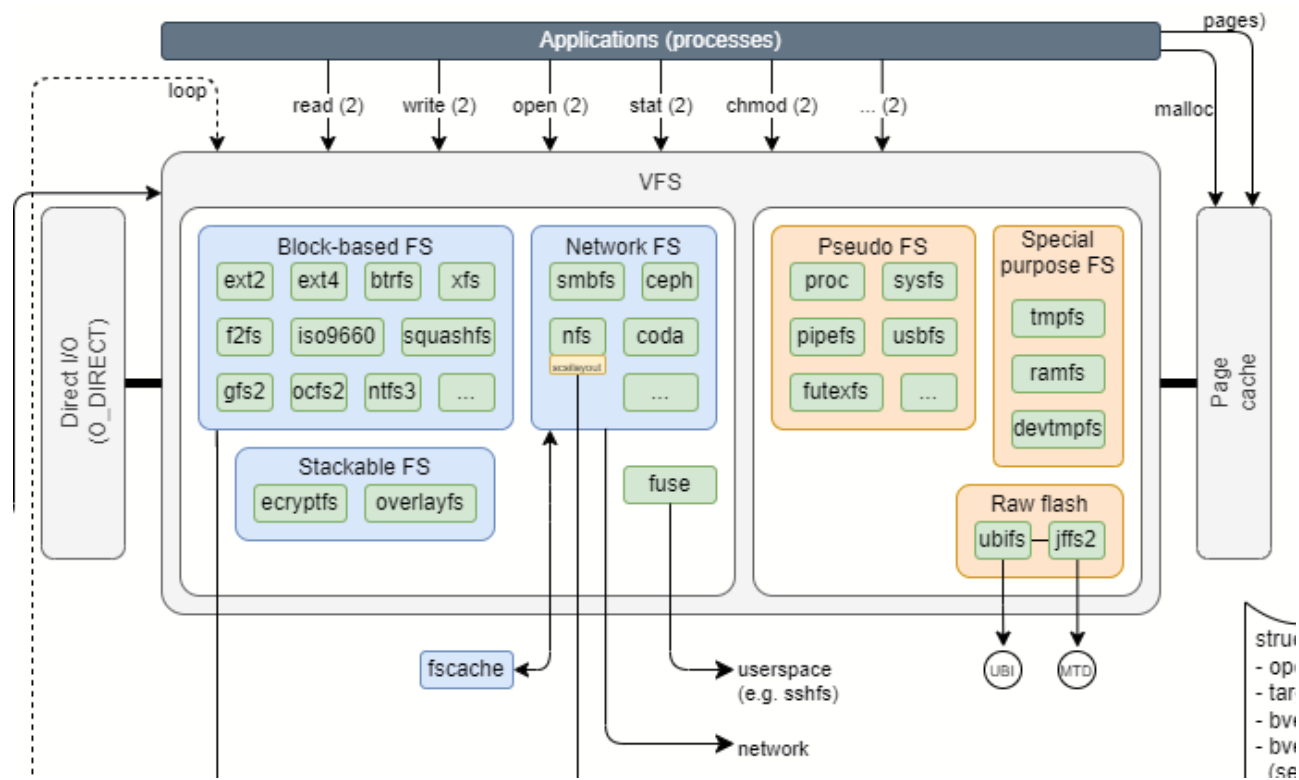
이 교육에서는 튜닝 및 파라미터에 대해서는 다루지 않는다.

# 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 UDEV를 구성하였다.

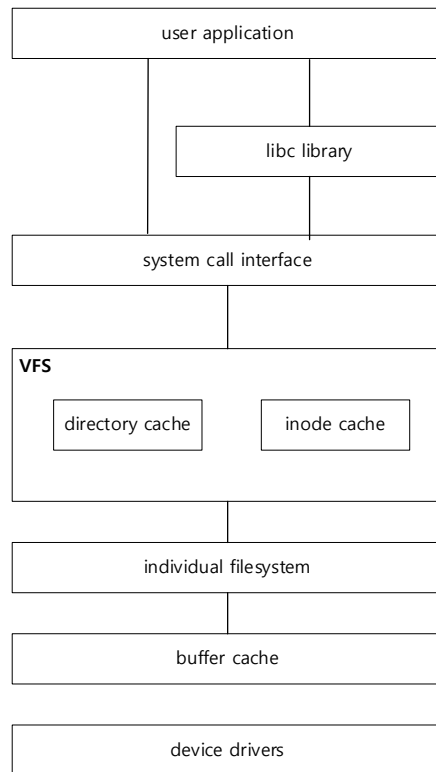
현재 UDEV는systemd 블록에 통합이 되어서 모든 대다수 리눅스 배포판은 Udev기반으로 구성한다. Udev의 최대 장점은 사용자가 규칙(rule)기반으로 손쉽게 작성 및 구성이 가능하다. 이 기능은 커널 2.5에 도입이 되었으며, 완성형은 2.6버전이 넘어가면서 완전히 커널 및 시스템과 통합이 이루어 졌다.

# 리눅스 블록 구조

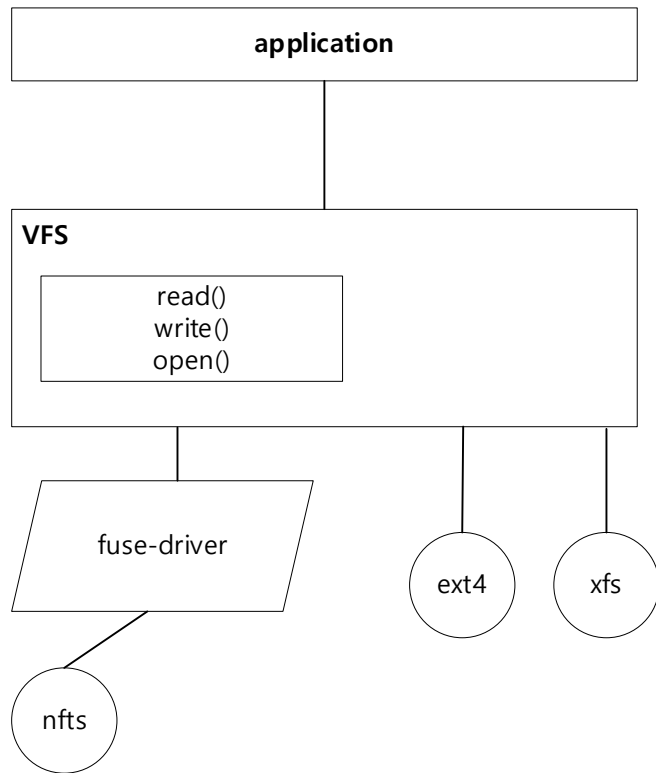




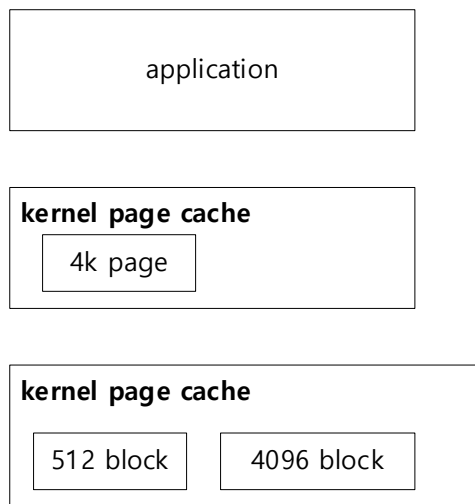
# VFS



# VFS



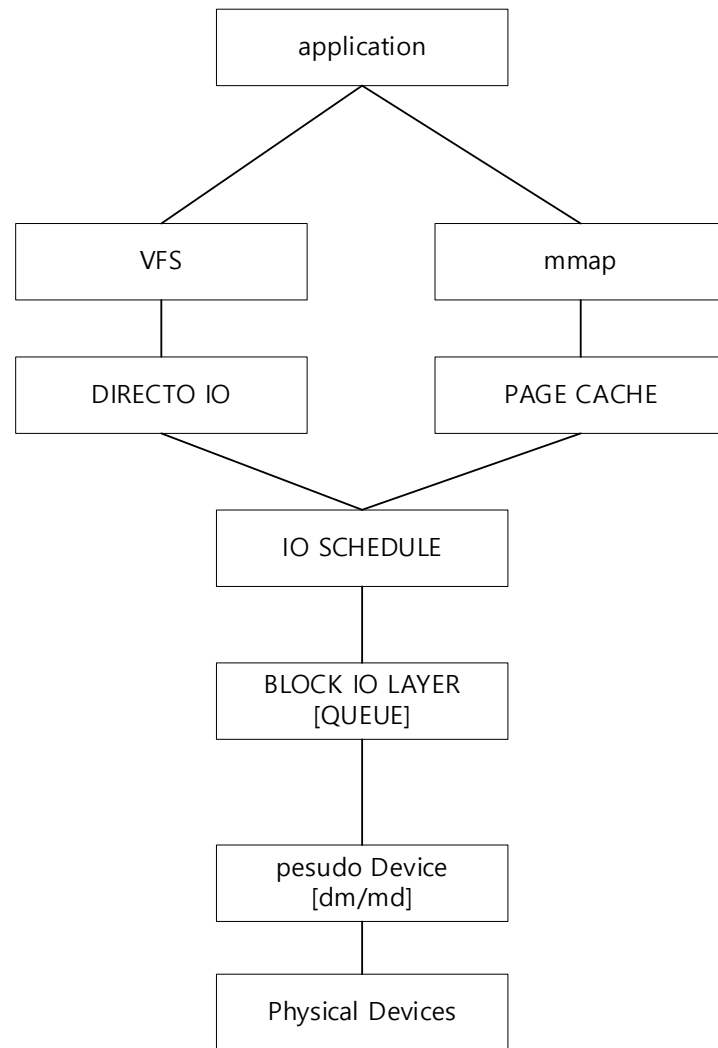
# VFS



# devicemapper

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 이 라고 부르기도 하며 , **LVM2/VDO/Stratis**에서 블록장치 백-엔드 관리자로 사용한다.

DM는 소프트웨어 계층과 연결이 가능하기 때문에, 디스크 암호가 필요한 경우, DM를 통해서 디스크 전체나 혹은 파티션을 암호화하여 사용이 가능하다. 현재 DM에서 제공하는 기능은 다음과 같다.



# devicemapper function

**cache:** cache를 통해서 하이브리드 디스크를 생성한다. 예를 들어서 SSD, HDD디스크를 동시에 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**: luks나 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/Podman(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.



# 디스크 파티션

디스크에 파티션은 파일 시스템이 생성되기 전, 논리적으로 블록의 범주를 정한다. 이를 통해서 디스크 공간을 효율적으로 사용할 수 있도록 한다. 예를 들어서 10메가의 파티션을 하나 생성한다.

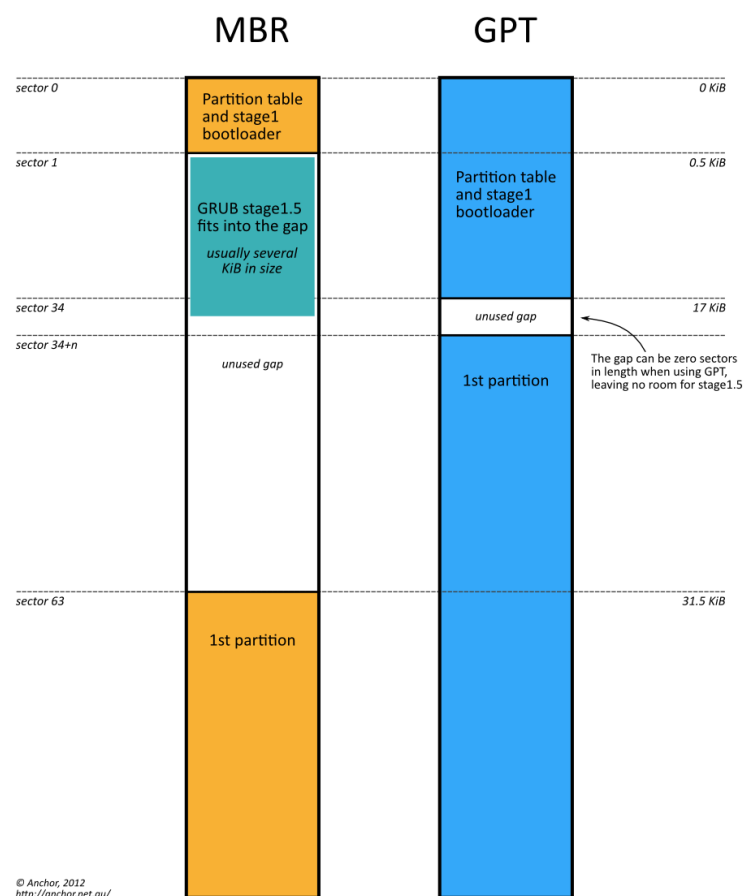
Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/sdb1		2048	12287	10240	5M	83	Linux

# fdisk/gdisk

'fdisk', 'gdisk'는 디스크에 구역을 나누는 파티셔닝(**Partitioning**)기능을 제공한다. 'fdisk'는 MBR를 제공하며, 'gdisk'는 GPT타입을 제공한다.

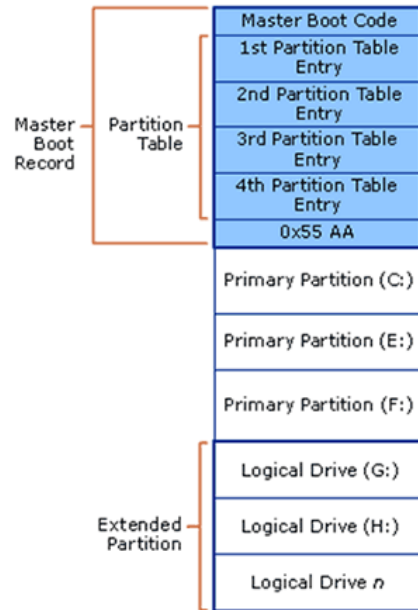
그래서 사용 시 MBR, GPT타입을 구별하여 사용해야 한다. 그렇지 않는 경우 디스크 영역 정보가 손상이 되어서 올바르게 부팅이나 혹은 접근을 할 수 없다.

# MBR/GPT

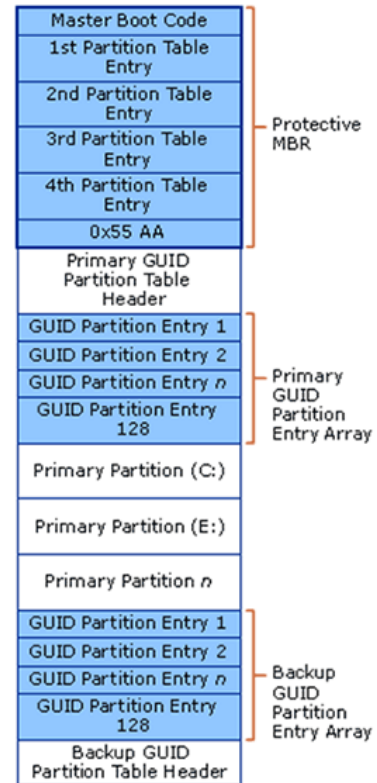


# MBR/GPT

MBR Partition Table Scheme



GPT Partition Table Scheme



# fdisk/gdisk

## MBR: Master Boot Recode

최대 4개의 Primary 파티션 논리적 파티션으로 최대 32개 파티션.

## GPT: GUID Partition Table

GUID는 globally unique identifier 약자이며, 최대 128개의 파티션을 제공함. 위의 두 개 파티션 정보 영역은 벤더사에서 제공하는 기술로 조금씩 다를 수 있다.

# fdisk/gdisk

일반적으로 많이 사용하는 파티션 에디터.

**# fdisk <DISK\_DEV>**

**# gdisk <DISK\_DEV>**

# cdisk/sfdisk

만약, 일반적인 에디터가 불편한 경우 cfdisk를 통해서 디스크 편집이 가능함.

**# cfdisk**

sfdisk는 파티션 정보를 덤프 뜨거나 확인 시 많이 사용한다. 일반적으로 디스크 정보를 수정하기 전에 백업하는 용도로 많이 사용한다.

**# echo "label: gpt" | sudo sfdisk /dev/sdc**

**# sudo sfdisk /dev/sda < sda-tables.txt**

# partprobe

**partprobe**는 갱신 디스크 정보를 커널에 반영한다. **partprobe**가 적용이 가능한 부분은 디스크 및 파티션 영역이다.

다만, 사용중인 디스크나 파티션 경우에는 **partprobe**가 올바르게 작업을 못하는 경우가 있어서 재시작이 필요하다. 실제 운영중인 서버에서 적용하는 경우 명령어가 실행이 되면서 잠깐 멈추는 상황이 발생하니 가급적이면 유지보수시 해당 명령어 사용을 권장 한다.

- **s**: 장치 및 파티션 정보 출력
- **d**: 실제로 반영하지 않고 적용가능한지 확인



# partprobe

일반적으로 많이 사용하는 디스크 파티션 정보 업데이트 도구. 특정 디스크나 혹은 파티션을 커널에 갱신 시 사용한다.

**# partprobe -d**

**# partprobe /dev/sdb**

# partx

'**partx**'는 '**partprobe**'와 비슷하지만, 조금 더 다양한디스크 및 파티션 인식 및 인식해제 기능을 제공한다. 다만, 올바른 사용법 및 옵션을 잘 모르는 경우에는 'partx'보다는 'partprobe'사용을 권장한다.

단, 앞에서 이야기 하였지만, '**partprobe**'는 종종 사용중인 시스템에서 커널 정보를 업데이트를 못하거나 혹은 시스템콜이 중지가 되는 경우가 있다(아주 잠깐).

그래서 재부팅이 없이 실시간으로 반영이 필요한 경우는 'partx'로 반영하는 것을 권장한다.

# partx

```
# partx --show - /dev/sda3
```

```
# partx -d /dev/sdb3
```

```
# partx -a /dev/sdb -n <PARTITION_NUMBER>
```

```
# partprobe /dev/sdb3
```

# kpartx

'kpartx'는 **DM(Device Mapper)**기반으로 할당 받은 장치를 커널에서 인식 시 사용한다.

이 명령어는 'partx'와 거의 동일하다. 다만, 기존의 'partx'의 기능을 사용하며, 추가적으로 "Device Mapper"와 연동이 되기 때문에 멀티 패스 설정 시, 인식이 잘 안되면 'kpartx'로 디스크 및 파티션 정보 갱신을 시도한다.

# 디스크 및 파티션 도구 정리

## partprobe

일반적인 디스크 및 파티션 갱신 도구. 하지만, 온라인중에 인식이 안되거나 사용을 권장하지 않음.

## partx

온라인 중 디스크 및 파티션을 제거 및 추가가 가능. 하지만, 디스크 및 파티션에 대한 정보가 확실하지 않으면 사용을 권장 하지 않음.

## kpartx

'partx'기능에서 "Device Mapper" 및 "Multi Path"에서 사용이 가능한 도구. 기본적으로 'partx'의 기능에서 확장이 되어 있다.

# 스왑

SWAP within Partition and File

# 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.

# 다중 스왑 및 파일 스왑

만약, "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
```



# ZSWAP

만약, 메모리 형태의 스왑을 사용하고 싶은 경우, RHEL 및 CentOS 8에서는 **zswap** 다음과 같은 명령어로 사용이 가능하다.

```
# modprobe lz4hc lz4hc_compress
```

```
# vi /etc/dracut.conf.d/lz4hc.conf
```

```
add_drivers+="lz4hc lz4hc_compress"
```

# ZSWAP

```
# dracut --regenerate-all --force
```

```
# echo lz4hc > /sys/module/zswap/parameters/compressor
```

```
# echo 25 > /sys/module/zswap/parameters/max_pool_percent
```

```
# echo 1 > /sys/module/zswap/parameters/enabled
```

```
# vi /etc/default/grub
```

```
GRUB_CMDLINE_LINUX="... zswap.enabled=1 zswap.max_pool_percent=25  
zswap.compressor=lz4hc"
```

```
# grub-mkconfig -o /etc/grub2.cfg
```

# 다중 스왑 및 파일 스왑

```
# dd if=/dev/zero of=/root/temp_swap.dat bs=1G count=1
```

```
# mkswap /root/temp_swap.data /dev/sdd2
```

```
# swapon /root/temp_swap.dat /dev/sdd2
```

```
# swapon -S
```

```
# /root/temp_swap.dat
```

```
# /dev/sdd1
```

```
# nano /etc/fstab
```

```
/dev/sdd2 swap swap defaults,pri=10 0 0
```

```
/root/temp_swap.dat none swap defaults,pri=20 0 0
```

# 볼륨 관리하기

LVM2

VDO

Stratis

# 볼륨

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

# VDO

COMPRESSED STORAGE

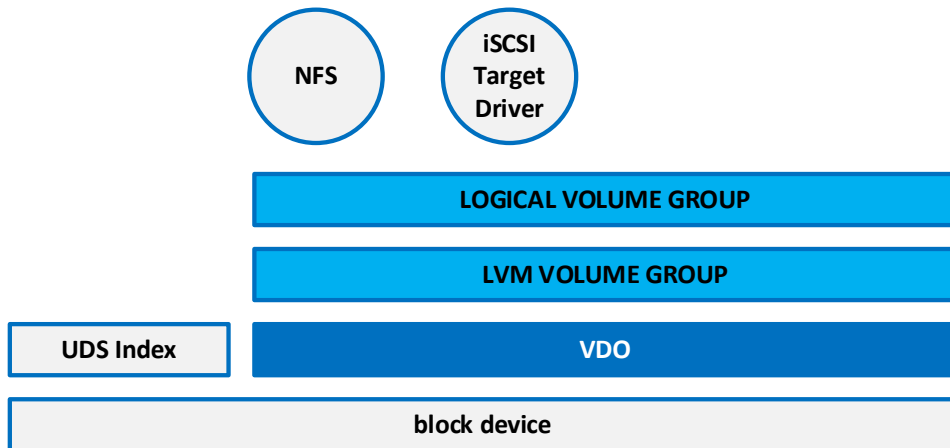
# VDO

**VDO(Virtual Data Optimizer)**는 이전에 독립적인 서비스로 사용 하였으나, 지금은 **LVM2**기반으로 **VDO**스토리지 구성 및 관리가 된다. 현재 **vdo**는 **vdo.service**사용이 불필요 하다.

# VDO

**VDO**서비스는 가상 머신 기반에서 사용을 많이 한다. 가상머신은 다량의 디스크 이미지 파일을 생성하며, 각각 이미지 디스크는 동일한 블록 영역을 사용한다. 이를 **vdo.ko**모듈이 블록 장치에서 확인하여 중복된 부분을 압축한다.

현재는 **LVM2**로 통합이 되었기 때문에, 레드햇 계열 7/8버전과 사용방법이 다르다.





# VDO

	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. **SecureBoot** 끄고 수행하면 잘 됨. 최소 크기가 3기가 이상이면 동작.

```
# dnf install vdo kmod-kvdo -y
```

```
# ln -snf ../extra/kmod-kvdo/vdo/kvdo.ko dm-vdo.ko
```

```
# depmod -a
```

```
# pvcreate /dev/sdb
```

```
# vgcreate vg-vdo /dev/sdb
```

```
# lvcreate --type vdo --name lv-vdo --size <VG_SIZE> vg-vdo
```

```
# dnf install kernel -y
```

```
# find / -name kvdo.ko -type f -print
```

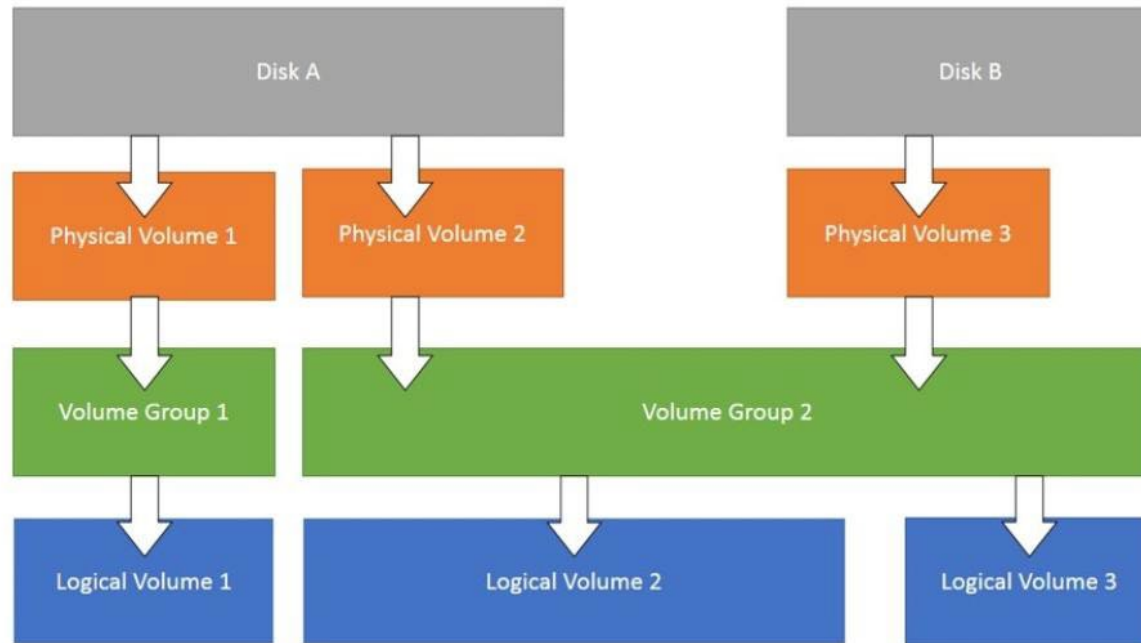
```
# uname -r
```

```
# lvcreate --type vdo --name lv-vdo --size <VG_SIZE> vg-vdo
```

# LVM2

LOGICAL VOLUME

# LVM2



# LVM2 볼륨 및 논리 장치 생성

기본적인 LVM 그룹 및 논리적 장치 생성. 아래 명령어로 생성이 가능하다.

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.

500메가를 추가한다.

```
# fdisk /dev/sdb
```

추가로 만든 파티션 혹은 디스크를 test-vg에 추가한다.

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

확장된 볼륨 크기만큼 논리 디스크를 확장한다.

```
# lvextend -r /dev/test-vg/test-lv
```

**-r:** resize to FS는 자동으로 수행한다. 예를 들어서 ext4는 'resize2fs', xfs는 'xfs\_growfs'명령어를 사용한다.

# STRATIS

XFS POOL

# STRATIS

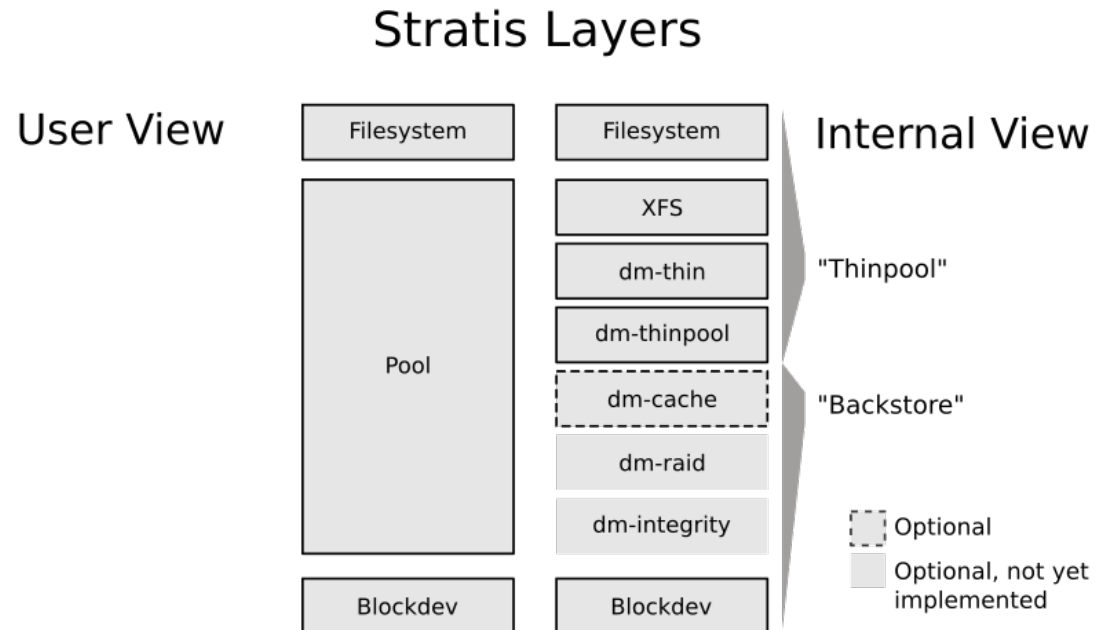
레드햇에서 사용하는 **XFS는 POOL기능이 없다**. 이러한 부분은 **LVM2**로 해결을 하였지만, 운영이 복잡하고 pooling기능 보다는 기존 레이드 기술과 가까운 부분이 있다.

그래서, 좀 더 쉽게 사용하기 위해서 **Stratis**라는 파일 시스템 도구를 만들었다. 이는 초창기 VDO마찬가지로, 독립적인 서비스로 구성되었다가 지금은, LVM2와 통합이 되었다.



# STRATIS

**Stratis**는 기존에 사용하던 LVM2를 기반으로 백-엔드 구성이 되어있다. LVM2에서 구성하던 **DeviceMapper(DM)**를 Pool기능으로 구현하였다.



# STRATIS

```
# dnf install stratis-cli stratisd
```

```
# systemctl enable --now stratisd
```

```
# stratis pool create test-pool /dev/sdd
```

```
# stratis pool list
```

```
# stratis pool add-data test-pool /dev/sde
```

```
# stratis blockdev list test-pool
```

# STRATIS

**# stratis filesystem create test-pool testfs**

**# stratis filesystem list**

**# stratis filesystem snapshot test-pool testfs snapshot1**

**# lsblk --output=UUID /dev/stratis/test-pool/testfs**

**# nano /etc/fstab**

**UUID=<UUID> /mnt/stratis xfs defaults,x-systemd.requires=stratisd.service 0 0**

# NetworkManager

nmcli

# NetworkManager

네트워크 매니저는 **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. NetoworkManager는 `"systemd-networkd"`로 통합이 되었음.

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

# NETWORK MANAGER

기존에 사용하던 **network configuration**은 **init**기반의 "**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.

이는 모든 레드햇 리눅스 기반, 혹은 LSB를 따르는 배포판은 네트워크 설정 기능을 "**Network Manager**"기반으로 관리를 하게 된다.

## 관리 명령어

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

# NETWORK MANAGER COMMAND

## **nmtui**

TUI기반으로 네트워크 설정한다. 자동화 용도로 사용하기가 어렵다.

## **nm-connection-editor**

엑스 윈도우 기반으로 네트워크 설정 변경. 일반 사용자들이 많이 사용한다.

# NETWORK MANAGER COMMAND

## nmcli

CLI기반으로 네트워크 인터페이스 변경. 쉽지는 않지만, 자동화나 혹은 반복적으로 수정 시 도움이 된다. 아래 내용은 RHEL 9기반에서 사용하는 **NetworkManager**설정 내용이다. 더 이상 "ifcfg-rh"를 사용하지 않는다.

```
[main]
#plugins=keyfile,ifcfg-rh
```

## /etc/sysconfig/network-scripts/

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



# 네트워크 관리 명령어

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

1. `ip <- ifconfig(namespace 지원안됨)`
2. `ss <- netstat(namespace 지원안됨, 느림)`
3. `ip r <- route(namespace 지원안됨)`
4. `nftables <- iptables(강화된 버전)`

명령어 및 기능이 변경이 된 제일 큰 이유는 "다중 자원 관리(컨테이너, 가상화)", "namespace 조회 및 관리"

# 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 네트워크 설정은 NetworkManager에서 구성해야 됨.

**# ip addresss add <DEV>**

# 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

# 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

## # traceroute

원하는 경우, 여전히 구형 명령어는 설치 후 사용이 가능하다.

# 네트워크 관리 명령어

```
# ip a add 192.168.1.200/255.255.255.0 dev eth1
```

```
# ip a s eth1
```

```
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default  
qlen 1000
```

```
    link/ether 56:6f:08:c8:03:cb brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.1.200/24 scope global eth1
```

```
        valid_lft forever preferred_lft forever
```

```
# nmcli connection show eth1
```

```
# nmcli device
```

# 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](#)



# NetworkManager

'nmcli'는 스크립트 기반이나 혹은 여러 노드 구성 시 사용한다. 'nmtui'는 TUI기반으로 네트워크 설정 및 구성할 수 있다.

```
# nmcli connection add type dummy ifname dummy0 ipv4.method manual ipv4.addresses  
192.0.2.1/24 ipv6.method manual ipv6.addresses 2001:db8:2::1/64
```

```
# nmcli con sh
```

```
# nmtui
```

# 더미 장치 생성하는 방법

ip command 사용 방법은 다음과 같다. 아래 명령어는 예제이다.

```
# 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

```
# ip netns add test
```

```
# ip netns list
```

```
test
```

```
# ls -al /var/run/netns/
```

```
total 0
```

```
drwxr-xr-x. 2 root root 60 Sep 11 13:46 .
```

```
drwxr-xr-x. 24 root root 760 Sep 11 13:46 ..
```

```
-r--r--r--. 1 root root 0 Sep 11 13:46 test
```

# 네임 스페이스 장치

간단하게 ip명령어 기반으로 network namespace장치 생성 및 확인을 해본다.

```
# 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
```

# 네임 스페이스 장치

```
# ip -n switch1 netns set switch2 20
```

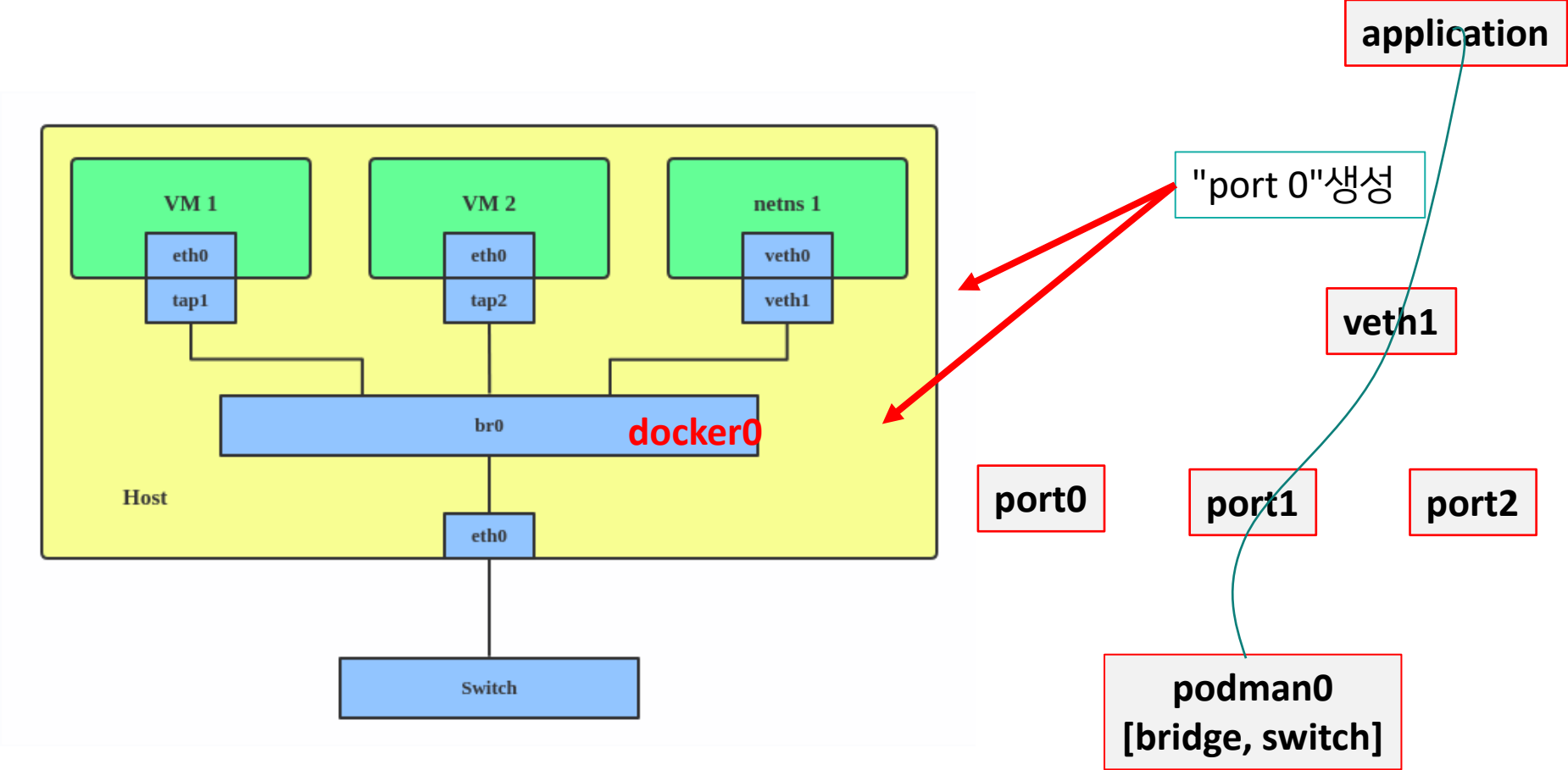
```
# ip -n switch2 netns set switch1 30
```

```
# ip -n switch2 netns set switch1 40
```

```
# ip netns list-id target-nsid 12
```

```
# ip netns list-id target-nsid 12 nsid 13
```

# 컨테이너 브리지 구조 및 구현



# 아이피 브릿지 구성

```
# 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
```

# 방화벽 및 네트워크 설정

nftables

firewalld

NetworkManager



# nftables

기존에 사용하던 **iptables**는 대용량의 컨테이너 및 가상머신 운영에는 적절하지 않았다. 그래서 기존에 사용하던 **{ip,ip6,arp,eb}tables** 명령어를 새로운 **"in-kernel packet farmwork"**로 변경하였다.

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

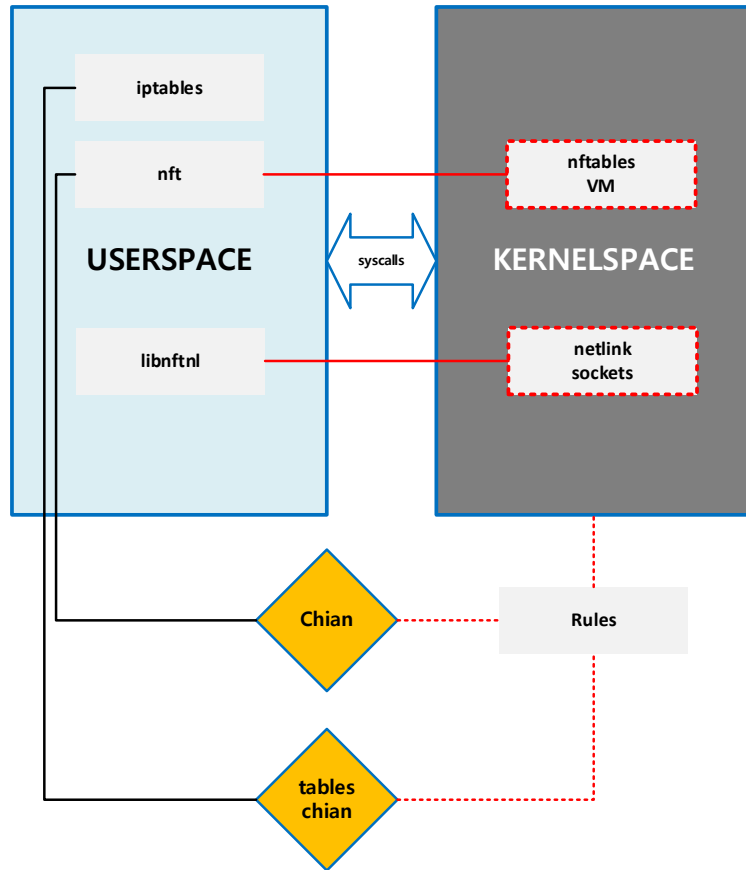
이전 iptables와 다른 부분은 JSON기반으로 정책파일을 관리한다. 그래서 이전 테이블보다 빠르게 입출력 및 검색이 가능하다.

# nftables

**nftables**는 **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.

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

# nftables



# nftables

여전히 호환성 모드로 iptables 사용은 가능하나 가급적이면 nftables 기반으로 작업 권장.

```
# yum install nftables
```

```
# systemctl enable --now nftables.service
```

# 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

rules 일반적으로 적용하기 위해서는 JSON형태로 파일을 작성 후, nft명령어로 밀어 넣으면 된다. nft는 "#"를 지원하지기 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

## nft

add	rule	ip	<NAME>
delete	table	ip6	
flush		inet	
insert		arp	
table		bridge	
chain		prerouting	
rule		input	
		forward	
		output	
		postrouting	

# nftables

특정 아이피 드롭, 하지만 카운팅 모듈 사용

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



대소문자 구별 합니다!

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

# nftables

명령어 사용 예제. 아래 명령어가 제일 많이 사용하는 명령어 중 하나이다.

icmp입력을 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
```

filter리스트의 테이블을 확인 합니다.

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

# nftables

특정 rule를 제거합니다. 이때 번호를 꼭 넣어 주어야 합니다.

```
# 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
```

- handle <NUMBER>                      ## 정책(rule)의 번호
- nft add rule filter/ip <TYPE>
- rule를 "filter"에 추가한다.

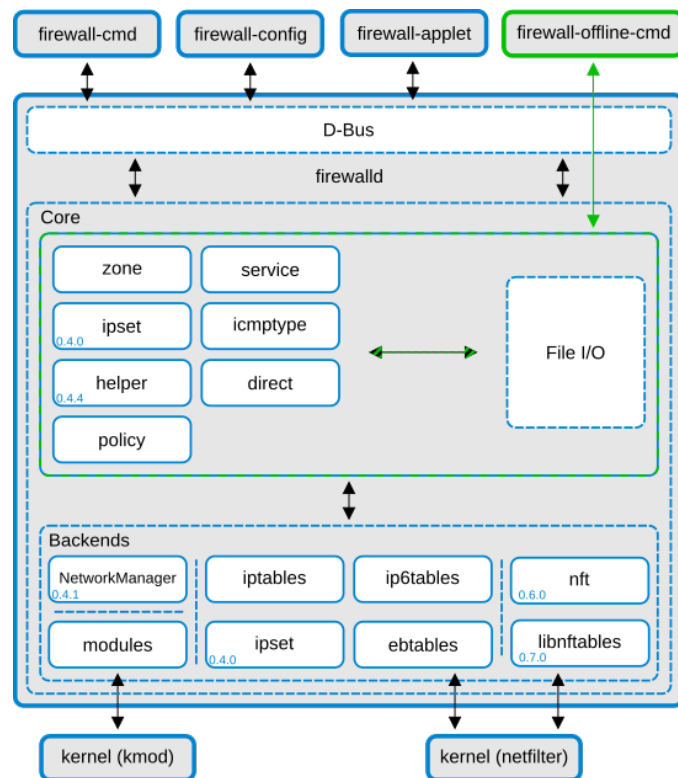
# firewalld

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

**firewall-cmd**명령어 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 8이후부터 nft/firewalld만 사용이 가능하다.

# firewalld





# firewalld

```
# systemctl start firewalld
```

```
# ls -l /lib/firewalld
```

```
# cd /lib/firewalld/services
```

```
...
```

```
http.xml
```

```
...
```

# firewalld

"zone"목록을 확인하는 방법

```
# firewall-cmd --get-zones
```

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

"특정 존"에다가 아이피 등록하는 방법

```
# firewall-cmd --add-source=10.10.10.0/24 --zone=block --permanent
```

```
# firewall-cmd --add-service=https --zone=drop
```

```
# firewall-cmd --add-service=http
```

```
# firewall-cmd --add-port=8899/tcp
```

```
# firewall-cmd --set-default-zone=
```

```
# firewall-cmd --get-default-zone=
```

# 패키지 관리 방법

dnf/yum

rpm/modules

# 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 module**를 통해서 패키지 제공이 되며, 또한 이 기능은 이전에 사용하였던 **SCL(Software Collection Library)**와 비슷한 기능을 제공 하기도 함.

# 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, common, runc, common, 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, common, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and supported for 24 monthd]s.
-----------------	-----	---

# rollback

yum 및 dnf는 rollback기능을 제공한다.

**dnf history rollback**

**undo**

**redo**

**list**

**info**

# 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 방지 할 수 있다.

현재 대다수 RPM기반의 배포판은 YUM에서 DNF-3로 변경이 된 상태이다.



# RPM commands

RPM명령어는 기존에는 '**rpm**'이라는 하나의 명령어만 사용하였다.

RPM은 기능이 확장 되면서, 명령어 분리하기 시작. 아래처럼 분리가 되었다.

**rpm**: 기본 설치 명령어. 일반적으로 패키지 추가/삭제/제거 및 확인.

**rpm2archive**: .rpm패키지를 ".tgz"로 변경한다.

**rpm2cpio**: .rpm에서 cpio묶여 있는 파일을 푸는 명령어.

**rpmdb**: RPM의 BerkelyDB 조회 시 사용하는 명령어.

**rpmkeys**: RPM에서 사용하는 공개키 관리.

**rpmquery**: RPM에서 사용하는 시그 키 관리.

**rpmverify**: RPM에 등록된 패키지의 자원들에 대한 의존성 검사.

# RPM commands

`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 commands

```
rpm -qi <PACKAGE_NAME>
```

```
rpm -Vp <PACKAGE_NAME>
```

```
rpm -Va <PACKAGE_NAME>
```

```
rpm -qa gpg-pubkey*
```

```
rpm -q --scripts systemd
```

```
rpm -q --triggers systemd
```

# 프로세스 관리를 위한 명령어

process utils

# ps

- 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.

# ps

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

## # ps -ef

위의 명령어는 AIX(Unix)에서 사용하던 유닉스 스타일 명령어

## # ps aux

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

# ps

ps명령어는 어떤 옵션을 사용하느냐 따라서 시스템에 부하 수준이 다르다. 구동중인 시스템에서는 'top'명령어보다 'ps'명령어 사용을 더 권장하고 있다.

## # ps aux or ps au

위의 명령어는 BSD스타일 명령어

# ps

# ps aux or ps au

"a" 옵션은 BSD 스타일의 "only yourself" 옵션

"u" 옵션은 EUID(effective user ID)

"x" 옵션은 x BSD 스타일의 "must have a tty" 옵션

```
[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/systemd
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

# ps -ef

"-f" 옵션은 "full-formatting list"

"-e" 옵션은 "all processes"

위의 명령어는 **Unix(AIX)**스타일의 명령어

```
[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

# 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

# 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

# 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

9, KILL, kill a process without page down

**15, TERM**, gracefully stop a process



# pgrep/pkill

**pgrep**은 프로세스의 이름 혹은 다른 속성을 확인하여 시그널 전달

```
$ pgrep <signal> <process_name>
```

```
$ pgrep -u root sshd
```

```
$ pgrep -u tang,root,daemon
```

# pskill/pgrep

```
$ pgrep -c ssh
```

```
10
```

```
$ pgrep -d "-" ssh
```

```
1363-30317-79439-79442-79459-79462-80023-80026-80047-80050-80051-80054-80825
```

# pgrep/pkill

```
$ pgrep -u tang ssh
```

```
80825
```

```
80828
```

```
80861
```

```
80864
```

```
80865
```

```
80868
```

```
131706
```

# pgrep/pkill

```
$ 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
```

# pgrep/pkill

**\$ 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

# pgrep/pkill

**\$ pgrep -v ssh**

ssh와 일치하지 않는 프로세스만 출력.

**\$ pgrep -x sshd**

정확하게 sshd이름과 일치하는 프로세스만 출력.

**\$ pgrep -n ssh**

최근에 생성된 ssh process만 출력.

**\$ pgrep -o ssh**

이전에 생성된 ssh process만 출력.

# pskill/pgrep

```
$ pskill '^ssh$'
```

```
$ pskill -9 -f "ping 8.8.8.8"
```

```
$ pskill -u mark
```

```
$ pskill -u mark gnome -9
```

```
$ pskill -9 -n screen
```

# 사용자 관리

coreutils

autofs



# useradd, passwd

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

## useradd

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

# useradd, passwd

## adduser

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

```
[root@node1 NetworkManager]# ls -l /usr/sbin/adduser
lrwxrwxrwx. 1 root root 7 Nov  1 2022 /usr/sbin/adduser -> useradd
[root@node1 NetworkManager]# |
```

# useradd + openssl

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>
```

혹은 아래와 같이 사용자 암호생성 및 등록을 동시에 수행이 가능하다.

```
# adduser --password $(echo "helloworld" | mkpasswd --stdin) test1
```

# autofs diagram

# autofs

# autofs server

```
# dnf install nfs-utils
```

```
# systemctl enable --now nfs-server
```

```
# mkdir -p /srv/indirect/aufs-user1 aufs-user2
```

```
# mkdir -p /srv/direct/
```

```
# vi /etc/exports
```

```
/srv/indirect *(rw,sync)
```

```
/srv/direct *(rw,sync)
```

```
# exportfs -avrs
```

# autofs client

**# dnf search autofs**

**# dnf install autofs**

**# systemctl enable --now autofs.service**

# autofs client

```
# 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

# showmount -e node1.example.com

# ssh [aufs-user@node1.example.com](#)

# pwd

# 저장소 구성 및 미러링

dnf repository

# 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

```
# mkdir -p /mnt/reposdisk/rpms
```

```
# dnf reposync -p /mnt/reposdisk/rpms
```

```
# createrepos_c /mnt/reposdisk/rpms .
```

```
# dnf install apache -y
```

```
# mount -obind /mnt/reposdisk/rpms /var/www/html/
```

# 가상머신 및 컨테이너 구성 및 구현

podman

libvirtd

ansible



# 컨테이너

리눅스에서는 기본적으로 다음과 같은 컨테이너 런타임을 사용한다.

- Podman
- CRI-OI

# podman

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

**Podman**은 다음과 같은 서비스가 있다.

- podman.service
- io.podman.service

# podman

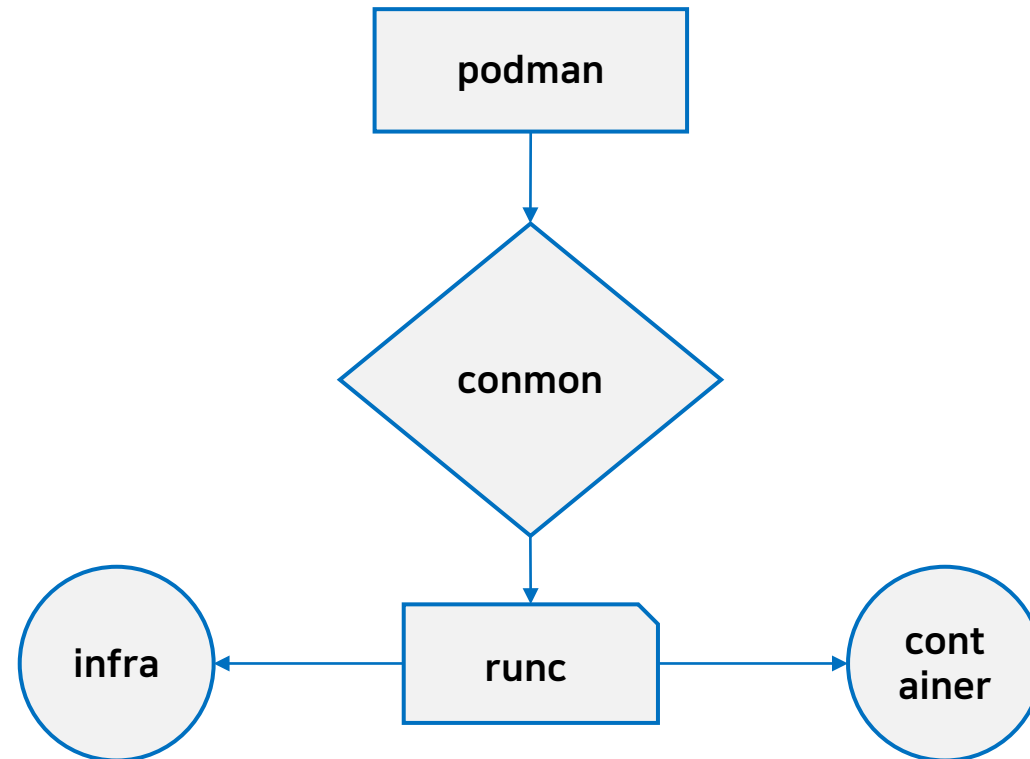
배포판 별로 다르지만 보통은 "podman.service"이름으로 운영이 된다. 레드햇 계열은 설치 시 다음과 같은 명령어로 설치한다.

```
# dnf install podman
```

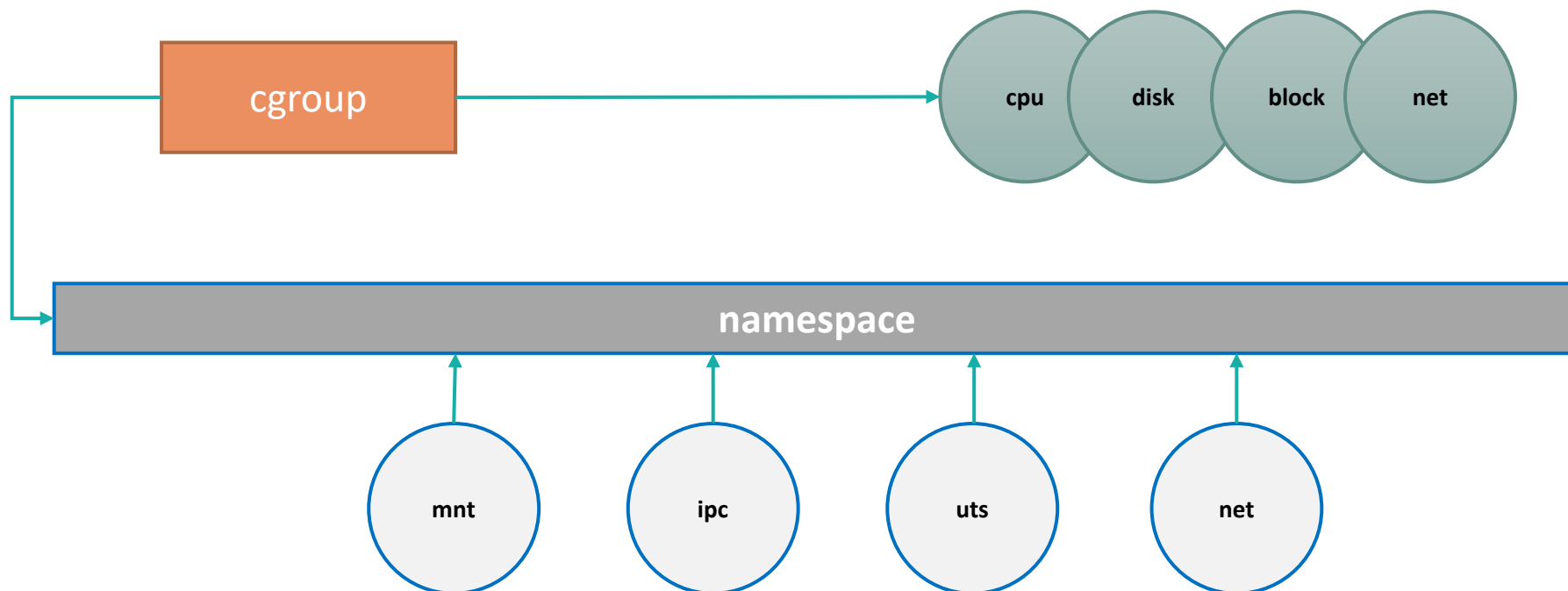
```
# dnf module list containers
```



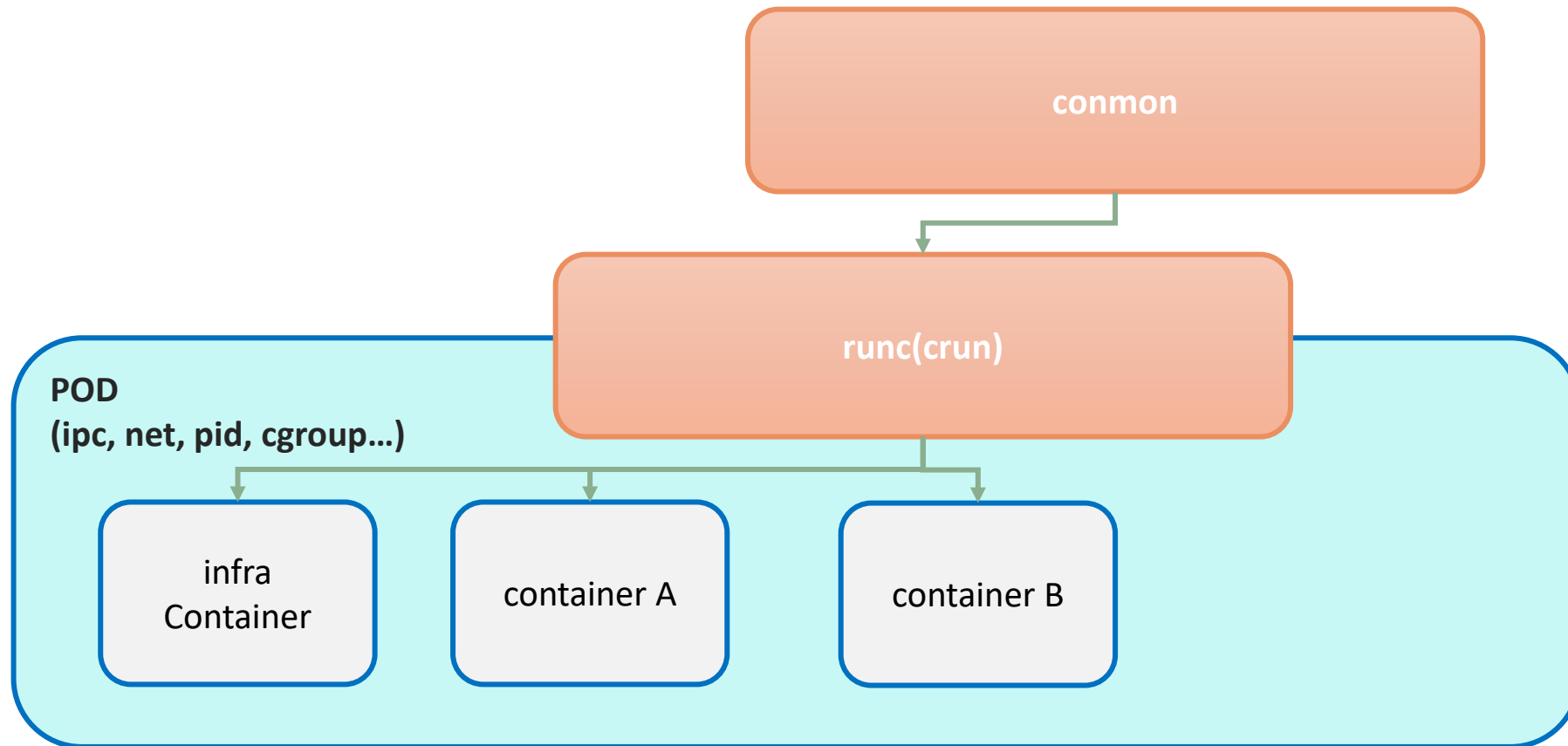
# podman



# podman



# 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.

```
# podman container create --name httpd --pod new:pod_http httpd
```

```
# podman container ls
```

# container runtime storage

## **/var/lib/containers/**

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

## **/run/containers/**

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



# container runtime storage

## **/etc/containers/**

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

## **/etc/cni/**

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.

# 컨테이너 기반 서비스 만들기

```
# 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
```

```
# podman ps
```

```
# curl http://localhost:8080
```

# 자동화를 위한 앤서블 구성

앤서블 맛보기 :)



# 앤서블 기본

introduce of ansible

# 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/vim**에디터가 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 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

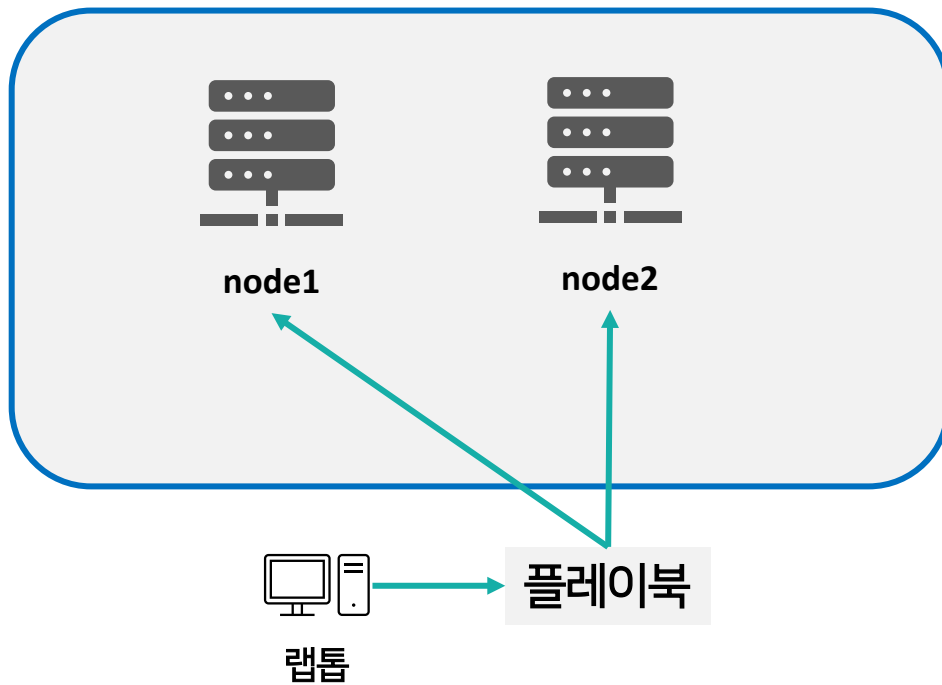
앤서블은 기본적으로 두 가지 접근 방식을 제공한다.

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

가상머신 생성이 불가능하면, 그냥 랩톱만 있어도 상관 없다.



# SSH

```
$ ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
```

```
$ ssh-copy-id <ID>@<HOST>
```

# SSH

---

- hosts: all

tasks:

- name: send a ssh public key to each hosts from inventory

**authorized\_key:**

user: "{{ ansible\_user }}"

key: "{{ lookup('file', '/root/' + '.ssh/id\_rsa.pub') }}"

SSH: root

ssh public key 위치



# 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:  
  <keyword>  
  <module>
```

```
- name:  
  <keyword>  
  <module>
```

# YAML

그래서 YAML상단에는 다음과 같은 형태로 키워드 명령어를 사용한다.

```
- 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.cfg의 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 문법

sample-yaml.yaml 파일 참고

# 앤서블 모듈 및 패키지

제일 많이 사용하는 모듈 **copy**기반으로 기능을 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 셸 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 셸 스크립트에서 같이 사용하거나 혹은 몇몇 셸 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



# 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**라는 이름으로 구성함. 다른 이름으로 변경을 원하는 경우 **ansible.cfg**에서 변경이 가능함.

혹은 "-i"으로 임의적으로 선택 가능.

# 앤서블 인벤토리

```
# mkdir ansible
```

```
# cd ansible
```

```
# nano hosts
```

```
[web]
```

```
localhost
```

```
[db]
```

```
localhost
```

# 앤서블 인벤토리

인벤토리 파일 기반으로 실행 시 다음과 같이 실행한다.

```
$ ansible-playbook <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

```
# vim ping.yaml
```

```
---
```

```
- hosts: all
```

```
  tasks :
```

```
    - name: ping to all hosts
```

```
      ping:
```

# 앤서블 플레이북

**# vim copy.yaml**

- hosts: all

tasks :

- name: copy to /etc/hostname 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**



# ansible-doc -l

**# ansible-doc -l**

사용이 가능한 모듈 리스트 확인

**# ansible-doc copy**

모듈 사용 방법 및 속성 확인

# 앤서블 플레이북

```
# vi package.yaml
```

```
- hosts: all
```

```
tasks :
```

```
- name: install the postfix package
```

```
yum:
```

```
name:
```

```
- postfix
```

```
- httpd
```

```
- mariadb-server
```

```
state: present    ## 최신 패키지가 아닌, 사용이 가능한 패키지(업데이트가 최대한 안되도록)
```

```
# 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](http://www.example.com)

**# ansible-playbook hostname.yaml**

**# hostnamectl**