

커뮤니티 OS/AUTOMATE

최국현

tang@linux.com

10/6/2022

DAY 1

오픈소스

리눅스와 오픈소스 관계

오픈소스

오픈소스는 **리차드 스톨만**이 GNU도구 및 라이선스를 선언하였음.

리눅스 토발즈는 GNU도구 중 **리눅스 커널(Linux Kernel)**를 만들었다.

리차드 스톨만은 **Hurd**라는 커널을 만들었지만, 선진적인 디자인의 **마이크로 커널 디자인(Micro Kernel)**를 사용하여서, 당시 성능으로는 구현 및 사용하기가 어려운 부분이 많았음.

리눅스

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 Mac OS의 커널로 사용이 될 뻔 하였으나, 애플에서 비공개라는 조건으로 토발즈에게 요구(?) 하였고, 이에 대해서 리누즈 토발즈는 거절 하였다.

오픈소스 라이선스

현재 리눅스는 다음과 아래와 같이 소프트웨어 구성이 되어 있다.

BSD Software

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

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.

리눅스 쉘 및 커널

user-space, kernel-space

Shell POSIX

POSIX에서 명시한 echo명령어는 불행하게도 UNIX System V, BSD 그리고 GNU기반에 따라서 각기 다르게 동작한다.

예를 들어서 GNU 'echo'명령어는 표준은 UNIX System V 혹은 BSD와 다르게 동작한다. (완전 다르지 않는다. 조금씩 결과가 다르게 나온다)

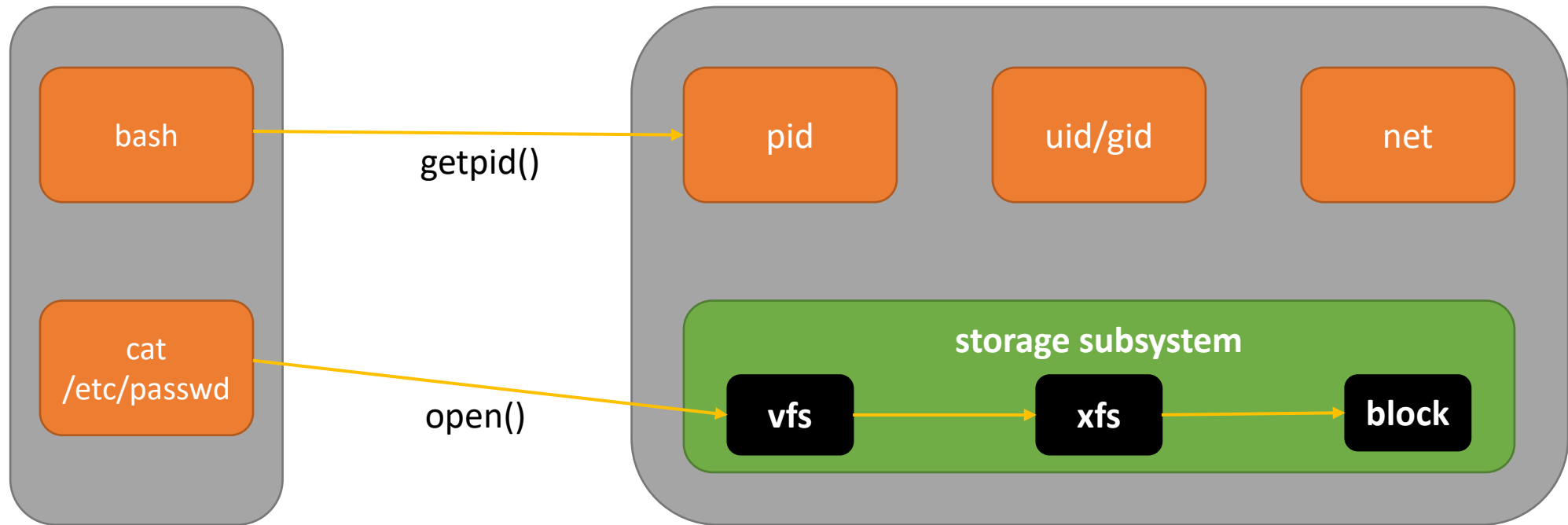
그러한 이유로, 'echo'명령어 대신 호환성을 높이기 위해서 'printf'명령어 사용하는 것을 권장한다.

BASH 무엇인가?

셸은 사용자가 커널 및 사용자 프로그램 사이에서 서로 대화 할 수 있도록 해주는 도구.

셸은 ssh같은 도구가 아님. 컴퓨터와 대화하기 위한 도구. 좀 더 정확하게는 시스템 호출.

BASH 무엇인가?



BASH 무엇인가?

셸은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 **TTY = SHELL**라는 오해를 가지고 있음.

PuTTY/iTerm/Xterm와 같은 터미널 도구가 아님.

Bash는 명령어 라인(CLI)도구가 아니다. Bash는 대화형 모드를 가지고 있으며 이를 통해서 프로그램들을 사용함.

예를 들어서 ls같은 명령어.

BASH 무엇인가?

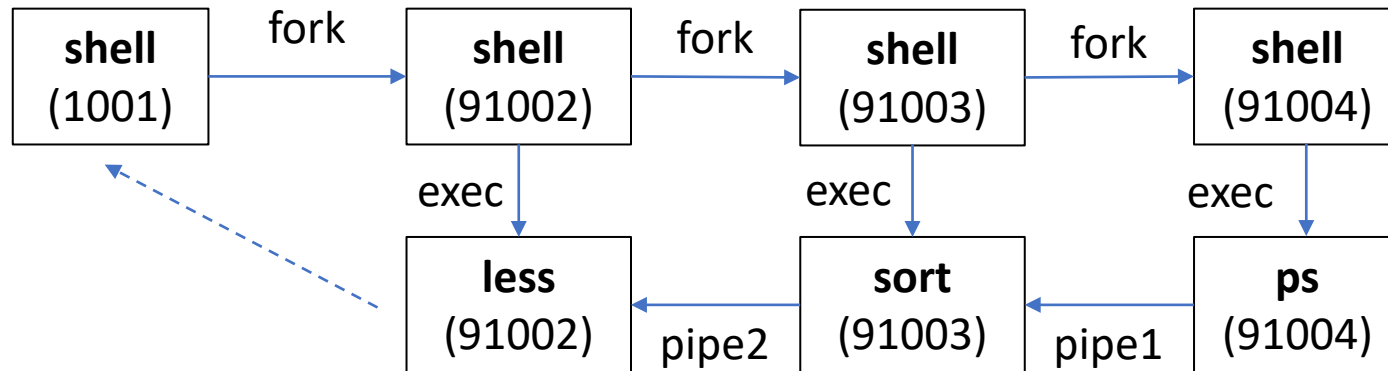
거의 대다수의 리눅스는 bash가 설치되어 있음.

다른 말로 **GNU/Linux**는 **bash가 표준**이라고 생각해도 됨.

예를 들어서 **FreeBSD/NetBSD/OpenBSD** 그 외 다른 유닉스 들에서는 bash가 아닌 csh, tsh를 사용하기도 한다. 하지만 bash사용이 가능하며, 기존에 사용하였던 csh,tsh에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

BASH가 할 수 있는 것.

\$ ps | sort | less



프로토타입

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.

필터링 그리고 입력 변환

어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다.

이러는 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다.

예를 들어서 'awk', 'sed'같은 프로그램들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.

베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다.

이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장 가능하다.

우리는 이 기능을 **와일드카드(wildcard)**라고 부른다.

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 셸 프로그램이 이식 되어 있으면 손쉽게 사용이 가능하다.

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

리눅스 커널

리눅스 커널의 정확한 명칭은 GNU/Linux이다.
실제로 리눅스의 커널은 두 가지가 있다.

GNU/Linux

현재 제일 많이 사용하는 **모놀릭 타입**의 커널.

윈도우 서버와 마찬가지로 완전한 모놀릭 타입이 아니라 현재는
마이크로 커널의 타입도 어느정도 도입이 되어서 유연하게 모듈 운영이
가능하다.

보통 **하이브리드 커널** 이라고 부르기도 한다.

리눅스 커널

리눅스 커널의 정확한 명칭은 GNU/Linux이다. 실제로 리눅스의 커널은 두 가지가 있다.

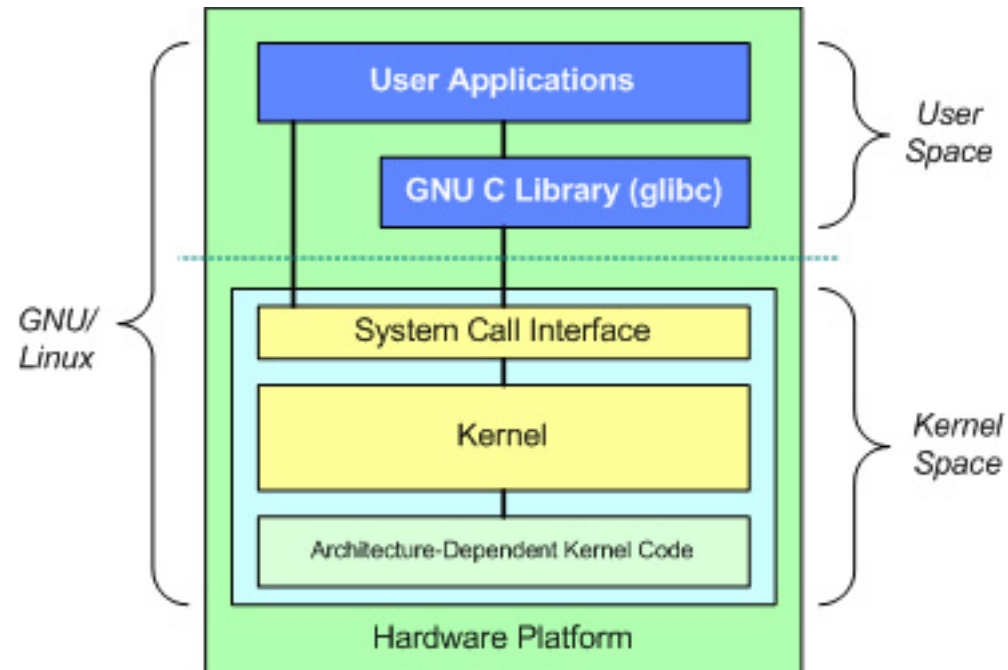
GNU/Hurd

허드 커널은 **리처드 스톨만**이 만들었으며, 여전히 개발이 되고 있다.

다만, 너무 이르게 디자인 적용 및 릴리즈가 되어서 성능 문제 및 아키텍트 적용에 많은 문제가 있었다.

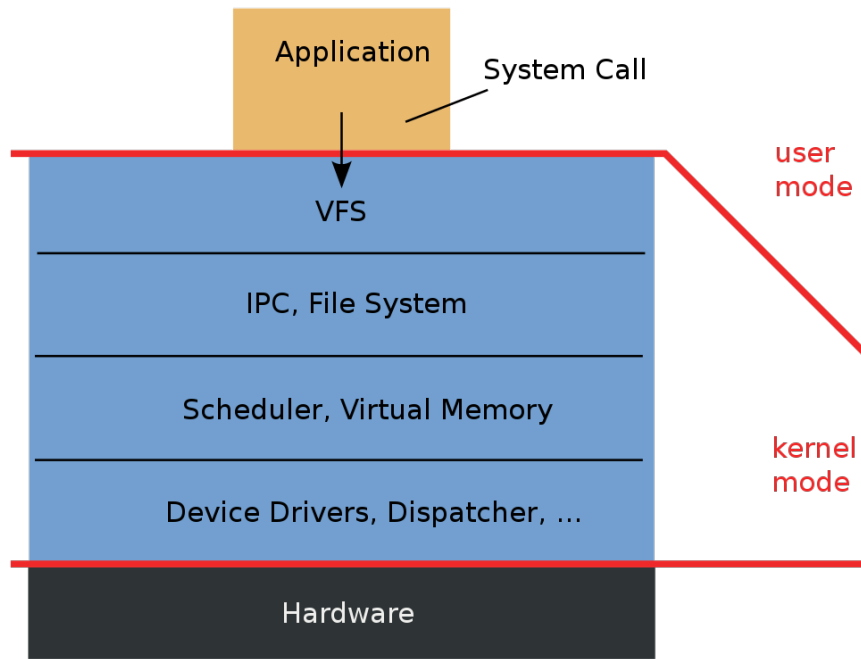
현재 허드 커널은 데비안 리눅스에서 사용이 가능하나, 아직 많은 장치를 지원하지 않기 때문에 사용율이 낮다.

리눅스 커널

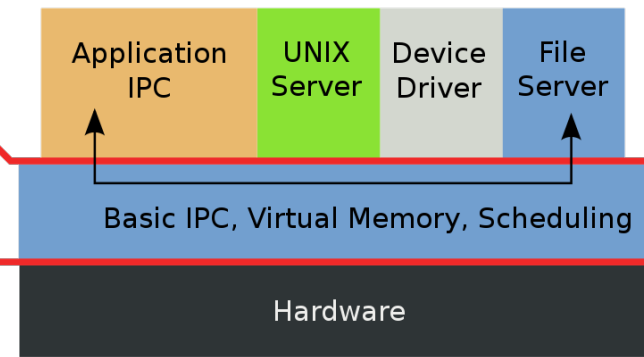


리눅스 커널

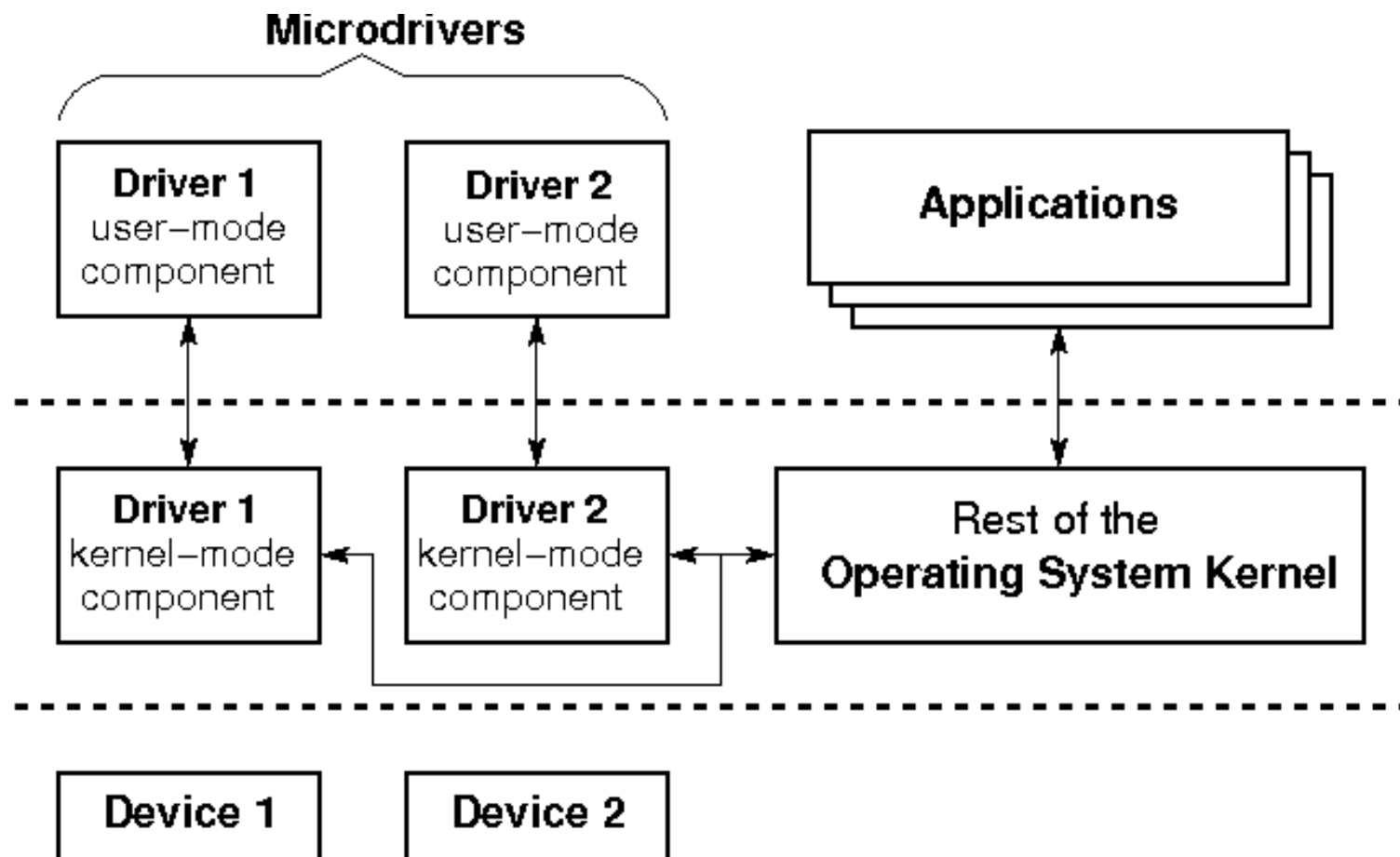
Monolithic Kernel
based Operating System



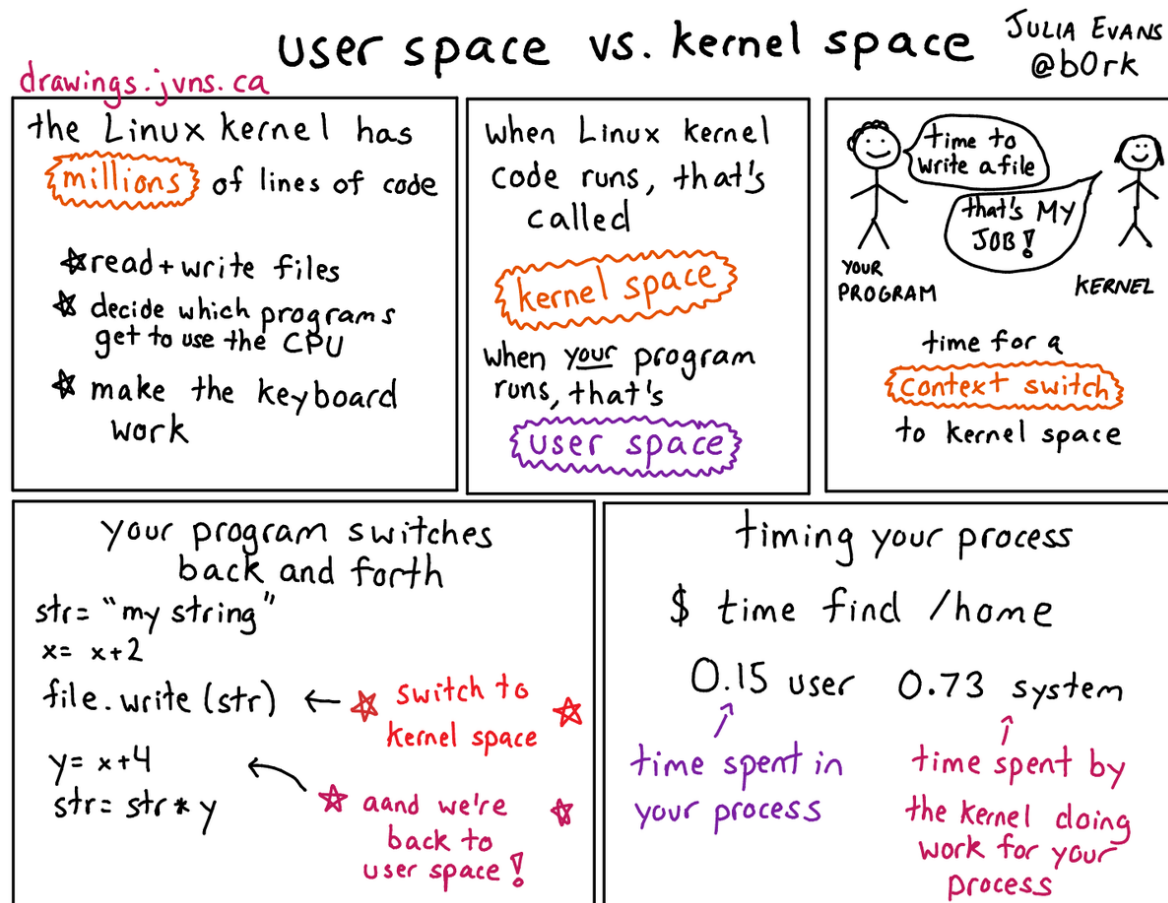
Microkernel
based Operating System



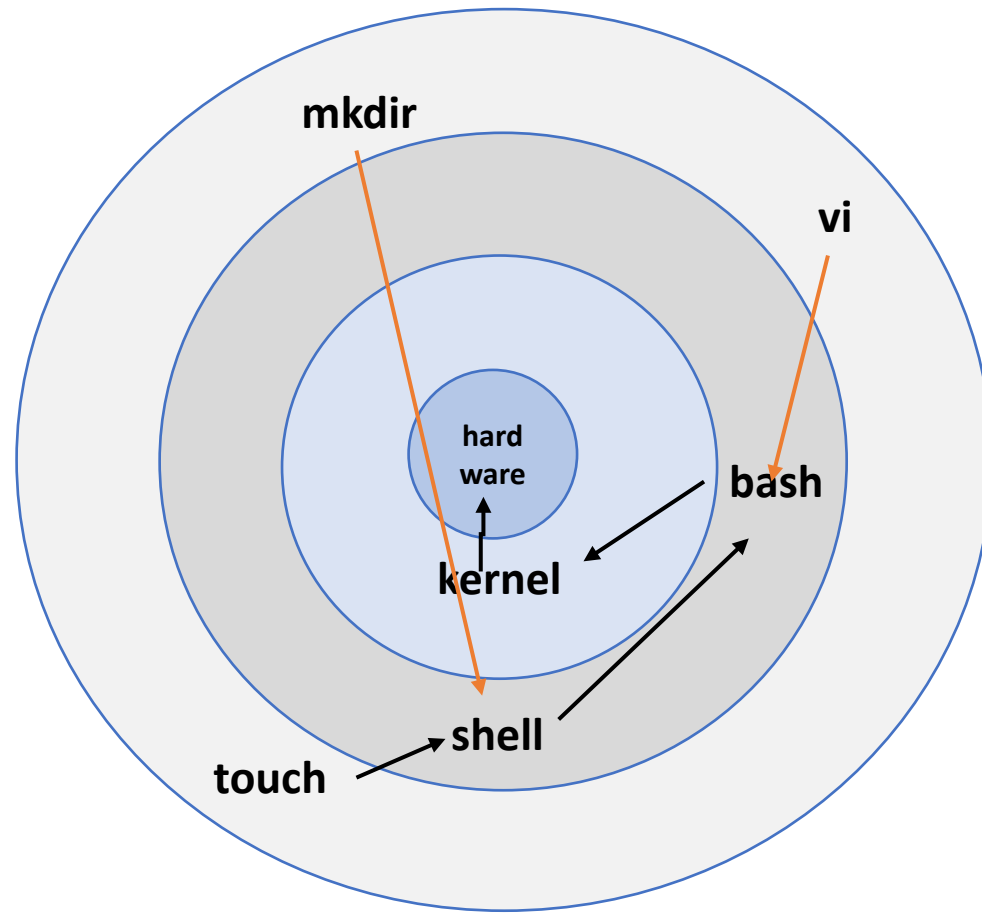
리눅스 커널



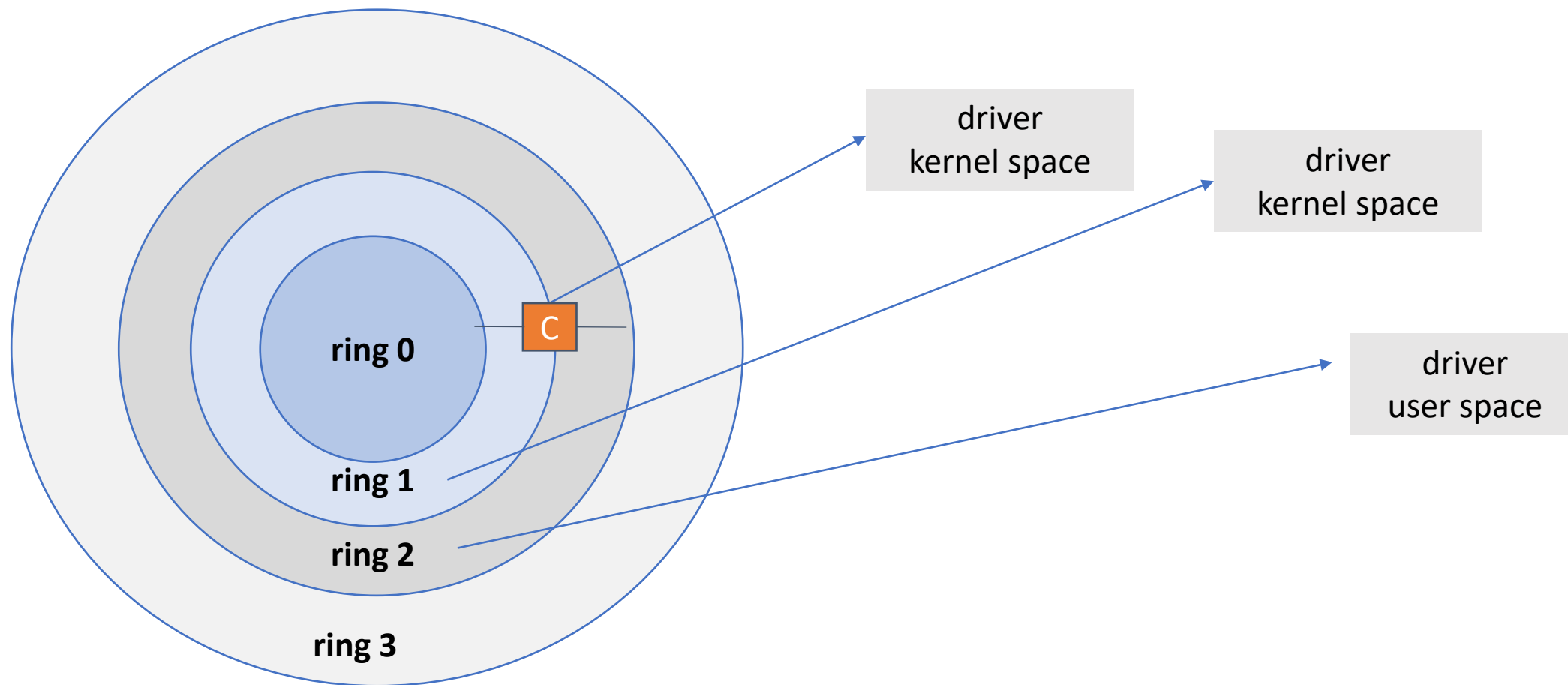
리눅스 커널



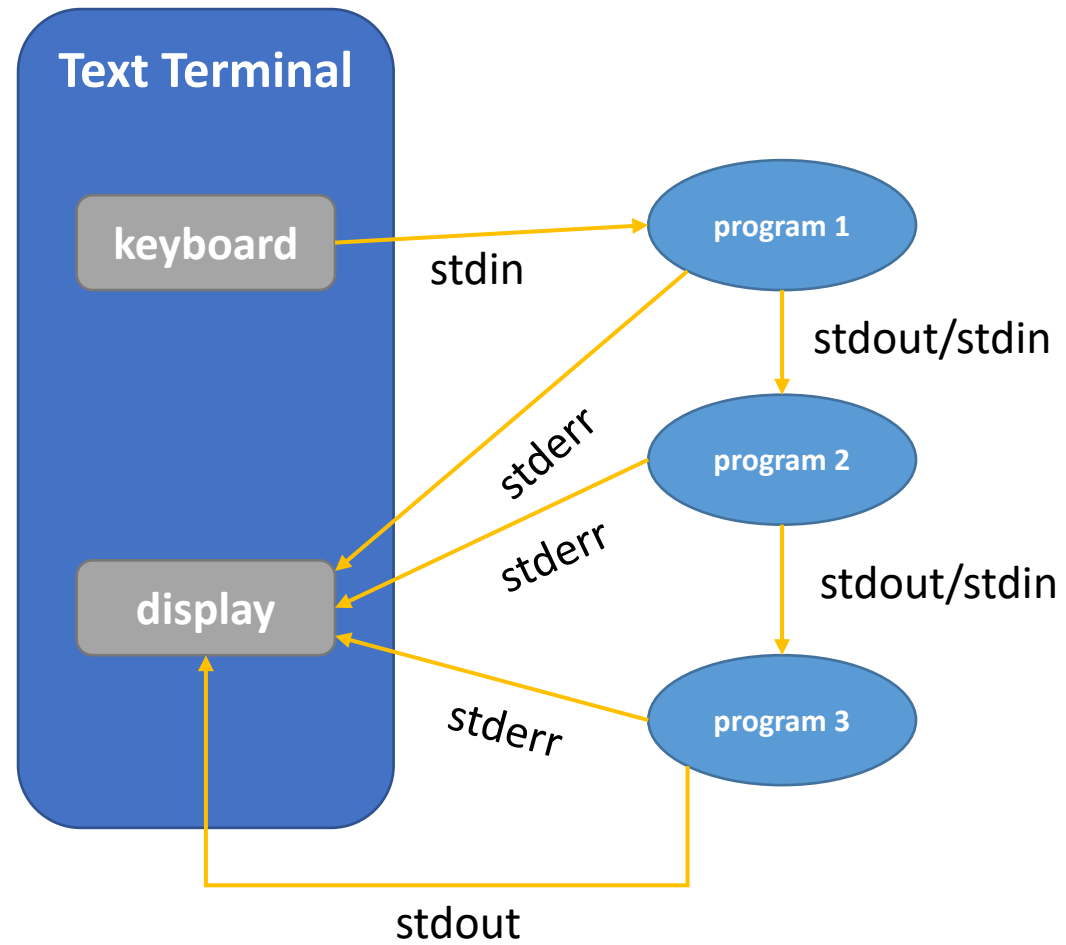
셸 + 커널 동작 방식



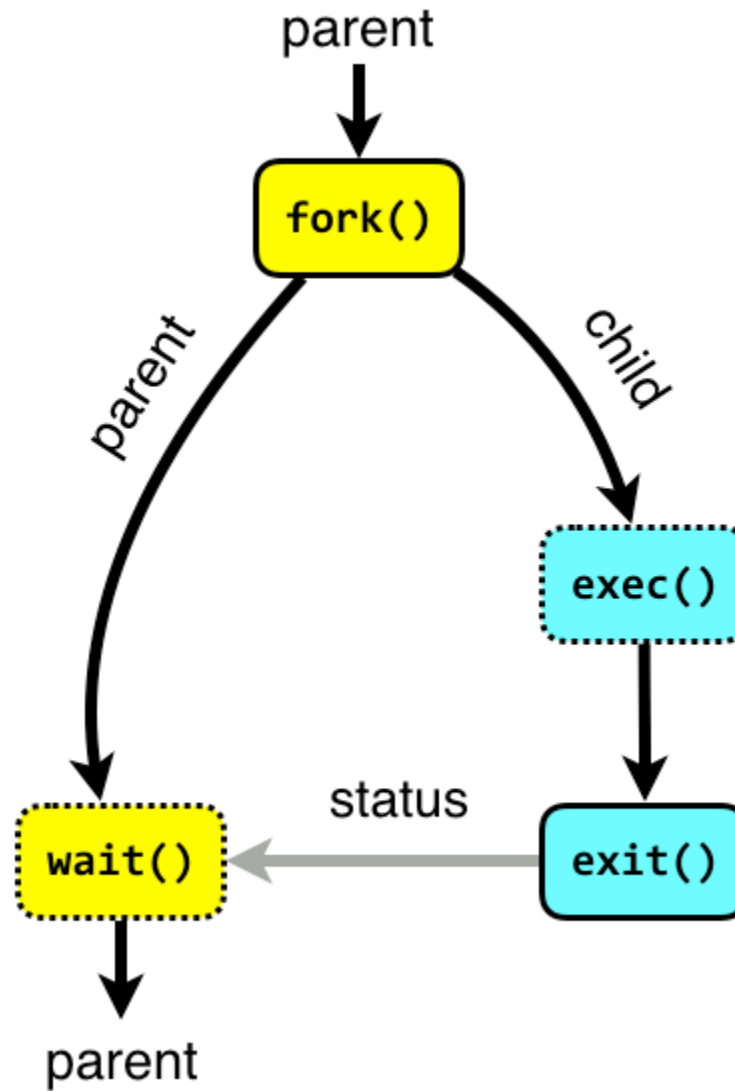
호스트 링 구조



Fork



셸 기초



웹 기초

웹 기초 내용

KSH

KSH은 유닉스에서 많이 사용하는 쉘 중 하나.
보통 다음과 같은 유닉스 시스템에서 많이 사용한다.

BSD계열

Unix 계열(HP, AIX, Solaris)

셸 기초

Bash 버전 확인하는 방법은 다음과 같다.

```
bash$ declare --p BASH
```

```
BASH="/usr/local/bin/bash"
```

chsh에서 **bash**로 변경하는 경우 아래와 같이 변경이 가능하다.

```
$ chsh -s /usr/local/bin/bash
```

셸 기초

리눅스 시스템에서 사용하는 셸 목록은 아래와 같이 확인이 가능하다.

```
$ chsh -l
```

설치가 안되어 있으면,

```
# yum install chsh -y
```

베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다.

이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장 가능하다.

우리는 이 기능을 **와일드카드(wildcard)**라고 부른다.

BASH 이동성

bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다.

이러한 이유로 POSIX를 셸 스크립트를 표준적으로 지원하며, 각기 다른 런타임 환경에서 사용하는 경우 bash는 강력한 도구가 될 수 있다.

마치, 자바처럼 셸 프로그램이 이식 되어 있으면 손쉽게 사용이 가능하다.

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

명령어 구조

bash

이미 알고 계시겠지만...

기본적인 내용이지만 좀 더 자세하게 기본 내용을 확인 하도록 한다.
만약 잘 아신다고 생각하시면...



간단한 명령어

bash에서 명령어 사용 방법은 다음과 같다.

```
$ mkdir -p testdir/subdir
```

매우 간단하다! :)")

간단한 명령어

하지만, 이러한 방식으로만 사용하지 않는다.

앞에서 이야기 하였던 쉘에서 사용하는 **특수문자(metacharacters)**가 있다.
예를 들어서 다음과 같다.

```
$ touch test file
```

```
$ ls -l
```

```
test
```

```
file
```

위와 같이 나온다. 여기에 좀 더 복잡하게 특수문자를 사용하면 다음처럼 결과가 나온다.

간단한 명령어

\$ touch report<SRV1>.txt

\$ touch report;Final.txt

위의 명령어를 실행을 하면 올바르게 실행이 되지 않는다.

왜?

bash 명령어 구조



설명해라 강사야!!

간단한 명령어

몇 가지 명령어를 더 테스트 해본다.

```
$touch $$$likeMoney.txt
```

```
$ ls
```

```
12234likeMoney.txt
```

역시 올바르게 실행이 되지 않는다. 왜 이럴까요?

간단한 명령어

이러한 문제를 해결 하기 위해서 **쿼트(Quoting/Quote)**을 사용해야 한다.

보통 다음과 같은 방법이 있다.

'<string>'

"<string>"

\<escape>\<escape>

특수문자 해결 방법

\$ touch report\<SRV1\>.txt

\$ touch report\;Final.txt

\$ touch \\$\\$likeMoney.txt

그러면 만약 백 슬래시(\)를 출력하는 방법?

```
$ echo \\backslash\\  
\\backslash\\
```

그냥 두 번 입력하면 된다. 첫 번째는 **탈출문자(metacharacter)**, 두 번째는 문자로써 입력.

그러면 만약 백 슬래시(\)를 출력하는 방법?

```
$ echo backslash\
```

```
> hello \
```

```
> ygl
```

```
backslashhelloygl
```

특수문자

\$ touch 'report document'

\$ touch 'report<SRV1>.txt'

\$ touch 'report;Final.txt'

\$ touch '\$\$\$report.txt'

싱글쿼트

싱글 쿼트 줄 내림(Single Quote)

```
$ echo 'quotes
```

```
> foo
```

```
> bar'
```

```
quotes
```

```
foo
```

```
bar
```

싱글쿼트

```
$ echo 'it\'s today'
```

```
$ echo 'it"s today'
```

```
$ echo 'it\'s today'
```

어떤 게 출력이 될까요?

더블쿼트

```
$ echo "This is my shell: $SHELL"
```

```
$ echo 'This is my shell: $SHELL'
```

```
$ echo "The variable: \ $total"
```

```
$ echo "Back\\to\\back\\back\\slashes\\_world"
```

```
$ echo "hello $USER""!!"
```

백틱 (backticks)

```
$ echo "backticks: \`\`\`"
```

복합적 쿼트

```
$ echo Hello, \ "$USER"! Hell to the "$HOSTNAME"!"
```

Shell Exit Values

간단하게...

\$ echo \$?

\$ true ; echo \$?

\$ false ; echo \$?

1: False

0: True

Shell Exit Values

명령어가 없는 경우...

\$ thereisnocommand ; echo \$?

127

Shell Exit Values

시그널 코드(이름)	시그널 값	효과
SIGHUP	1	Hangup
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal
SIGSTOP	17,19,23	Stop the process

bash ctrl combination

배쉬 네비게이션

Ctrl + A	CLI 맨 처음으로 커서를 이동 합니다.
Ctrl + E	CLI 맨 끝으로 커서를 이동 합니다.
Ctrl + F	글자 하나씩 앞으로 이동 합니다.
Ctrl + B	글자 하나씩 뒤로 이동 합니다.
Ctrl + XX	커서를 CLI의 맨 처음과 끝으로 위치를 바꾸면서 이동 합니다.
Alt + F / Esc + F	단어 단위로 이동 합니다. 이동 시 기준은 빈 공간 혹은 세미콜론이 기준이 됩니다.
Alt + B / Esc + B	반대로 단어 단위로 이동 합니다.

bash ctrl combination

배쉬 프로세스 관리

Ctrl + L

화면에 출력된 내용을 전부 지웁니다.

Ctrl + S

화면에 출력을 중지 합니다.

Ctrl + Z

현재 실행중인 프로세스를 일시적으로 백그라운드로 중지 합니다.

Ctrl + Q

일시적으로 중지된 프로세서를 재개 합니다.

Ctrl + C

프로세스에 SIGI시그널을 보내서 현재 실행중인 프로세스 중지 합니다.

Ctrl + D

현재 사용중인 터미널을 종료 합니다.

bash ctrl combination

Bash History

Ctrl + R	배쉬 명령어 기록을 역순으로 검색 합니다.
Alt + P	콜론이 출력이 되면, 실행하였던 명령어 번호 혹은 명령어 일부분을 입력하면 됩니다.
Ctrl + P / Up arrow	과거에 사용했던 명령어를 조회 합니다.
Ctrl + N / Down arrow	과거 기준으로 사용하였던 명령어 조회 합니다.
Ctrl + S	화면을 잠깐 중지 합니다. 취소는 Ctrl + Q를 누르면 됩니다.
Ctrl + O	Ctrl + R로 명령어 선택 후 실행 합니다.
Ctrl + G	Ctrl + G로 종료 합니다.
!!	마지막에 실행한 명령어를 수행 합니다.
!*	이전에 수행한 명령어를 실행 합니다.

bash ctrl combination

Bash Editing

Ctrl + U	커서 뒤로 있는 모든 문자열을 제거 합니다.
Ctrl + K	커서 뒤로 있는 단어 혹은 문자를 제거 합니다.
Ctrl + W	명령어 혹은 인자 값 커서 기준으로 뒤로 제거 합니다.
Ctrl + D	한 글자씩 제거 합니다.
Ctrl + H	커서가 있는 위치에 문자를 제거 합니다.

bash ctrl combination

Alt + D	커서 뒤로 있는 문자들을 제거 합니다.
Alt + Backspace	단어별로 삭제 합니다.
Alt + . / Esc+.	마지막에 실행한 명령어 인자 값도 같이 불러옵니다.
Alt + <	히스토리에 맨 처음에 있는 기록을 출력 합니다.
Alt + >	마지막에 실행한 명령어를 출력한다.
Esc + T	두 단어의 위치를 커서 기준으로 변경 합니다.
Alt + T	현재 단어 위치를 앞 단어와 변경 합니다.

bash ctrl combination

Bash Information

TAB	명령어 혹은 자원에 대해서 자동완성을 합니다.
~TAB TAB	현재 등록중인 모든 사용자를 출력 합니다.
Ctrl + I	단어로 시작된 명령어를 화면에 출력 합니다.
Alt + *	현재 디렉터리에서 파일 혹은 디렉터리에 맞는 값들을 화면에 출력 합니다.

명령어 연속 실행 시 오류 중지

여러 명령어를 수행하는 방법은 두 가지가 있다.

```
$ cd ; rm -r /testdir ; ls
```

```
$ cd / && rm -r /testdir ; ls
```

첫 번째 방법은 오류가 발생하여도 다음명령어가 실행이 된다.

두 번째 방법은 첫 번째 명령어가 실행 중 문제가 발생하면 중지.

포그라운드/백그라운드

명령어 수행 시 포그라운드/백그라운드 실행은 다음과 같은 명령어로 실행이 가능하다.

\$ sleep 10

\$ sleep 10 &

포그라운드/백그라운드

백 그라운드로 실행이 된 명령어를 확인 하려면 아래와 같은 명령어로 확인이 가능하다.

\$ bg

다시 포그라운드로 실행하려면 다음 명령어로 포그라운드로 실행이 가능하다.

\$ fg <ID>

변수

bash

변수

셸에서 변수는 다음과 같이 선언한다.

```
tangshell='This is mine shell'
```

tangshell를 변수 이름이 되며, 그 뒤에 홑 따옴표에 있는 내용은 셸 변수에서 사용하는 **자료(data)**가 된다.

이걸 **BASH**에서는 "**string content**"라고 한다.

변수

bash는 c 스타일을 사용하기 때문에 변수 이름에 절대 빈 공간이 있으면 안된다. 보통 다음과 같이 많이 선언한다.

```
tangshell = 'fish'
```

```
tangvar='GNU'\''s Not Unix!'
```

```
tangprompt="$USER$HOST"
```

변수를 호출 할 때 c언어와 마찬가지로 "\$"붙여서 접근이 가능하며, **읽기전용(read only)**상태이다.

변수 목록

변수 목록 및 출력을 하기 위해서는 다음과 같은 방법을 사용해서 호출한다.

```
$ today="2022-06-26"
```

```
$ printf '%s\n' "$today"
```

```
2022-06-26
```

```
$ declare -p today
```

```
declare -- today='2022-06-26'
```

변수 목록

'declare'에 표시되는 "--"는 특별한 속성이 아니라는 표시이다.

'declare -p'명령어를 실행하면 다음과 같은 내용을 볼 수 있다.

\$ declare -p

```
declare -- BASH="/bin/bash"
```

```
declare -- PS1="\u@\h:\w\$"
```


변수 네이밍

변수 네이밍에 대한 규칙은 따로 없지만 일반적으로 다음과 같은 규칙을 가진다.

test_var

TestVar

testvar

Testvar

_testvar

testVar

TestVar_

변수 네이밍

하지만 다음과 같은 이름은 허용하지 않는다.

1testVar

test-var

test.var

test:var

변수 네이밍

또한 변수 설정 시 기존 설정된 이름과 충돌을 주의해야 한다. 매우 자주 있는 케이스는 보통 다음과 같다.

```
#!/bin/bash
PATH=/home/tang/sbin:/home/tang/bin
grep -- teststring "$PATH"
```

결과는 다음과 같이 출력이 된다.

```
grep: command not found
```

변수 네이밍

이유는 간단하다. \$PATH는 시스템 변수이며, 기존에 설정된 경로 값에 덮어쓰기가 되어버린 상태다. 그래서 path라는 이름을 사용한다면 다음처럼 선언을 보통 한다.

```
#!/bin/bash
```

```
path=/home/tang/bin:/home/tang/sbin
```

```
grep -- teststring "$path"
```

변수 네이밍

혹은 기존 **\$PATH**를 사용하면서 내용을 추가하는 경우, 다음처럼 처리가 가능하다.

```
PATH="$PATH:/home/tang/bin/:/home/tang/sbin"
```

```
MYHOME=/home/tang
```

```
export MYHOME
```

변수 취소

변수 초기화 하는 방법은 다음과 같다.

```
$ system_var =
```

그리고 초기화 된 내용을 확인 시 다음과 같은 방법으로 가능하다.

```
$ declare -p system
```

```
declare -- system=""
```

변수 확인

POSIX 기준으로 변수에 값이 설정이 되어 있는지 확인 시, 다음과 같이 if문으로 확인한다.

```
#!/bin/bash
```

```
test_system=linux
```

```
system=windows
```

```
if [[ -n $test_system ]] ; then
```

```
    printf 'the 'test_system' value is: %s\n' "$system"
```

```
fi
```

변수 취소

변수를 완전히 제거하려면 unset명령어를 통해서 할 수 있다.

```
$ unset -v system
```

```
$ declare -p system
```

```
bash: declare: system: not found
```


시스템 변수

```
$ bash
```

```
$ declare -p BASHPID
```

```
declare -ir BASHPID= "331219"
```

```
$ declare -p shellusername
```

```
declare: shellusername: not found
```

시스템 변수

다시 이전 환경으로 돌아가기 위해서는 **exit** 명령어 실행한다.

```
$ exit
```

```
$ declare -p BASHPID
```

```
$ declare -p shellusername
```

시스템 변수

만약 다른 환경에도 값을 전달하고 싶은 경우 **POSIX**명령어 **export**로 선언한다.

```
$ USERNAME="choi"
```

```
$ export USERNAME
```

```
$ declare -p USERNAME
```

```
declare -x USERNAME="choi"
```

시스템 변수

앞에서 이야기 하였지만, 변수 이름 설정 시 전부 대문자로 설정한다.
'**export**'로 설정하는 경우 '**declare**'명령어로 변수를 확인하면 다음처럼 변수 이름이 출력이 된다.

```
$ bash
```

```
$ declare -p USERNAME
```

```
declare -x USERNAME="choi"
```

```
$ exit
```

시스템 변수

환경변수는 **bash**에서만 사용하는 게 아니다.

모든 프로그램 및 운영체제 환경에서 적용이 되며, 다른 프로그램 언어 **Python, PHP, Awk**같은 명령어에서도 적용이 된다.

예를 들어서 '**sort**', '**date**' 혹은 '**ls**'같은 명령어는 몇몇 시스템 변수에 적용이 받는데 그중 하나가 **LANG, LC_ALL, LC_COLLATE**변수이다.

시스템 변수

```
$ printf '%s\n' 'helloworld' 'HelloTang' > words
```

```
$ LC_COLLATE=en_US.UTF-8
```

```
$ export LC_COLLATE
```

```
$ sort words
```

```
helloworld
```

```
HelloTang
```

시스템 변수

\$ LC_COLLATE=C

\$ sort words

HelloTang

helloworld

시스템 변수

\$ LC_COLLATE=C sort words

HelloTang

helloworld

일반변수

변수 이름을 좀 더 명확하게 보기 위해서 중괄호로 표시를 할 수 있다.

```
$ yourname='tang'
```

```
$ printf '%s\n' "__Hello, ${yourname}__."
```

```
__Hello, Tang__.
```

일반변수

역시 변수에 값이 선언이 되어 있지 않는 경우에는 다음처럼 출력이 된다.

```
$ printf '%s\n' "Hello, $notest."
```

Hello, .

일반변수

만약에 선언이 되지 않는 변수에 대해서 오류 메시지를 출력하고 싶으면 다음처럼 선언한다.

```
$ set -u
```

```
$ printf '%s\n' "Hello, $notest."
```

```
bash: notest: unbound variable
```

일반변수

변수를 외부에서 받아 오기 혹은 읽어오기 할 때 내장함수인 **read**를 사용해서 사용한다.

이때 같이 사용하는 옵션은 **"-r"**를 같이 사용해서 변수에 할당한다.

```
$ read -r yourname
```

```
$ declare -p yourname
```

```
declare -- yourname='tang'
```

명령어 결과를 변수로

종종 사용하는 방법으로, 실행된 명령어의 결과를 변수에 저장한다. 이유는 여러가지가 있지만 일반적으로 재가공 혹은 출력 용도로 사용한다. 다음과 같은 예제를 본다.

```
$ whom=$(whoami)
```

```
$ printf '%s\n' "$whom"
```

```
tang
```

명령어 결과를 변수로

두개 이상의 명령어를 동시에 사용해야 되는 다음처럼 명령어를 구성한다.

```
$ yourpasswd="$(getent passwd $(whoami))"
```

기본값 설정

기본값 설정을 종종 필요하다.

예를 들어서 시스템에서 올바르게 기본값이 설정이 안되어 있으면 올바르게 동작이 안될 수 있다.

그래서 쉘에서는 `"${default_var:-default}"` 같은 형식을 제공한다.

```
$ cd -- "${MYDATA_DIR:-/usr/local/data}"
```

혹은 다음과 같이 재선언이 가능하다.

```
$ cd -- "${MYDATA_DIR:=/usr/local/data}"
```

루프 및 조건문

조건문

셸에서 조건문을 사용하기 위해서 다음과 같은 조건이 필요하다.

if

[, [[

((

case

for

while

조건문

if문을 사용하기 위해서는 다음과 같은 방법으로 구현이 가능하다.

```
$ grep -q bash /etc/shells && printf "bash is a system shell\n"
```

먼저 중간에 있는 "&&"기호는 뒤에 두번째 명령어를 실행하는데, 이때 조건이 있다.

"-q" bash 문자열이 /etc/shells에 있으면 true를 반환하면, 그 다음 명령어 printf를 실행한다.

만약, 해당 문자열이 없으면 false반환이 되기 때문에 뒤에 명령어가 실행이 되지 않는다.

조건문

이걸 if문으로 변경하면 다음처럼 된다.

```
$ if grep -q bash /etc/shells ; then printf 'bash is a system shell\n' ; fi
```

좀 더 읽기 편하게 하기 위해서 다음처럼 구현이 가능하다.

```
if grep -q bash /etc/shells ; then  
    printf 'bash is a system shell\n'  
fi
```

조건문

if 명령어는 반드시 **fi**로 종료를 해야 한다.

```
$ if1.sh
```

```
if cd /usr/local ; then
```

```
    printf ">&2 'changed directory\n'
```

```
fi
```

조건문

부정적인 조건은 아래처럼 사용이 가능하다. 간단한 조건문은 대괄호로 감싸지 않아도 된다.

```
$ if2.sh
```

```
if ! cd /usr/local/ ; then
```

```
    printf ">&2 'Failed to changed directory\n'
```

```
    exit 1
```

```
fi
```

조건문

if문에서 조건이 맞지 않았을 때 동작하는 else문법을 구현하기 위해서는 **elif**키워드를 사용한다.

```
$ if4.sh
```

```
if cd /usr/local/mysql ; then
    printf >&2 'Changed to primary directory'
elif cd /opt/mysql ; then
    printf >&2 'Changed to secondary directory'
else
    printf >&2 'Cound\'\"t find a directory!'
    exit 1
fi
```

조건문

위와 같이 조건문을 사용하는 경우는 간단하게 true, false인 경우만 사용이 가능하다.

좀 더 고급 적이고 복잡한 조건문을 사용하는 경우 아래와 같이 조건문을 사용해야 한다.

조건문

- [-a FILE] 파일이 존재하면 참.
- [-b FILE] 파일이 존재하고 블록파일이면 참.
- [-c FILE] 파일이 존재하고 특별한 캐릭터 파일이면 참.
- [-d FILE] 파일이 존재하고 그리고 디렉터리가 존재하면 참.
- [-e FILE] 파일이 존재하면 참.
- [-f FILE] 파일이 존재하고 일반 파일이면 참.

조건문

[-g FILE] 파일이 존재하고 SetGID가 설정이 되어 있으면 참.

[-h FILE] 파일이 존재하고 심볼 링크면 참.

[-k FILE] 파일이 존재하고 Sticky이면 참.

[-p FILE] 파일이 존재하고 FIFO이면 참.

[-r FILE] 파일이 존재하고 읽기전용이면 참.

[-s FILE] 파일이 존재하고 파일 크기가 0보다 크면 참.

조건문

- [-t FD] 파일이 FD(file Descriptor)이며, 터미널이 열려 있으면 참.
- [-u FILE] 파일이 존재하고 SUID가 설정이 되어 있으면 참.
- [-w FILE] 파일이 존재하고 쓰기가 가능하면 참.
- [-x FILE] 파일이 존재하고 실행이 가능하면 참.
- [-O FILE] 파일이 존재하고 사용자가 소유하고 있으면 참.
- [-G FILE] 파일이 존재하고 그룹 사용자가 소유하고 있으면 참.
- [-L FILE] 파일이 존재하고 심볼릭 링크인 경우.
- [-N FILE] 파일이 존재하고 마지막으로 읽은 이후 수정된 경우 참.
- [-S FILE] 파일이 존재하고 소켓 파일인 경우.

Beep!!

POSIX기준으로 스크립팅 언어에서 **test**라는 도구를 사용할 수 있다.
그냥 **test**명령어만 사용하는 경우 다음처럼 사용하면 된다.

```
$ yourshell=bash
```

```
$ test "$yourshell" = 'sh' && printf 'DingDong!\n' echo -ne '\007\'
```

```
$ test "$yourshell" = 'bash' && printf 'DingDong!\n' echo -ne '\007\'
```

조건문

-e: file이라는 디렉터리나 파일이 있으면 참

```
$ test -e 'file':
```

```
$ echo $?
```

-n: \$vars라는 변수가 선언이 되어 있고 비어있 지 않는 경우.

```
$ test -n "$vars"
```

```
$ echo $?
```

if

'if'문은 다음과 같이 사용한다. 좀 더 복잡한 형태로 조건문을 사용하려면 대 괄호 기반으로 'if'문법을 사용하며

```
if [ -f /var/log/message ]
```

```
fi
```

if elif else fi

'if'문에서 'else if'는 'elif'키워드로 사용이 가능하다.

또한 'else', 'elif'를 사용하는 경우 반드시 'then'키워드를 같이 사용해야 한다.

if [condition]; then

elif [other_condition]; then

else

fi

if elif else fi

```
$ vi elifelse.sh
```

```
#!/bin/bash
seconds=$2
if [ "$seconds" -eq 0 ] ; then
    timezone_string="Z"
elif [ "$seconds" -gt 0 ] ; then
    timezone_string=$(printf "%02d:%02d" $((seconds/3600)) $(((seconds / 60) % 60)))
    $timezone_string
else
    echo "Unknown parameter"
fi
```

while/until

'while'조건이 참이면 코드를 계속 실행 한다. 하지만, 값이 거짓인 경우에는 중지합니다.

```
$ while.sh
```

```
#!/bin/bash  
COUNTER=0  
while [[ $COUNTER -lt 10 ]]; do  
    echo The counter is $COUNTER  
    let COUNTER=COUNTER+1  
done
```


while/until

'until'은 'while'과 비슷하지만, 조금 다른 부분은 코드가 실행된 후에 거짓이면 계속 실행 참이면 반복문을 종료 합니다.

\$ until.sh

```
#!/bin/bash
```

```
COUNTER=20
```

```
until [ $COUNTER -lt 10 ]; do
```

```
    echo COUNTER $COUNTER
```

```
    let COUNTER-=1
```

```
done
```

for

'for' 반복문은 "**End Of Array**" 카운터 변수에 입력된 개수만큼 동작한다.
중지되는 기준은 이다.

\$ for.sh

```
#!/bin/bash  
for i in $( ls ); do  
    echo item: $i  
done
```

루프 및 조건문

조건문

-d: 디렉터리 이름이 dir인 경우

```
$ test -d 'dir'
```

```
$ echo $?
```

-s: 파일이 존재하며 파일이 비어 있지 않는 경우

```
$ test -s 'file'
```

```
$ echo $?
```

test, [, [[

test 명령어는 [명령어.

[명령어는 두 개의 명령어가 있어야 올바르게 동작한다. "[", "]"이 두개의 명령어는 쉘 스크립트에서 많이 사용하며, 또한 이것들은 그냥 명령어이다.

셸 스크립트의 키워드와 다르다.

```
$ [ "$yourshell"='bash' ] && printf 'Correct!\n'
```

if문은 grep, cd명령어처럼 사용이 가능하다.

```
yourshell=bash
if [ "$myshell" = 'bash' ] ; then
    printf 'Correct!\n'
fi
```

test, [, [[

이전 bash는 "["를 사용하였지만, bash 4.0이후부터는 "[["으로 변경되었다.

이 문법은 bash에서 특별한 문법이며 일반적인 빌트인 명령어가 아니다.

```
#!/bin/bash
```

```
yourshell=bash
```

```
if [[ $yourshell = 'bash' ]] ; then
```

```
    printf 'correct!\n'
```

```
fi
```

test, [, [[

"[, "["하고 제일 큰 차이점은 다음과 같다.

```
if [ "$myshell" = 'bash' ] ; then
```

```
if [[ $yourshell = 'bash' ]] ; then
```

"[" 경우에는 변수를 쌍 따옴표로 감싸야 한다.

test, [, [[

구체적인 차이는 다음과 같다.

"["강화 버전은 "[["이며, 강화가 된 이유는 이전 bash 3.x때 치명적인 버그로 파싱 부분에 대해서 강화가 필요 하였다.

이 부분은 다음처럼 사용이 가능하다.

if [-f "\$file"] -> if [[-f \$file]]

test, [, [[

```
[[ $yourshell = "$yourshell" ]]
```

하지만, 기존 방식하고 혼동이 될 수 있기 때문에 그냥 이전에 사용하던 방식으로 사용해도 문제가 없다.

```
[[ "$yourshell" = "$yourshell" ]]
```

[방식과 비슷하지만, [[에서 크게 문제 없이 동작한다.

호환성을 유지하기 위해서는 "["로 사용을 권장한다.

test, [, [[

기존에 사용하였던 **test**, "["처럼 "["에서도 조건문 사용이 가능하다.

```
[[ $yourshell = b* ]]
```

위와 같이 실행하는 경우 이전 앞에서 사용한 글러빙 처럼 "b"로 시작하는 문자열을 찾는다.

test, [, [[

혹은 다음 "`=~`" 새로운 지시자(operator)를 사용해서 시작 문자를 지정할 수 있다.

```
[[ $yourshell =~ 'sh$' ]]
```

위의 명령어를 실행하면 마지막 문자가 sh로 끝나는 찾아서 화면에 출력한다.

test, [, [[

복잡한 오퍼레이션 사용이 필요한 경우 좀 더 간단하게 표현이 가능하다.

```
if [ "$answer" = y -o "$answer" = yes ]
```

```
if [[ $answer =~ ^y(es)?$ ]]
```

이전에 사용이 가능하였으나, 사용하기 조건이 복잡하였던
틸드(~)도 파싱 및 사용이 가능하다.

산술조건(Arithmetic condition)

POSIX기준으로 **'test'**, **"["** 명령어를 사용해서 산술 연산자 사용이 가능하다.

["\$a" -eq "\$b"]: 같다

["\$a" -ne "\$b"]: 같지 않다

["\$a" -lt "\$b"]: 작거나

["\$a" -le "\$b"]: 작거나 혹은 같거나

["\$a" -gt "\$b"]: 크거나

["\$a" -ge "\$b"]: 크거나 혹은 같거나

산술조건(Arithmetic condition)

바이트, 키로바이트 출력을 위한 조건문을 적으면 아래와 같이 1,000으로 나누어서 비교하는 조건문을 아래와 같이 일반적으로 작성한다.

```
if [ "$((bytes / 1000))" -gt "$kbytes" ] ; then
```

```
fi
```

산술조건(Arithmetic condition)

하지만 bash에서 산술을 위한 특별한 문법이 있다.

"\$"대신 "("으로 표현이 가능하다.

```
if ((bytes / 1000 > kbytes )) ; then
```

```
fi
```

산술조건(Arithmetic condition)

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

`((2 > 0))` : True, exit 0

`((a = 1))`: True, exit 0

`((0 >3))`: False, exit 1

`((0))`: False, exit 1

`((a = 0))`: False, exit 1

산술조건(Arithmetic condition)

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

`((2 > 0))` : True, exit 0

`((a = 1))`: True, exit 0

`((0 >3))`: False, exit 1

`((0))`: False, exit 1

`((a = 0))`: False, exit 1

기본 명령어

bash

bash 기본 명령어

셸에서 기본적으로 제공하는 **내부 명령어(internal command)**가 있다.

이 명령어를 통해서 좀 더 편하게 셸 환경을 꾸밀 수 있다.

셸 프롬프트를 꾸미고 싶으면 아래와 같이 사용이 가능하다.
PS1에서는 다음과 같이 활용하여 프롬프트를 꾸민다.

```
$ export PS1="\u@\h \w> "
```

\u: 사용자 이름(username)

\h: 호스트 이름(hostname)

\w: 전체 경로(full pathname)

bash 기본 명령어

PS2는 계속 연속적으로 대화형으로 입력이 가능하도록 하는 프롬프트.

대표적으로 **PS2**에서 다음과 같은 옵션으로 사용한다. 자세한 내용은 아래에서 더 설명한다.

보통은 백 슬래쉬로 표현한다.

```
$ echo Hello This is \
```

```
> World \
```

```
> Tang
```

```
" ":
```

```
$ echo export PS2="continue-> "
```

위와 같이 표현하는 경우, 화면에 아래와 같이 표시가 된다.

```
$ echo Hello This is\
```

```
continue-> World \
```

```
continue-> Tang
```

bash 기본 명령어

PS3는 스크립트에서 출력하는 프롬프트를 설정한다.

```
PS3="Select a day (1-4): "  
select i in one two three exit  
do  
  case $i in  
    one) echo "One";;  
    two) echo "Two";;  
    three) echo "Three";;  
    exit) exit;;  
  esac  
done
```

bash 기본 명령어

PS4

아래서 이야기 할 'set -x'와 같은 역할. 쉘 스크립트가 동작하면서 발생하는 이벤트를 화면에 출력한다. 다만, 좀 더 자세하게 오류를 출력하기 위해서 아래와 같은 옵션을 추가한다.

이 부분은 'set'에서 한번 더 다룬다.

명령줄

```
# set -x test.sh
```

스크립트

```
#!/bin/bash -x
```

```
export PS4='$0.$LINENO+ '
```

bash 기본 명령어

PS5

PS5는 PS1과 비슷한 동작 하지만, PS1의 값을 수정하지 않고 추가적으로 내용을 더 추가하여 화면에 출력한다.

```
export PROMPT_COMMAND="echo -n [$(date +%k:%m:%S)]"  
[ 9:10:14][root@test ~]#
```

빌트인 명령어

내부 명령어는 외부에 별도의 명령어 없이 실행이 가능한 내장형 명령어.

대표적인 명령어는 다음과 같다.

echo, type, source

런타임 명령어

런타임 명령어(runtime)는 기본적으로 쉘 스크립트(bash)명령어 기반으로 구성되어 있다.

이러한 도구들은 보통 function(), alias로 구성되어 있으며, 자체적으로 실행은 불가능하며, 쉘을 통한 조합 실행이 가능하다.

시스템 명령어 혹은 외부 명령어

시스템 명령어는 파일 시스템에 독립적으로 구성이 되어 있는 프로그램이다.

이러면 명령어는 어떠한 패키지가 설치가 되어 있느냐에 따라서 다르다.

대표적인 명령어는 다음과 같다.

grep, ping, rm, ls, nmcli

셸 대표 명령어

type: 명령어 형식 확인. 셸에 alias로 구성이 되어 있는 경우 alias를 출력한다.

echo: 문자열 출력

printf: echo와 비슷하지만 포매팅 형식으로 출력

pwd: 셸의 현재 위치 출력

cd: 셸 위치 변경

set: 현재 셸에 설정된 값을 출력

declare: 변수나 함수(function)를 관리하는 명령어

test, [, [[: 정규식 구성, 대괄호로 구성이 된다.

대표 명령어

ls: 디렉터리 및 파일 출력

mv: 파일 이동 및 이름 변경

cp: 파일 복사

rm: 파일 및 디렉터리 제거, 다른 명령어로 rmdir이 있다.

grep: 문자열 검색하는 정규식 도구

cut: 행 단위로 자르는 명령어

wc: 줄 및 문자열 카운팅 도구

find: 파일이나 혹은 디렉터리를 특정 조건으로 찾는 도구

sort, uniq: 정렬 및 중복제거 하는 도구

type

bash에서 명령어 사용 방법이 궁금한 경우 아래와 같이 사용이 가능하다.

```
bash$ help type
```

type명령어를 사용하면 다음과 같이 출력한다.

```
bash $ type cp
```

```
cp is /usr/bin/cp
```

```
bash $ type ls
```

```
ls is aliased to `ls --color=auto`
```

```
bash $ help cp
```

```
-sh: help: no help topics match `cp`. Try `help help' or `man -k cp' or `info cp'.
```

```
bash $ type for
```

```
for is a shell keyword
```

type

function()를 type으로 확인하기.

```
bash $ myfunc() { : ; }
```

```
bash $ type myfunc
```

```
myfunc is a function
```

```
myfunc()
```

```
{
```

```
:
```

```
}
```

```
bash $ alias myalias=:
```

```
bash $ type myalias
```

```
myalias is aliased to ':'
```

type

```
bash $ type -a true
```

```
true is a shell builtin
```

```
true is /bin/true
```

echo

```
bash $ echo hello  
hello
```

```
bash $ echo 'Hello, "$USER"\!  
Hello, bashuser!
```

printf

```
bash $ printf '%s\n' 'Hello!'  
Hello!
```

이 명령어는 C의 printf와 흡사한 포매팅 구조를 가지고 있다.

echo vs printf

'echo', 'printf' 명령어 둘 다 셸에서 사용이 가능하다. 다만 다음과 같은 차이점으로 'echo'가 아닌 'printf'를 사용하기도 한다.

echo

실행 시 "0"이외 리턴을 하지 못한다. 그러기에 출력에서 오류가 발생하였는지 확인이 어렵다.

printf

"0"이외 리턴이 가능하기 때문에 출력 오류에 대해서 확인이 가능하다. 또한 기본적으로 'printf'는 'echo'명령어보다 수행속도가 빠르다.

echo vs printf

하지만, 쉘 버전이 업데이트 및 업그레이드가 되면서 현재는 'echo', 'printf'의 실행 속도에 큰 차이가 없지만 일반적으로 'printf' 명령어가 더 빠르다.

성능 테스트 하기 위해서 다음과 같이 명령어를 실행한다.

```
$ time for i in {1..999999}; do echo "$i" >/dev/null; done
```

```
real  0m20.095s
```

```
user   0m10.509s
```

```
sys    0m9.566s
```

```
$ time for i in {1..999999}; do printf "$i" >/dev/null; done
```

```
real  0m18.851s
```

```
user   0m10.115s
```

```
sys    0m8.715s
```

echo vs printf

'printf' 명령어가 'echo'와 비슷하게 동작하기 위해서는 다음과 같이 실행한다.

```
$ time for i in {1..999999}; do printf "$i\n" >/dev/null; done
```

```
real  0m19.614s
```

```
user  0m10.678s
```

```
sys   0m8.881s
```

빌트인?

몇몇 명령어는 시스템에 존재하지만, 실제로는 쉘의 빌트인 명령어.
다음과 같은 명령어로 확인이 가능하다.

type echo

echo is a shell builtin

whereis alias

alias: /usr/bin/alias /usr/share/man/man1/alias.1.gz

file /usr/bin/alias

/usr/bin/alias: a /usr/bin/sh script, ASCII text executable

cat /usr/bin/alias

#!/usr/bin/sh

builtin alias "\$@"

printf

```
bash $ printf '%s\n' -n
```

```
-n
```

```
bash $ string=-n
```

```
bash $ printf '%s\n' "$string"
```

```
-n
```

printf

```
$ printf '%s\n' foo bar baz
```

foo

bar

baz

pwd

현재 작업하는 위치를 확인 시 사용하는 명령어.

\$pwd

/home/bash/working

\$ ls secret/report.txt

\$ ls report_review.txt

\$ ls nonexistent

tildes(~)

경로 관련된 ~(틸드, tild)특수 문자. 이 문자를 통해서 홈 디렉터리로 이동이 가능하다. 틸드 path이외 다른 특수 문자 '.', '..'등 있지만, 보통 홈 디렉터리 이동 시 틸드를 더 선호 한다.

```
$ echo ~
```

```
$ echo ~/secret
```

```
# echo ~
```

```
# echo ~root/.ssh
```


tildes

만약 존재하지 않는 경우 그냥 문자열로 출력이 된다.

```
$ echo ~nouser
```

```
~nouser
```

tildes

특수 문자로 출력하고 싶은 경우 아래처럼 실행한다.

```
$ echo ~bash
```

```
/home/bash
```

```
$ echo \~bash '~bash' "~bash"
```

```
~bash ~bash ~bash
```

cd

cd는 change directory의 약자, 이를 통해서 디렉터리 이동을 한다.
이와 관련된 쉘 변수는 \$PWD 혹은 pwd명령어가 있다.

```
$ pwd
```

```
/home/bash
```

```
$ cd /tmp
```

```
$ pwd
```

```
/tmp
```

cd

앞에서 언급하였던 .. 혹은 .를 통해서 디렉터리 이동이 가능함.

```
$ cd .
```

```
$ cd ..
```

```
$ pwd
```

```
$ stat .
```

```
$ stat ..
```

pushd/popd

cd 명령어와 같이 **pushd, popd** 명령어로 다음처럼 사용이 가능하다.

```
$ sudo yum install httpd -y
```

```
$ pushd /var/www
```

```
$ pushd /opt/
```

```
$ push -n /usr/local/
```

pushd/popd

```
$ pushd -n /usr/local/
```

-n 옵션이 붙어 있으면 실제로 디렉터리 이동하지는 않음
내역을 확인하기 위해서는 다음과 같은 명령어로 확인 가능.

```
$ dirs -l -v
```

```
$ pushd +2
```

```
$ pushd -1. == $ cd -
```

pushd/popd

popd 명령어로 이전에 사용하였던 하나씩 이동하면서 스택(stack)에서 하나씩 제거한다.

```
$ pushd /usr/local
```

```
$ pushd /opt/
```

```
$ pushd /var/www
```

```
$ popd
```

set

'set' 명령어는 빌트인 명령어.

'set'은 현재 사용중인 셸에 설정되어 있는 값을 화면에 변수 이름과 함께 출력한다.

기존에 사용하던 값을 'set'명령어를 통해서 설정 및 변경도 가능하다.

알파벳 순서대로 정렬해서 화면에 출력한다.

여기에 출력된 변수들은 전부 셸에서 명시된 값이다.

set

옵션	설명	참고 주소
declare	Declare variables and give them attributes. If no names are given, then display the values of variables instead.	https://www.gnu.org/software/bash/manual/bash.html#Bash-Builtins
set	This builtin is so complicated that it deserves its own section. set allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables	https://www.gnu.org/software/bash/manual/bash.html#The-Set-Builtin

set

옵션	설명	참고 주소
env	environment variable in Bash	https://www.gnu.org/software/bash/manual/bash.html#Bash-Variables
export	Mark each name to be passed to child processes in the environment	https://www.gnu.org/software/bash/manual/bash.html#Bourne-Shell-Builtins

set

실제로는 '**set**'명령어보다 '**declare**'명령어 사용을 더 권장한다.

set명령어는 "-v"옵션을 통해서 설정된 값 확인이 가능하다.

확인 하는 방법은 **printenv**, **env**, **compgen** 같은 명령어 사용을 권장한다.

set

-e: 프로그램이 종료가 되면 0 상태 값을 반환한다.

하지만 이 방법을 스크립트에서 사용하는 경우 거의 대다수가 올바르게 값을 반환한다.

그 이유는 -e 옵션은 결과값 예측이 어렵기 때문이다.

그래서 대다수 스크립트는 `set -e`를 사용하지 않고 자체적으로 오류 값 핸들링을 더 선호한다.

set -e

```
$ nano set1.sh
```

```
#!/bin/bash
```

```
set -e
```

```
function hello()
```

```
{
```

```
    echo "I hate the Java language!!"
```

```
}
```

```
hello
```

set -e

https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html

\$ nano set2.sh

```
#!/bin/bash
function hello()
{
    set -e
    echo "I hate the Java language!!"
}
hello
```

set -e

```
$ nano set3.sh
```

```
#!/bin/bash
set -e
function Error()
{
    echo "error occur at line $1"
}
trap 'Error $LINENO' ERR

errorss
```

"-e"가 올바르게 동작하면 아래처럼 어느 라인에 문제가 있는지 메시지를 출력한다. 하지만, 해당 오류는 정확한 정보가 아닌 경우가 더 많다.

set -e

\$ sh set3.sh

set3.sh: line 9: errorss: command not found
error occur at line 9

set -n

-n

명령어를 읽어 오지만 실행하지 않는다.

\$ nano set4.sh

```
#!/bin/bash
```

```
set -n
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -u

-u

변수가 선언이 되었지만, 값을 가지고 있지 않는 경우 경고 메시지 출력.

```
$ nano set5.sh
```

```
#!/bin/bash
```

```
set -u
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -o

-o

"-u"하고 비슷하지만 조금 더 자세하게 출력이 된다.

```
$ nano set6.sh
```

```
#!/bin/bash
```

```
set -o nounset
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -f

-f

글러빙 끄기

\$ set -f

\$ ls *

\$ set -f

\$ ls *

set -a

-a

변수를 외부에서도 사용할 수 있도록 한다.

\$ set -a

\$ number1=23

\$ number2=34

set -v

-v

읽어오는 값을 그대로 화면에 출력한다.

디버깅시 매우 유용하지만, **-x**와는 다른 정보를 출력한다.

\$ bash -v set1.sh

set -x

-x

이 옵션은 각각 인자 값이 실행하는 걸 출력한다. 이 옵션을 실행하면 bash스크립트 디버깅에 도움이 된다.

어떤 부분에서 bash프로그램이 입력을 받아서 확장이 되는지 확인이 가능하다.

\$ bash -x set3.sh

set -x

디버깅시 다음과 같은 방법으로 진행이 가능하다.

set -x 를 CLI에 같이 선언 혹은 **매직-뱅(Magic-Bang)**에 다음과 같이 선언한다.

#!/bin/bash -x

또한 set의 종료 문자는 "--"으로 사용이 가능하며, 이는 위치 값(positional parameter)로 사용이 가능하다.

set -x

```
$ nano set7.sh
```

```
#!/bin/bash  
var = 5  
while [ $var -gt 0 ] ; do  
    var=$(( $var-1 )  
    echo $var  
    sleep 2  
done
```

"-x"를 선언 전. 일반적인 결과만 출력한다.

set

```
$ nano set8.sh
```

```
#!/bin/bash
```

```
set -x
```

```
var = 5
```

```
while [ $var -gt 0 ] ; do
```

```
    var=$(( $var-1 )
```

```
    echo $var
```

```
    sleep 2
```

```
done
```

set -x

혹은 다음처럼 실행하여도 된다.

bash -x <SCRIPT>

declare

declare는 내장 명령어이다.

이를 통해서 변수, 배열 혹은 함수를 구현한다.

변수 구현을 위해서 자주 사용하지는 않지만, 셸의 현재 상태 정보 및 다른 정보들 set를 통해서 훑어보기(overview)가 가능하다.

declare

\$ declare -p | head -5

```
declare -- BASH="/bin/sh"
```

\$ declare -r

```
BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:  
extglob:extquote:force_fignore:histappend:interactive_comments:login  
_shell:progcomp:promptvars:sourcepath"
```

declare -r

```
$ declare -r test=ok
```

```
$ echo $test
```

```
ok
```

```
$ test=not-ok
```

```
-bash: test: readonly variable
```

declare -i (integer)

```
$ declare -i x=10
```

```
$ echo $x
```

```
$ x=ok
```

```
$ echo $x
```

```
0
```

```
$ x=15
```

```
$ echo $x
```

```
15
```

```
$ x=15+5
```

```
$ echo $x
```

```
20
```

declare

이 명령어를 사용할 때 -f 옵션을 -p와 함께 사용하면 declare는 선언이 되어 있는 모든 함수에 대해서 출력한다.

-f: 쉘의 함수(function)만 출력

-p: 변수의 속성 출력

변수 속성을 제어 할때는 "+, -" 기호로 속성 제어가 가능.

unset

```
$ unset -f <function_name>
```

```
$ unset <value_name>
```

선언된 변수를 메모리에서 해제한다. 'alias'도 비슷한 명령어를 가지고 있다.

```
$ unalias <alias_name>
```

일반 시스템 명령어

\$ ls

adir bdir cdir

\$ ls -a

. .. .profile .bash_profile Downloads

일반 시스템 명령어(ls)

\$ ls -al

```
drwxrwxr-x. 4 tang tang 4096 May 24 12:19 kubectl-plugins
drwxrwxr-x. 10 tang tang 249 May 24 12:16 kubectx
drwxrwxr-x. 9 tang tang 251 May 24 12:15 kubelogin
drwxrwxr-x. 3 tang tang 80 May 24 12:26 kubergui
```

일반 시스템 명령어(ls)

만약 ' ls ' 명령어 없이 파일 이름 및 목록을 얻으려면 어떻게 해야 할까?

\$ ls -l

\$ grep pattern -- *

일반 시스템 명령어(ls)

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

```
$ find . -type f -exec grep <PATTERN> -- {} \;
```

```
$ for file in * ; do grep -F <PATTERN> -- "$file" ; done
```

일반 시스템 명령어(mv)

'mv' 명령어는 디렉터리 및 파일을 이동하거나 혹은 이름을 바꾸는 명령어.

```
$ mv file path/to/directory
```

```
$ mv file1 file2
```

```
$ mv file1 file2 dir4 path/to/directory
```

```
$ mv testdir/ goodir/
```

```
$ touch etevers.txt
```

```
$ mv etevers.txt /tmp
```

```
$ touch etevers.txt
```

```
$ mv --backup=t etevers.txt /tmp
```

mv

'mv' 명령어는 디렉터리 및 파일을 이동하거나 혹은 이름을 바꾸는 명령어.

```
$ mv file path/to/directory
```

```
$ mv file1 file2
```

```
$ mv file1 file2 dir4 path/to/directory
```

```
$ mv testdir/ goodir/
```

```
$ touch etevers.txt
```

```
$ mv etevers.txt /tmp
```

```
$ touch etevers.txt
```

```
$ mv --backup=t etevers.txt /tmp
```

일반 시스템 명령어(mv)

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

```
$ mv -i olduser/ newuser/
```


일반 시스템 명령어(mv)

```
$ nano mv1.sh
```

```
#!/bin/bash
```

```
if [[ -e old_user ]] ; then
```

```
    printf 'the old user exists.\n'
```

```
    mv old_user new_user
```

```
fi
```

```
mv new_user user
```

일반 시스템 명령어(cp)

cp 명령어는 파일이나 혹은 파일을 복사 시 사용한다. 이와 비슷한 명령어는 ln명령어가 있다. 하지만, ln명령어는 복사 하지는 않는다.

```
$ cp doc1 doc2 doc3 dir/
```

```
$ ls
```

```
dir
```

```
$ cp dir/ back_dir/
```

일반 시스템 명령어(cp)

```
$ cp -R
```

```
$ cp -r
```

```
$ cp -R back_dir/ rel_dir/
```

-r, -R의 차이점은 기본 옵션은 GNU에서는 -r를 사용한다. 하지만, 전통적인 유닉스에서는 -R옵션을 사용한다.

일반 시스템 명령어(rm/rmdir)

'rm', 'rmdir'은 비슷한 동작을 하지만 약간의 차이가 있다.

```
$ rm testfile
```

```
$ rm -rf testdir/
```

```
$ rmdir testdir/
```

```
Directory not empty
```

일반 시스템 명령어(grep)

'grep' 명령어는 특정 문자열을 표준 출력에서 검색 및 출력한다.

```
$ grep 'ssh' /etc/services
```

```
ssh      22/tcp      # The Secure Shell (SSH) Protocol
ssh      22/udp      # The Secure Shell (SSH) Protocol
x11-ssh-offset 6010/tcp    # SSH X11 forwarding offset
ssh      22/sctp    # SSH
sshell   614/tcp    # SSLshell
sshell   614/udp    #   SSLshell
```

일반 시스템 명령어(grep)

```
$ grep '^ssh' /etc/services
```

ssh	22/tcp	# The Secure Shell (SSH) Protocol
ssh	22/udp	# The Secure Shell (SSH) Protocol
ssh	22/sctp	# SSH
sshell	614/tcp	# SSLshell
sshell	614/udp	# SSLshell
ssh-mgmt	17235/tcp	# SSH Tectia Manager
ssh-mgmt	17235/udp	# SSH Tectia Manager

일반 시스템 명령어(grep)

```
$ grep -i 'ssh$' /etc/services
```

ssh	22/sctp	# SSH
netconf-ssh	830/tcp	# NETCONF over SSH
netconf-ssh	830/udp	# NETCONF over SSH
sdo-ssh	3897/tcp	# Simple Distributed Objects over SSH
sdo-ssh	3897/udp	# Simple Distributed Objects over SSH
tl1-ssh	6252/tcp	# TL1 over SSH

일반 시스템 명령어(grep)

중복된 출력 결과를 다음과 같은 명령어로 정리가 가능하다.

```
$ uniq
```

```
$ sort
```

혹은 'grep'에서 '-q'옵션을 사용하여 조건에 맞는 문자열이 있는지 확인이 가능하다.

```
$ grep -q telnet /etc/service && echo 'Matched'
```

```
$ echo $?
```

```
$ grep -q foobar /etc/service && echo 'Matched'
```

```
$ echo $?
```


일반 시스템 명령어(grep)

혹은 몇 개의 라인이 일치하는지 확인 하려면 다음과 같은 방법이 있다.

```
$ grep -c telnet /etc/service
```

5

```
$ grep telnet /etc/service | wc -l
```

복합적으로 문자열을 확인이 필요한 경우는 다음과 같은 명령어로 확인이 가능하다.

```
$ grep -e ssh -e telnet /etc/services
```

일반 시스템 명령어(grep)

'-F' 옵션을 통해서 문자열 검색이 가능하다.

매뉴얼 설명은 다음처럼 나와있다.

Interpret PATTERNS as fixed strings, not regular expressions.

```
$ grep -F '$' ikea
```

만약 매치가 되지 않는 문자열만 출력하고 싶은 경우 -v 문자열을 사용한다.

'^#' 옵션은 주석 문자인 #를 제외하고 화면에 출력.

```
$ grep -v '^#' /etc/services
```

일반 시스템 명령어(grep)

조금 더 복잡하게 특정 문자만 카운팅이 필요한 경우 다음과 같이 실행한다

```
$ grep -cFv -e bash -e '.' /etc/shells
```

10

-c: 카운팅

-F: 정규식 기호 문자열 취급

-v: 예외

-e: 복수 조건(정규식)

일반 시스템 명령어(cut)

'cut'명령어는 말 그대로 자르는 명령어.

특정 필드나 문자열을 자르고 싶은 경우 cut명령어를 통해서 사용이 가능하다.

자주 사용하는 옵션은 보통 '-d', '-f'이며, '-d'는 필드 분리 문자 '-f'는 필드이다.

일반 시스템 명령어(cut)

```
$ cut -d: -f1 /etc/passwd
```

root

bin

daemon

adm

lp

sync

shutdown

일반 시스템 명령어(cut)

만약, 맨 끝 필드를 출력하고 싶은 경우 아래처럼 실행한다.

```
$ cut -d: -f6- /etc/passwd
```

```
/root:/bin/bash
```

```
/bin:/sbin/nologin
```

```
/sbin:/sbin/nologin
```

일반 시스템 명령어(wc)

단어 혹은 줄 개수 그리고 바이트를 출력한다. 기본 값은 전부를 출력한다.

```
$ wc .bash_history
```

```
200 437 3373 .bash_history
```

위의 출력된 내용은 200줄, 437줄 그리고 총 3373바이트.
여기에서 특정 값만 출력을 원하는 경우 '-c', '-w', '-l'같은 옵션을 사용해서 출력이 가능하다.

'-l'옵션은 엄연히 따지고 보면 newline개수만 출력한다.

일반 시스템 명령어(wc)

-c 옵션 경우에는 바이트를 출력한다.

```
$ wc -c .bash_history
```

```
3373 .bash_history
```


일반 시스템 명령어(wc)

-m 총 문자 개수를 카운팅 하기 위해서는 -m 옵션을 사용한다.

```
$ wc -m .bash_history
```

```
3373 .bash_history
```

일반 시스템 명령어(wc)

```
$ wc -m -c Korean
```

```
35 93 test
```

한국어를 UTF-8로 입력하면 영문과 다르게 바이트 및 문자가 카운트가 된다. 그 이유는 인코딩 및 2바이트 문자열로 인하여 개수와 크기가 다르게 출력이 된다.

일반 시스템 명령어(du)

'wc'명령어로 파일의 크기를 알 수 있다.

```
$ wc -c /bin/bash
```

하지만 전체적인 크기에 대해서는 확인이 불가능 하기 때문에 du명령어를 사용하요 확인한다.

```
$ du -k /bin/bash
```

```
1124  /bin/bash
```

디렉터리 크기를 알고 싶은 경우 아래처럼 실행한다. 모든 결과는 **POSIX 1kibibyte(1024)**로 출력이 된다.

```
$ du -k /bin
```

find

'find' 명령어는 리눅스에서 많이 사용하는 명령어 중 하나이다.
이 명령어는 디렉터리 및 파일 검색이 가능하다.

기본적인 동작 방식은 다음과 같다.

```
$ find ~/
```

출력만 하고 싶은 경우 다음처럼 옵션을 지정한다.

```
$ find /usr/share -print
```

find

여러 위치에서 검색이 필요한 경우 다음처럼 사용이 가능하다. 아래 명령어는 "email.txt"파일에서 "**docs**", "**googles**"에서 찾는 명령어이다.

```
$ find ~/docs ~/googles email.txt
```

출력된 내용을 정렬하기 위해서 다음과 같이 혼용해서 사용이 가능하다. 아래 명령어는 결과를 sort명령어로 정렬한다.

```
$ find ~/docs | sort
```

find

test라는 문자열을 가지고 있는 파일 및 디렉터리

```
$ find ~/docs -name '*test*' -print
```

fish라는 문자열 가지고 있는 파일만 출력

```
$ find ~/docs -name '*fish*' -type f -print
```

수정하지 3일이 지난 파일

```
$ find ~/docs -mtime +3 -print
```

5일전에 수정한 파일

```
$ find ~/docs -mtime -5 -print
```

find

이름에 beef라는 문자열이 없는 것만 검색

```
$ find ~/docs ! -name '*beef*'
```

만약 특정 문자열에 대해서 무시를 하고 싶은 경우, 아래처럼 명령어를 실행한다.

```
$ find . -name '.git*' -prune -o -print
```

find

find를 사용하면, 각각 결과마다 명령어를 같이 실행이 가능하다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- {} \;
```

-exec: 명령어 실행, 여기서는 'grep -F search --' 명령어

{}: 이 부분에 find에 찾을 출력 결과물로 대체

\;: 명령어 종료 알림

--: 인자 값으로 전달 되는 값들을 옵션으로 처리 하지 않는다.(맨 페이지에서는 시작점(start point를 ./ 혹은 /으로 처리 권장)

find

위 명령어를 나누면 다음과 같이 두개 구조로 나누어진다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- '{}' \;
```

```
$ find /usr/share/doc -type f -name '*.vim'
```

```
$ grep /usr/share/doc/vim/map.vim  
$ grep /usr/share/doc/vim/ansible.vim
```

인자 끝
(명령어)

명령어 보호

```
$ grep -F search -- vim/map.vim  
$ grep -F search -- vim/test.vim
```

find

만약 'find'의 '-exec'에 더 많은 인자 값이 필요한 경우 +기호를 사용하여 좀 더 많은 인자 값 옵션 사용이 가능하다.

```
$ find / -type f -name '*.vim' -exec grep -F search -- '{}' +
```

```
$ grep -F search -- vim/map.vim vim/test.vim
```

첫번째 결과

두번째 결과

연속처리

find -exec +

"+"와 ";" 기능을 비교하면 다음과 같다. 명령어에 전달할 인자 값이 없는 경우 "--"를 사용하지 않아도 된다. 아래 예제는 'echo'를 사용하기에 "--"사용하지 않았다.

```
$ find /usr/share/doc -name "*.md" -type f -exec echo {} +
```

위의 명령어로 실행하는 경우 줄 매번 echo로 출력해주는 내용이라서 newline없이 화면에 출력이 된다.

```
$ find /usr/share/doc -name "*.md" -exec echo {} \;
```

위의 명령어는 이와 반대로 "newline"이 포함이 된 상태로 화면에 출력이 된다.

find with cp

'find' 명령어로 파일 복사 시, 다음과 같이 활용한다.

```
# mkdir -p /tmp/md_files/  
# find /usr/share/doc -name "*.md" \  
> -type f -print -exec cp '{}' /tmp/md_files \;
```

xargs

이 명령어는 find -exec와 비슷하지만 조금 다른 동작 구조를 가지고 있다.


데이터 스트림

```
$ find /usr/share/doc -type f -name '*.md' | xargs grep -F search --
```

위의 명령어는 find에서 실행한 결과를 오른쪽 'xargs'명령어 전달, 'grep' 명령어가 작업을 수행한다.

xargs

만약, 출력되는 결과에 null-byte로 종료되는 경우가 있다.
올바르게 동작이 안될 수 있기 때문에 find에서 xargs를 사용시 -print0이라는 옵션을 사용한다.

'find -print0' 혹은 'xargs -0'

```
$ find /usr/share -type f -name '*.vim' -print0 | xargs -0 grep -F search  
--
```

xargs

만약, 출력되는 결과에 null-byte로 종료되는 경우가 있다.
올바르게 동작이 안될 수 있기 때문에 find에서 xargs를 사용시 -print0이라는 옵션을 사용한다.


'find -print0' 혹은 'xargs -0'

```
$ find /usr/share -type f -name '*.vim' -print0 | xargs -0 grep -F search  
--
```

find + cp + xargs

```
$ find / -type f -name LICENSE -exec cp {} -- /tmp/LICENSE-$(od -An -N2 -i /dev/random | grep -v ^$ | tr -d [:blank:]) \;
```

```
$ find / -type f -name LICENSE | while IFS= read -r f; do cp -v "$f" "/tmp/${f//\//_}"; done
```



표준 출력으로 전달 받은 입력 내용중 디렉터리 파싱

find+

추가적으로 'find'의 -exec 및 '{}' '+'에 대해서 추가적인 설명.

앞에서 말한 옵션 "-exec {} \", "-exec {} +", "| xargs" 총 3가지로 사용이 가능하다.

기본적으로 'find'에서 검색된 내용은 "-exec"에서 이벤트가 발생한다. 만약 삭제라고 한다면 'find'에서 찾은 내용은 'rm'명령어가 삭제를 한다.

find+

1. `$ find . -name "*.txt" -exec rm {} \;`
2. `$ find . -name "*.txt" -exec rm {} +`
3. `$ find . -name "*.txt" | xargs rm`

find+

1번 경우에는 실행이 되는 1:1 명령어로 수행이 된다. 예를 들어서 다음과 같이 동작한다.

삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} \; → rm -f test1.txt
```

```
$ find . -name "*.txt" -exec rm {} \; → rm -f test2.txt
```

find+

2번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다.

삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} + → rm -f test1.txt test2.txt
```

find+

3번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다.

삭제 대상 파일이 test1.txt test2.txt파일 이라고 하면 다음처럼 동작한다.

```
$ find . -name "*.txt" -exec rm {} + ➔ rm -f test1.txt test2.txt
```

하지만 3번 경우에는 문제가 있다.

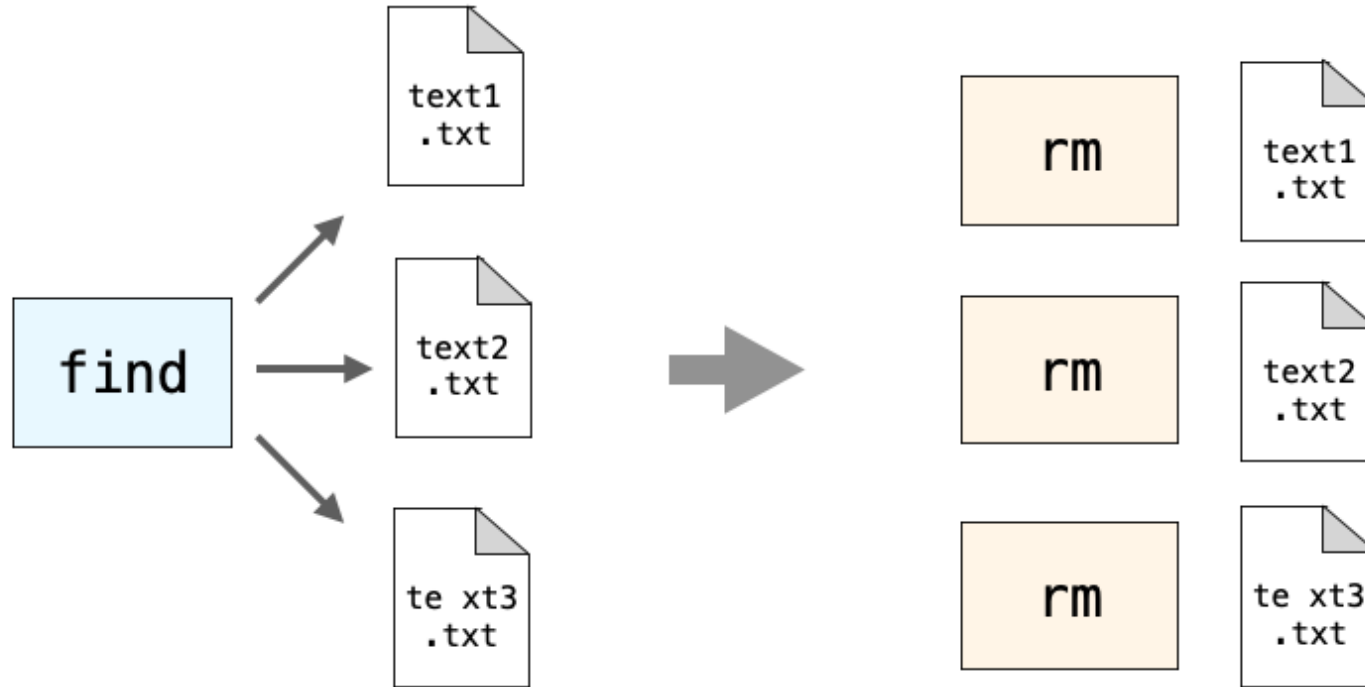
find+

3번은 "**null string**"이 들어오는 경우 올바르게 처리는 하지 못하는 문제가 있다.

여기서 말하는 "null string"은 말 그래도 **비어 있는 문자**이다. 예를 들어서 파일명이 "**test1.txt**", "**test2.txt**", "**test document.txt**"되어 있는 경우 "test document.txt"는 **비어 있는 공간**이 있기 때문에 다른 파일로 인식이 된다.

```
$ rm -f test1.txt test2.txt test document.txt
```

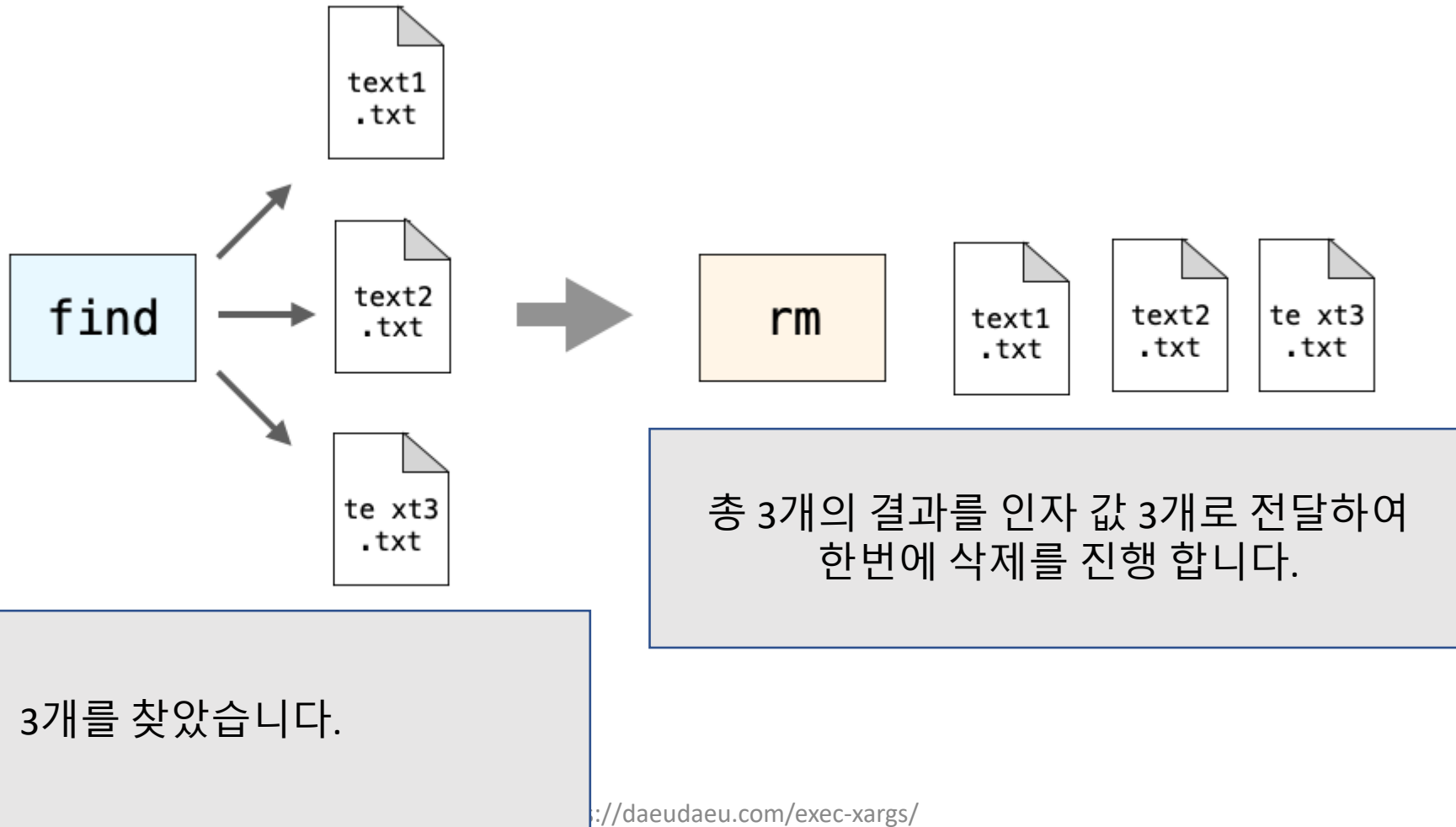
find+



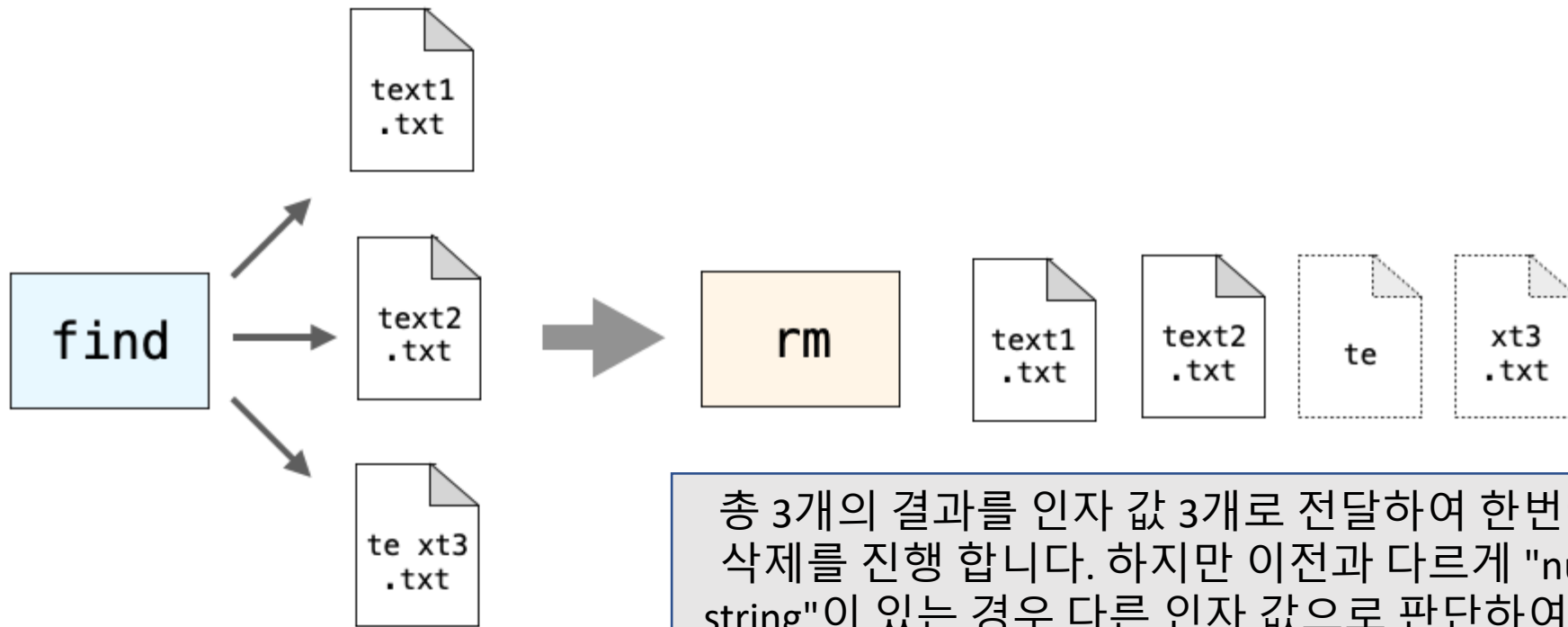
3개를 찾았습니다.

`find`에서 대상을 찾을 때마다 1번씩 명령어 수행, 총 3번 실행을 합니다.

find+



find+



3개를 찾았습니다.

총 3개의 결과를 인자 값 3개로 전달하여 한번에 삭제를 진행 합니다. 하지만 이전과 다르게 "null string"이 있는 경우 다른 인자 값으로 판단하여 총 4개가 전달이 됩니다.

find+

```
./test1.txt\0./te st2.txt\0
```



\0

위의 test1.txt\0 "null string"이 존재 합니다.

引数 1 : ./test1.txt

引数 2 : ./te st2.txt

기본값: find / -name test1.txt -type f -print
널 스트링: find / -name test2.txt -type f -print 0

빈 공간이 있는 경우, 이 부분을 문자열로 취급하여
올바르게 인식되게 합니다.

find+



-print0 옵션을 사용하기 전

find



./test1.txt
./test2.txt



-print0 옵션을 사용하는 경우, 빈 공간이 있으면 이 부분 "null string"으로 처리한다.

find



./test1.txt\0./test2.txt\0

Null String vs Empty String

positive value



1



0



negative value



Infinity



NaN



null



undefined



sort

'sort'는 출력된 결과를 알파벳 순서로 정렬 시 사용하는 문자이다.

```
$ sort /etc/shells
```

```
$ sort ~/file1 ~/file2
```

```
$ printf '%s\n' 'line 2' 'line 1' | sort
```

sort

또한 같이 사용하는 옵션은 다음과 같다.

- n: 숫자가 알파벳 순서보다 먼저 출력이 된다.
- r: 역 정렬, 맨 마지막 결과가 맨 처음에 출력

uniq

```
$ cat ipaddress.txt
```

```
$ sort ipaddress.txt | uniq -c | sort -k1, 1nr
```

Input, Output, Redirection

```
$ printf "Hello World!\n"
```

Hello World!

```
$ printf "Hello World!\n" > myfile
```

```
$ cat myfile
```


Input, Output, Redirection

```
$ printf '' > empty.txt
```

```
$ wc -c empty.txt
```

```
0
```

디렉터리 위치로 재전달 혹은 출력하는 경우 다음처럼 한다.

```
$ cd
```

```
$ printf 'Hello World!\n' > ~/tmp/test.txt
```

```
$ printf 'Hello World!\n' > /tmp/test.txt
```

Input, Output, Redirection

```
$ printf 'Hello World!\n' > "$HOME"/text.txt
```

```
$ printf 'Hello, root! \n' > ~root/tmp/text.txt
```

덮어쓰기 금지

bash에서 덮어쓰기를 방지하고 싶은 경우 -c 옵션을 통해서 가능하다.

```
$ print 'First Command' > /tmp/test.txt
```

```
$ print 'Second Command' > /tmp/test.txt
```

```
$ cat test.txt
```

```
Second Command
```

덮어쓰기 금지

```
$ set -C
```

다음처럼 다시 시도한다.

```
$ printf 'Third Command\n' > /tmp/test.txt
```

덮어쓰기 금지

덮어쓰기 금지 상태에서 강제로 덮어쓰기를 하려면 다음처럼 명령어를 실행한다.

```
$ set -C
```

```
$ printf 'Thrid Command\n' >| /tmp/test.txt
```

```
$ echo $?
```

```
$ set +C          ## set -C 해제
```

이어 붙이기

기존 텍스트 내용에 추가를 하거나 혹은 이어 붙이기 하는 경우
다음처럼 명령어를 실행한다.

```
$ printf 'First Command\n' > /tmp/test_doc.txt
```

```
$ printf 'Second Command\n' >> /tmp/test_doc.txt
```

```
$ cat test_doc.txt
```

First Command

Second Command

이어 붙이기

스크립트 기반으로 이어 붙이기.

\$ vi append.sh

```
#!/bin/bash
printf 'Starting script\n' >> log
printf 'Creating test directory\n' >> log
mkdir test || exit
printf 'Changing into test directory\n' >> log
cd test || exit
printf 'Writing current date\n' >> log
date > date || exit
```

이어 붙이기

```
$ mkdir test || exit
```

"||" 만약 해당 디렉터리가 이미 생성이 되어 있는 경우
exit 명령어를 실행한다.

"||" 연산자는 or 역할을 한다.

퍼미션

여기에서는 chmod, chown에 대해서는 자세하게 다루지 않는다.

다만, 쉘 스크립트에서 소유권 및 퍼미션을 어떻게 핸들링 하는지 학습 한다.

이 부분은 간단하게 메모장으로 설명 했습니다.
깃헙의 메모 확인 부탁드립니다. 😊

퍼미션

'stat', 'stat'명령어를 통해서 0~7사이의 숫자 8진수를 통해서 퍼미션 확인이 가능하다.

```
$ stat /tmp/test.txt
```

'stat'명령어를 통해서 퍼미션 확인을 원하는 경우 아래와 같은 명령어로 확인 가능하다.

```
$ stat -c %a /tmp/test.txt
```

일반 시스템 명령어(cut)

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다.

```
$ cut -d: -f1,6 /etc/passwd
```

```
root:/root
```

```
bin:/bin
```

```
daemon:/sbin
```

```
adm:/var/adm
```

```
lp:/var/spool/lpd
```

```
sync:/sbin
```

입출력 전환

표준 입출력 오류는 각각 숫자를 가지고 있다.

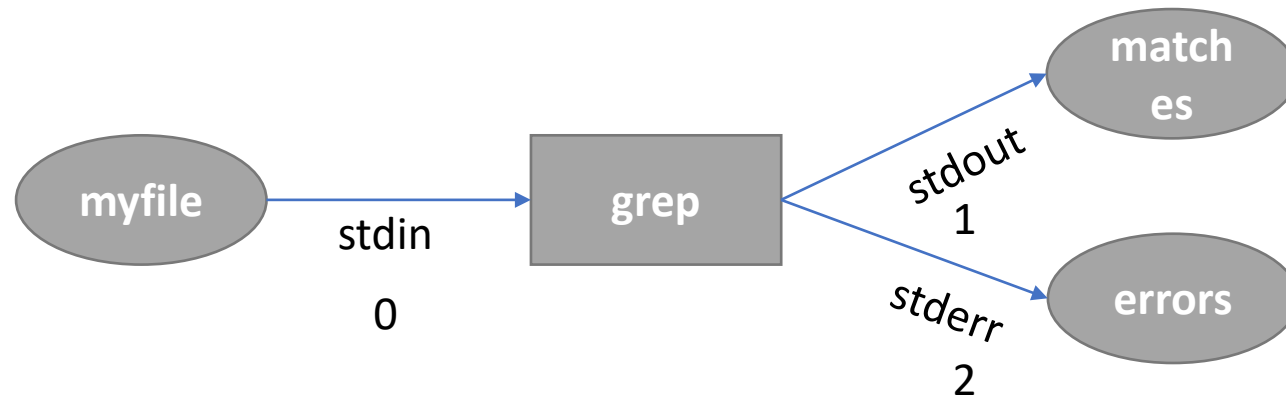
bash에서는 "**stdout(standard output)**"이라고 부르며 이걸 “**표준출력**”이라고 말한다.

오류는 "**stderr(standard error)**"그리고 표준 오류라고 부른다.

이 옵션은 매우 bash에서 스크립트에서 매우 일반적으로 사용한다.

다른 호칭으로 **파일-디스크립터(File Descriptor)**라고 부른다. 디스크립터는 **총 3개**를 가지고 있다.

입출력 전환



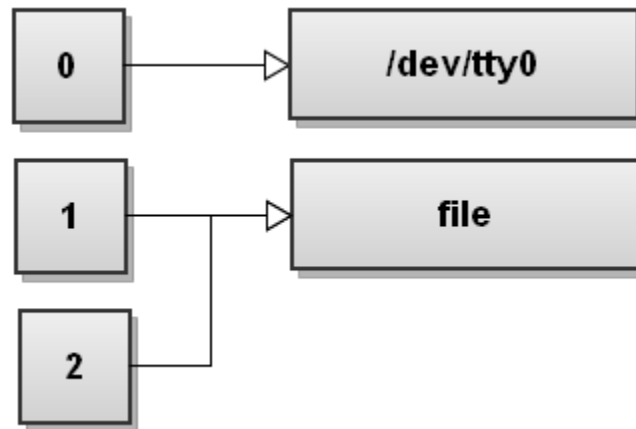
입출력 전환

"0" 표준 출력 및 오류에 대한 동시 출력, 번호로는 0번 사용.

"1" 표준출력, 오류가 정상 메시지를 화면에 출력한다.

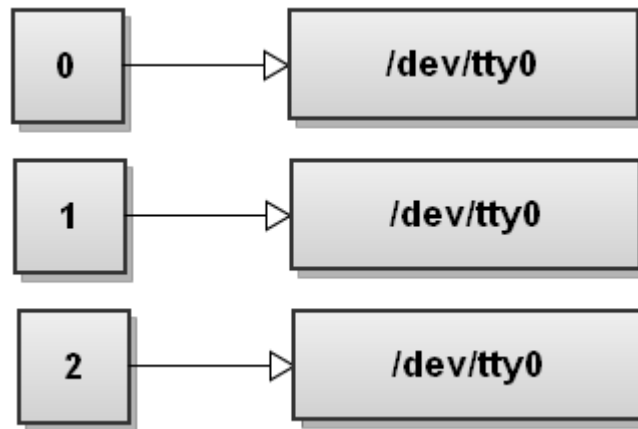
"2" 표준오류, 오류 부분에 대해서만 화면에 출력한다.

입출력 전환



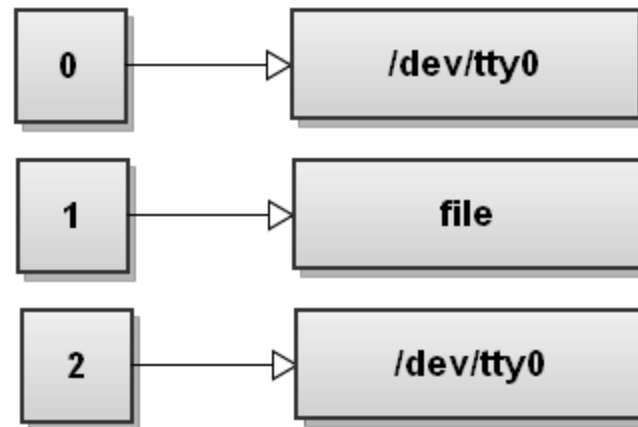
\$ ls >file 2>&1

입출력 전환



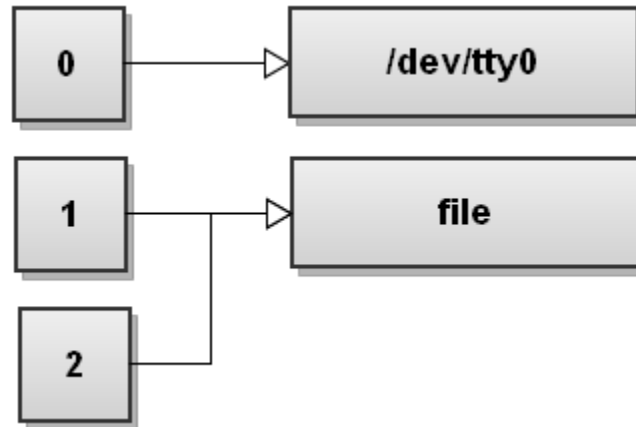
\$ ls

입출력 전환



`$ ls > file > 1`

입출력 전환



\$ ls > file 2>&1

입출력 전환

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

```
$ grep pattern testfile.txt /thereisnowhere > matches 2> matches_error.txt
```

위에 처럼 사용하는 경우 현재 오류 메시지만 matches_error.txt파일에 기록 정상적인 메시지는 matches에 저장되어 있다.

그래서 같이 출력을 하기 위해서는 **&(ampersand)**를 통해서 1에 대한 출력을 명시 해주어야 한다.

입출력 전환

```
$ grep pattern testfile.txt /thereisnowhere > matches 2>&1
```

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다.

주의 할 부분은 동일한 작업을 수행하기 위해서는 >앞에다가 2>&1을 넣을 수 없다.

리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

블록 오류 및 처리

가끔 발생한 모든 오류를 화면에 출력할 필요는 없다.
그럴 때 사용하는 유용한 장치가 있다. 바로....

`/dev/null`

위의 장치로 오류 내용을 내보내면 된다.
전달하는 방법은 매우 간단하다.

`$ grep pattern testfile /thereisnowhere > matches 2> /dev/null`

블록 오류 및 처리

블록 오류를 처리하기 위해서 comm명령어를 사용하여 테스트한다. comm명령어는 두 개 파일의 내용은 줄 별로 비교 확인한다.

```
$ comm test1.txt test2.txt
```

```
$ comm test1.txt test2.txt 2> /dev/null
```

```
$ echo $?
```

여러 위치로 재 전달

여러 파일로 내용을 재 전달 하기 위해서는 tee를 사용하여 다음과 같이 처리한다.

하지만 꼭 tee명령어를 사용할 필요는 없다.

```
$ printf 'Copy Copy rom rom\n' | tee my_copy_rom.txt
```

```
Copy Copy rom rom
```

여러 위치로 재 전달

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

```
$ printf 'Copy Copy rom rom\n' | tee myfile1 myfile2 myfile3
```

```
Copy Copy rom rom
```

```
$ ls
```

```
myfile1 myfile2 myfile3
```


입력 재 전달

리눅스 명령어 중에 입력 재전달의 대표적인 명령어가 있다.
tr명령어는 입력 받은 내용을 대문자 혹은 소문자로 다시 출력한다.

```
$ tr a-z A-Z
```

```
Hello, world!
```

```
HELLO, WORLD!
```

입력 재 전달

위의 명령어를 가지고 재 전달 프로세스를 다루면 다음처럼 다를 수 있다.

```
$ cat mysample | tr a-z A-Z
```

혹은 미리 작성된 파일을 대소문자로 변경하는 경우에는 다음과 같은 명령어로 실행이 가능하다.

```
$ tr a-z A-Z < mysample
```

입력 재 전달

재 전달 오퍼레이터를 손쉽게 다음처럼 사용이 가능하다. 출력을 위한 cat같은 명령어를 사용할 필요가 없다.

```
$ < mysample tr a-z A-Z
```

혹은 아래처럼 복잡하게 입출력을 제어할 수 있다. 아래 명령어는 표준오류만 출력하는 예제이다.

```
$ tr a-z A-Z < mylines > mylines.capitalized 2> mylines.error
```

문자열

bash

긴 문자열 처리

\$ longstring1.sh

```
#!/bin/bash
```

```
case $1 in
```

```
    -h | --help )
```

```
        cat << 'EOF'
```

```
-h, --help: show this help
```

```
-q, --quiet: run without diagnostics
```

```
-v, --verbose: add extra diagnostics
```

```
EOF
```

```
        exit 0
```

```
    ;;
```

```
esac
```

긴 문자열 처리

긴 문자열 처리가 필요 할 때는 연속적으로 받기 위해서 입력 재출력을 통해서 "<<" 추가적으로 계속 받는다.

이때 어느 부분이 파일의 끝 부분인지 표시하기 위해서 'EOF'라는 키워드를 사용한다.

"cat << 'EOF'" 명령어는 입력라인에 EOF라는 문자열이 들어오면 더 이상 입력을 중지한다.

긴 문자열 처리

```
$ longstring2.sh
```

```
#!/bin/bash
```

```
cat << EOF
```

```
Hello, $USER; you are running this program on $(hostname -s).
```

```
Your home directory is $HOME
```

```
EOF
```

긴 문자열 처리

```
$ longstring3.sh
```

```
#!/bin/bash
```

```
cat <<- 'EOF'
```

```
    leading tabs
```

```
        will not be included in the output
```

```
EOF
```


긴 문자열 처리

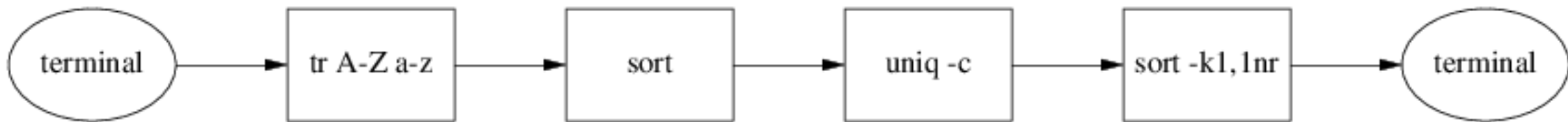
여러가지 명령어를 동시에 사용하기 위해서 쉘 스크립트 형식으로 명령어를 작성한다.

이를 동시에 처리하기 위해서 파이프를 사용한다.
예제로 tr명령어 기반으로 간단하게 구성을 하였다.

긴 문자열 처리

위와 같이 사용하는 경우 쉘 스크립트처럼 여러 명령어를 사용하기 때문에 다음처럼 파이프로 구성하여 동시에 처리 하도록 한다.

```
$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr
```



긴 문자열 처리

```
$ tr A-Z a-z | sort | uniq -c | sort -k1,1nr
```

이것저것 입력 후 ctrl+d 누르면 아래와 같이 내용이 출력이 된다.

"tr A-Z a-z" 대문자를 소문자로 변경한다. 그런 이후 "sort" 명령어로 내림차순으로 정렬하고 "uniq -c"로 카운트한다.

```
sdlkjskljieurwjldklskdj  
akljsnkcjnnjhdsiakjslia  
38473904889orjksdjm,snbns  
1 ($&$  
1 2948  
1 38473904889orjksdjm,snbns
```

긴 문자열 처리

이걸 쉘 스크립트로 변경하면 다음처럼 변경한다.

```
$ longstring4.sh
```

```
#!/bin/bash
```

```
tr A-Z a-z |
```

```
sort |
```

```
uniq -c |
```

```
sort -k1,1nr
```

Multi Pipe

여러 개의 텍스트 파일을 하나로 통합할 때 다음처럼 스트리밍 처리한다.

```
$ cat testfile1 testfile2 testfile3 > testfiles.combined
```

```
$ date > output
```

```
$ hostname >> output
```

```
$ cat output
```

Multi Pipe

그룹 명령어는 중괄호를 사용하여 처리한다. 아래처럼 실행하면 모든 출력이 대문자로 출력이 된다.

```
$ { date ; hostname ; } | tr a-z A-Z
```

Multi Pipe

```
$ { date ; printf '%s\n' apple carrot banana ; } > testwords
```

```
$ { read date ; printf '%s\n' "$date" ; sort ; } < testwords
```

stream editor

에디터가 아닌 표준 입력을 통해서 편집하는 방법. 보통 스크립트 처리시 아래와 같은 방법으로 작업한다.

```
$ cat /usr/share/doc/ls/LICENSE
```

```
$ sed '' /usr/share/doc/ls/LICENSE
```

```
$ sed '1,2d' /usr/share/doc/ls/LICENSE
```

```
$ sed '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/ls/LICENSE
```

```
$ sed 's/found/not_found/' /usr/share/doc/ls/LICENSE
```


자르기

특정 포매팅 형식으로 출력하기 위해서 아래와 같이 양식을 사용한다.

```
$ ta='type:json'
```

```
$ print 'Type: %s\n' ${ta#type:}'
```

```
Type: json
```

아래도 비슷한 형식이지만, 변수가 할당이 되어 있는 변수 이름을 출력한다.

```
$ ta='type:json'
```

```
$ printf 'Field name: %s\n' "${ta%:*}"
```

```
Field name: type
```

자르기

```
$ text='*WARNING'
```

```
$ printf '%s\n' "${text#\*}"
```

```
WARNING
```

```
$ printf '%s\n' "${text#'*}'
```

```
$ path=/usr/local/test/bin/test
```

```
$ printf 'Removed the path as look like a file : %s\n' "${path##*/}"
```

```
Removed the path as look like a file: test
```

문자열 자르기

문자열 처리 시, cut같은 명령어 사용이 가능하다.
조금 더 편하게 **`${var:<position>:<length>}`**사용이 가능하다.

```
$ title==' Password =='
```

```
$ printf '%s\n' "${title:3}"
```

```
Password ==
```

```
$ printf '%s\n' "${title:3:8}"
```

```
Password
```

```
$ alpha='abcdefghijklmn'
```

```
$ printf '%s\n' "${alpha: -3:2}"
```

```
ij
```

길이측정

문자열 길이는 '**wc**'같은 명령어로 확인이 가능하다. 쉘 빌트인 사용하는 경우 좀 더 쉽게 가능하다.

%u: 변수에 있는 문자열 길이를 확인 합니다.

```
$ alpha='abcdefghijklmn'
```

```
$ printf '%u\n' "${#alpha}"
```

11

문자열 바꾸기

```
$ notice='I\'\"I do it today.'
```

```
$ printf '%s\n' "${notice /today/tomorrow}'
```

```
$ notice='Yes, today. I\'\"I do it today.'
```

```
$ printf '%s\n' "${notice/today/tomorrow}"
```

Yes, tomorrow, I'll do it today.

```
$ printf '%s\n' "${notice//today/tomorrow}"
```

Yes, tomorrow, I'll do it tomorrow.

대소문자 변경

```
$ text='helloWorld'  
$ printf '%s\n' "${text^^}"  
HELLOWORLD
```

```
$ printf '%s\n' "${text^}"  
HelloWorld
```

```
$ printf '%s\n' "${text,}"  
helloWorld
```

글로브 혹은 글로빙

bash

글로브(*) 사용하기

글로브, **별표** 혹은 **눈**이라고 말하는 문자이다.

셸에서는 글로브 즉 별표를 사용하면 기본적으로 **모든 것(all of match)**라고 표현한다.

매치가 되는 자원은 거의 모든 자원에 적용이 된다.

```
$ ls -a
```

```
.  ..  Documents  Pictures  Downloads
```

```
$ printf '%s\n' *
```

```
hello
```

```
world
```

```
Documents
```

```
Downloads
```


글로브(*) 사용하기

```
$ printf '%s\n' ???uments
```

Documents

```
$ printf '%s\n' *[ts]
```

test1.txt

test2.txt

글로브(*) 사용하기

```
$ printf '%s\n' ???uments
```

Documents

```
$ printf '%s\n' *[^ts]
```

jsues.txt

sjwue.txt

글로브(*) 사용하기

```
$ cat -- c*
```

```
cat: 'c*' : No such file or directory
```

```
$ shopt -s nullglob
```

```
$ printf '%s\n' c*
```

'shopt' 명령어는 bash의 빌트인 명령어이며, 내부 함수를 설정한다.
위의 명령어는 'nullglob'를 활성화 한다.

글로브(*) 사용하기

```
$ shopt -s nocaseglob
```

```
$ printf '%s\n' A*
```

```
apple
```

```
Aux
```

글로빙 확장하기

bash는 부족한 글로빙 기능을 확장하기 위해서 **정규식(regular express)**기반으로 **고급기능(혹은 더 복잡하게)** 사용이 가능하다.

?(pattern): 하나라도 패턴에 맞으면 출력

+(pattern): 최소 한 개로도 맞으면 출력

***(pattern):** 숫자 하나라도 맞으면 출력

@(pattern): 명시한 조건만 맞으면 출력

!(pattern): 조건에 맞지 않는 것 만 출력

글로빙 활용하기

예를 들어서 자주 사용하는 `"/var/log"`를 예를 들면 다음처럼 사용한다.

`/var/log`에서 `.gz`로 끝나지 않는것만 출력

```
$ ls -l /var/log/!(*.log)
```

`haw`혹은 `mess`단어가 맞으면 화면에 출력

```
$ ls -l /var/log/@(haw|mess)*
```

배열

bash

배열 사용하기

셸에서 배열을 사용하기 위해서는 **경계기호(delimiter)**를 사용해서 변수와 배열을 구별한다.

```
$ devices=('iphone', 'android', 'winphone')
```

위의 배열은 매우 기본적인 배열 선언이다.

배열 사용하기

```
$ printf '%s\n' "${devices[0]}"
```

iphone

```
$ printf '%s\n' "${divces[2]}"
```

winphone

배열 사용하기

```
$ printf '%s\n' "${devices[-1]}"
```

winphone

```
$ printf '%s\n' "${devices[@]}"
```

iphone

android

winphone

배열 사용하기

```
$ printf '%s\n' "${devices[@]^}"
```

IPHONE

ANDROID

WINPHONE

배열 사용하기

```
$ homefiles=("$HOME"/*)
$ printf '%s\n' "${homefiles[@]}"
/home/tang/1.txt
/home/tang/2.txt
/home/tang/3.txt
/home/tang/4.txt
```

배열 사용하기

bash4.0 이후에 새로 할당된 배열 할당이 있다.

파이썬의 사전(dictionary)처럼 기능을 제공한다.

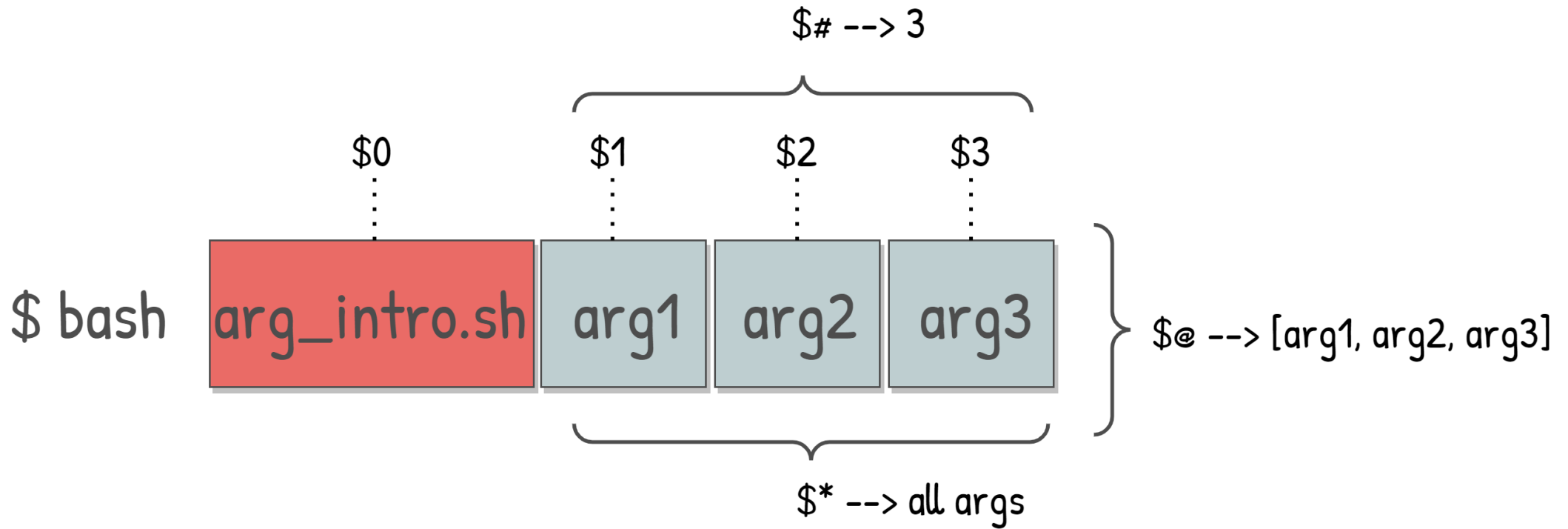
```
$ declare -A colors
```

```
$ colors=([tomato]="red" [orange]="yellow" [carrot]="orange")
```

```
$ printf '%s\n' "${colors[orange]}"
```

```
yellow
```

셸 입력 인자 값 처리



셸 입력 인자 값 처리

```
#!/bin/bash
```

```
echo "Script name is: $0"
```

```
echo "Arg1 is $1"
```

```
echo "Arg1 is $2"
```

```
echo "Arg1 is $3"
```

```
echo "-----"
```

```
echo "All args: $*"
```

```
echo "All args count: $#"
```

셸 입력 인자 값 처리

```
bash arg_intro.sh runtime inputs
```

```
Script name is: ./arg_intro.sh
```

```
Arg1 is runtime
```

```
Arg1 is inputs
```

```
Arg1 is
```

```
-----
```

```
All args: runtime inputs
```

```
All args count: 2
```


네트워크 관리 명령어

IP utility and nftables

NetworkManager vs network script

기존에 사용하는 network는 **init**기반의 "**shell scrip(/etc/init.d/)**"로 되어 있었다.

현재는 RHEL 7이후로는 "**NetworkManager**"기반으로 바뀌면서 "network script"를 더 이상 network스크립트로 관리하지 않고 NetworkManager에서 스크립트 생성/수정 및 제거를 관리한다.

NetworkManager.service: 서비스 이름. 'systemctl'로 확인이 가능

관리 명령어: nmtui, nm-connection-editor, nmcli

/etc/sysconfig/network-scripts/: 스크립트 저장되는 위치

네트워크 관리 명령어

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

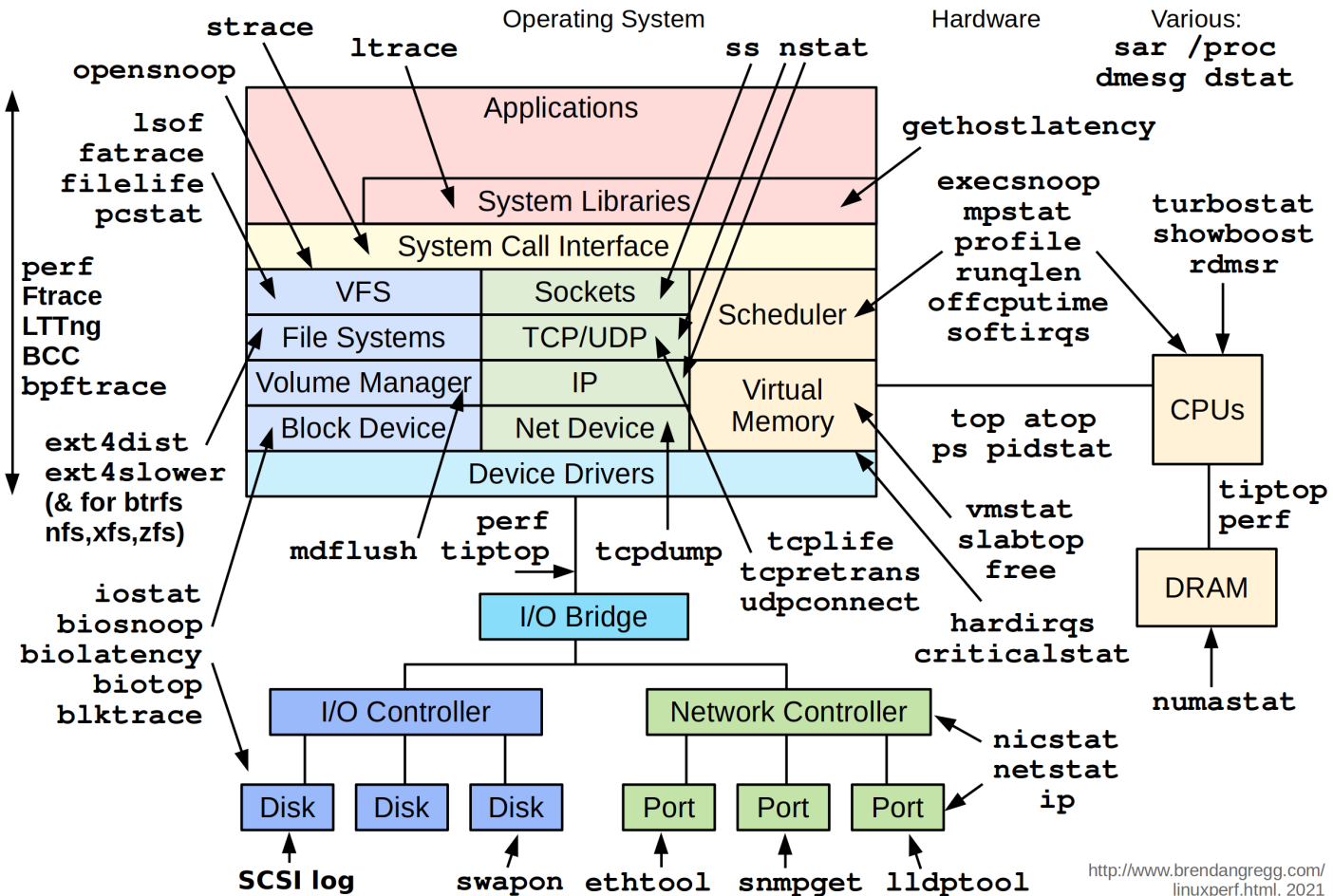
'nmcli'는 "NetworkManager" 런타임 관리 명령어. 아래 명령어 시스템 운영 및 관리 명령어.

```
ip <- ifconfig(namespace 지원안됨)
ss <- netstat(namespace 지원안됨, 느림)
ip r <- route(namespace 지원안됨)
nftables <- iptables(강화된 버전)
```

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

네트워크 관리 명령어

Linux Performance Observability Tools



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 일반적으로 다음과 같다.

네트워크 관리 명령어

ip address show

구성된 NIC의 아이피 정보 및 NIC상태 확인

ip addresss add <DEV>

여기에서 구성되는 아이피 정보는 일시적으로 시스템에서 저장. 재시작 시 해당 내용은 제거.
반드시 네트워크 설정은 NetworkManager에서 구성해야 됨.

ip link

연결이 되어 있는 NIC카드 정보

ip route

현재 구성이 되어있는 라우팅 테이블 정보

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

네트워크 관리 명령어

ss -antp

State	Recv-Q	Send-Q	Local Address:Port
LISTEN	0	128	*:22
.		users:(("sshd",pid=1210,fd=3))	
LISTEN	0	100	127.0.0.1:25
.		users:(("master",pid=1459,fd=13))	
ESTAB	0	0	192.168.90.226:22
192.168.90.91:39520		users:(("sshd",pid=1490,fd=3))	
LISTEN	0	128	:::22
:::*		users:(("sshd",pid=1210,fd=4))	
LISTEN	0	100	:::1:25
:::*			

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 관리 명령어

```
# 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
```

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.

더미 장치 생성하는 방법

NetworkManager

```
# 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 ← 명령어 기반 관리도구(자동화)

nmtui ← 텍스트 기반 관리도구

더미 장치 생성하는 방법

ip command 예제

```
# 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
```

네트워크 관리 명령어

```
# 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
```

네트워크 관리 명령어

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
```

네트워크 관리 명령어

ip -n switch1 netns set switch2 20

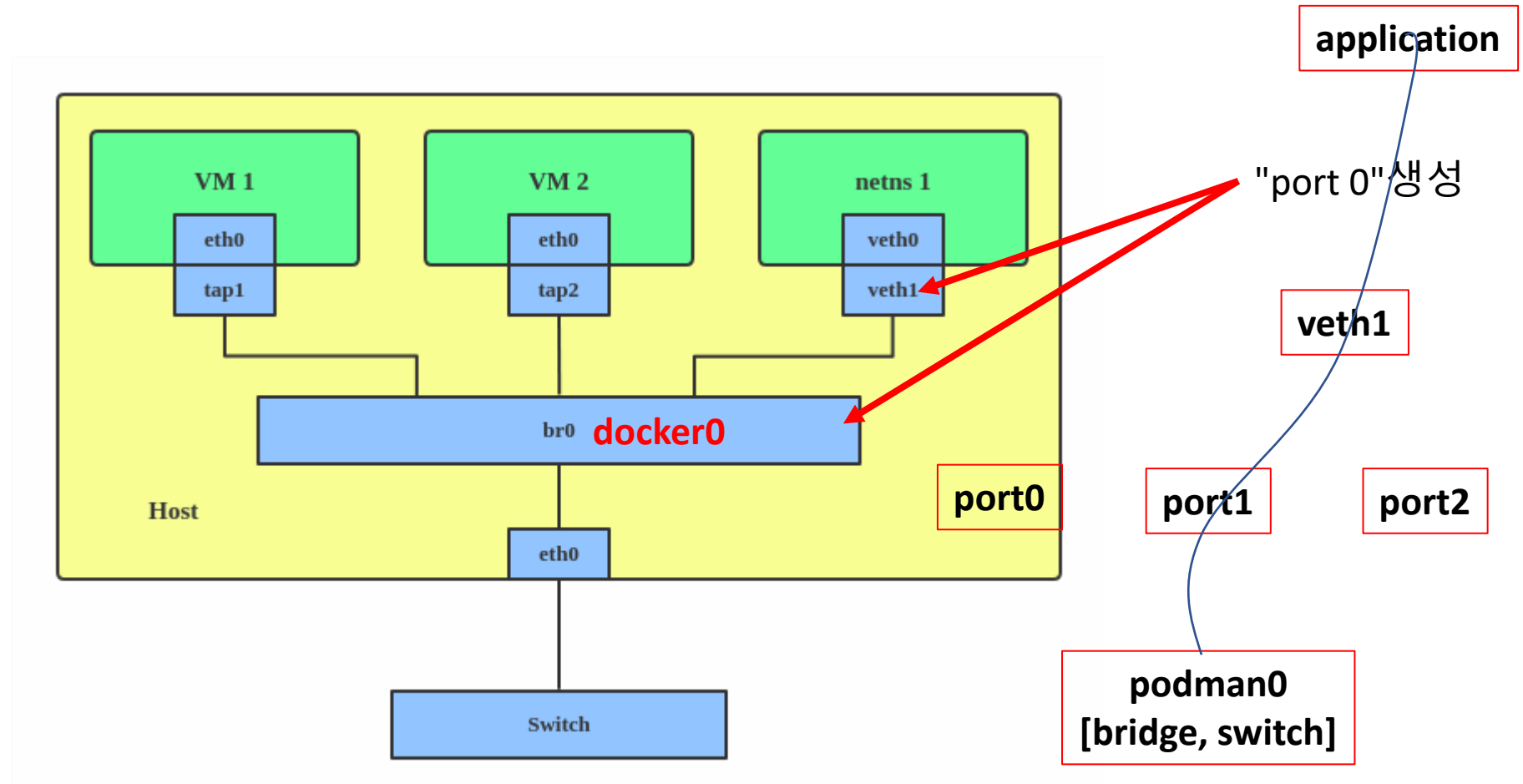
ip -n switch2 netns set switch1 30

ip -n switch2 netns set switch1 40

ip netns list-id target-nsid 12

ip netns list-id target-nsid 12 nsid 13

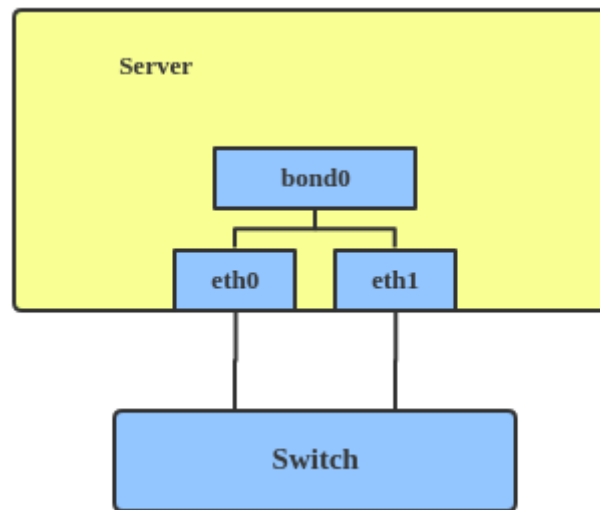
ip add bridge



ip add bridge

```
# 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
```

ip add bond



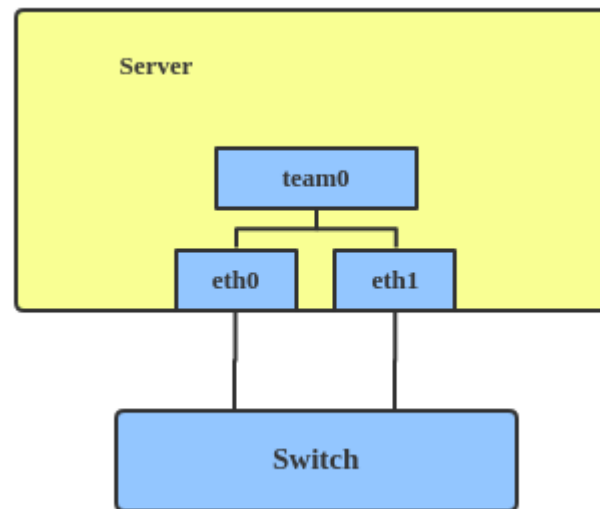
ip add bond

```
# ip link add bond1 type bond miimon 100 mode active-backup
```

```
# ip link set dummy0 master bond1
```

```
# ip link set dummy1 master bond1
```

ip add teamd



ip add teamd

```
# teamd -o -n -U -d -t team0 -c '{"runner": {"name":  
"activebackup"},"link_watch": {"name": "ethtool"}}'
```

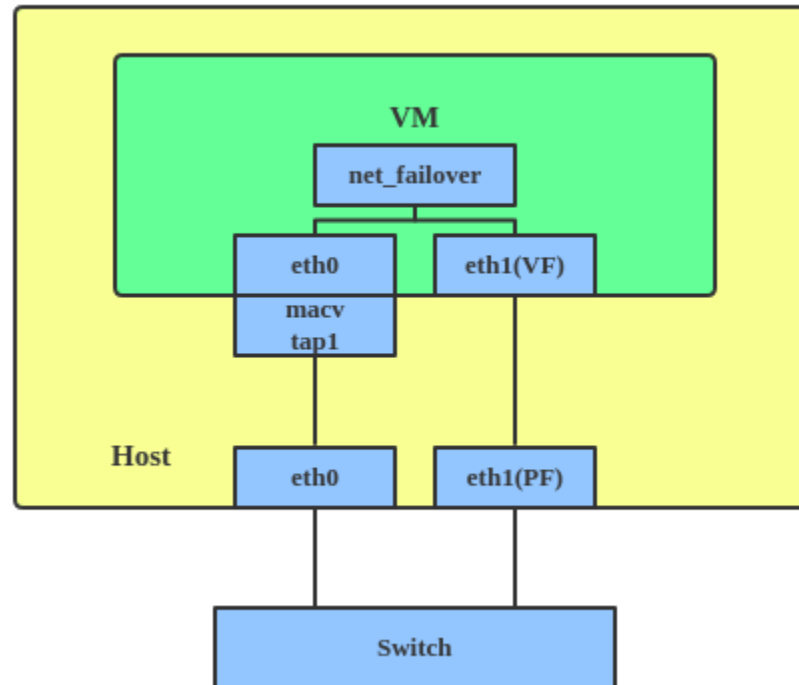
```
# ip link set dummy0 down
```

```
# ip link set dummy1 down
```

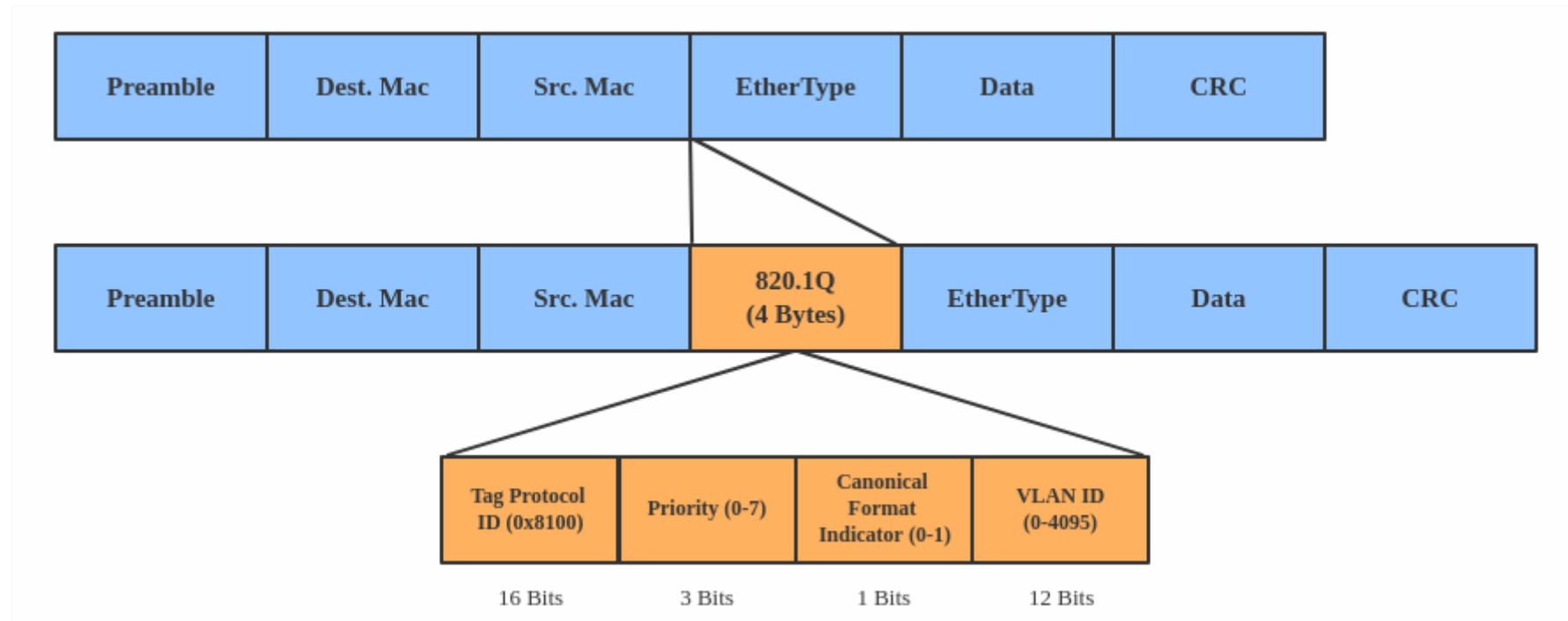
```
# teamdctl team0 port add dummy0
```

```
# teamdctl team0 port add dummy1
```

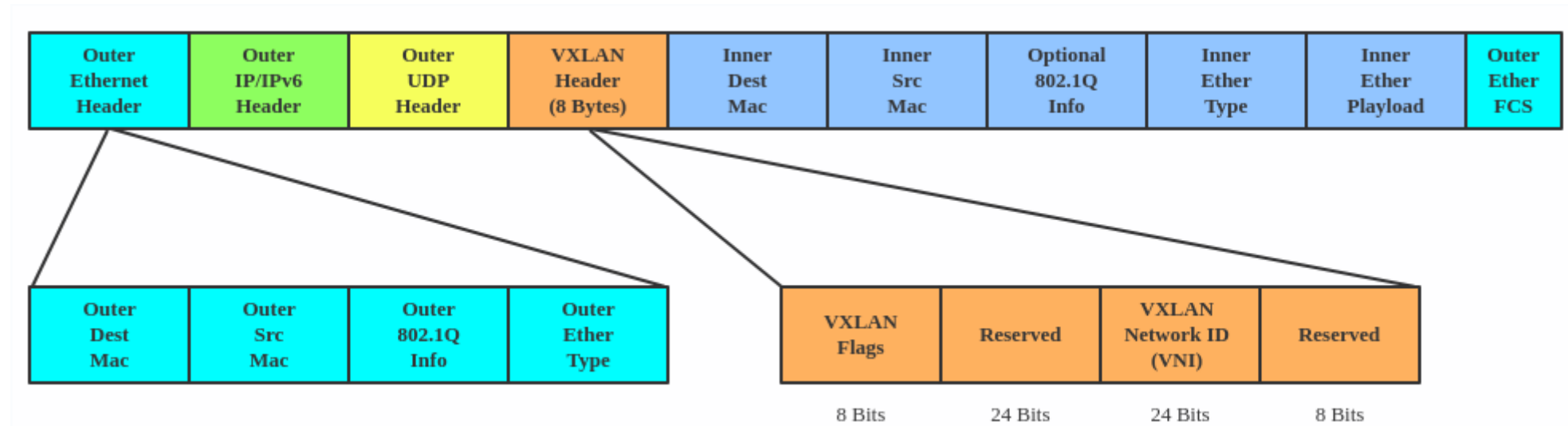
ip add vlan



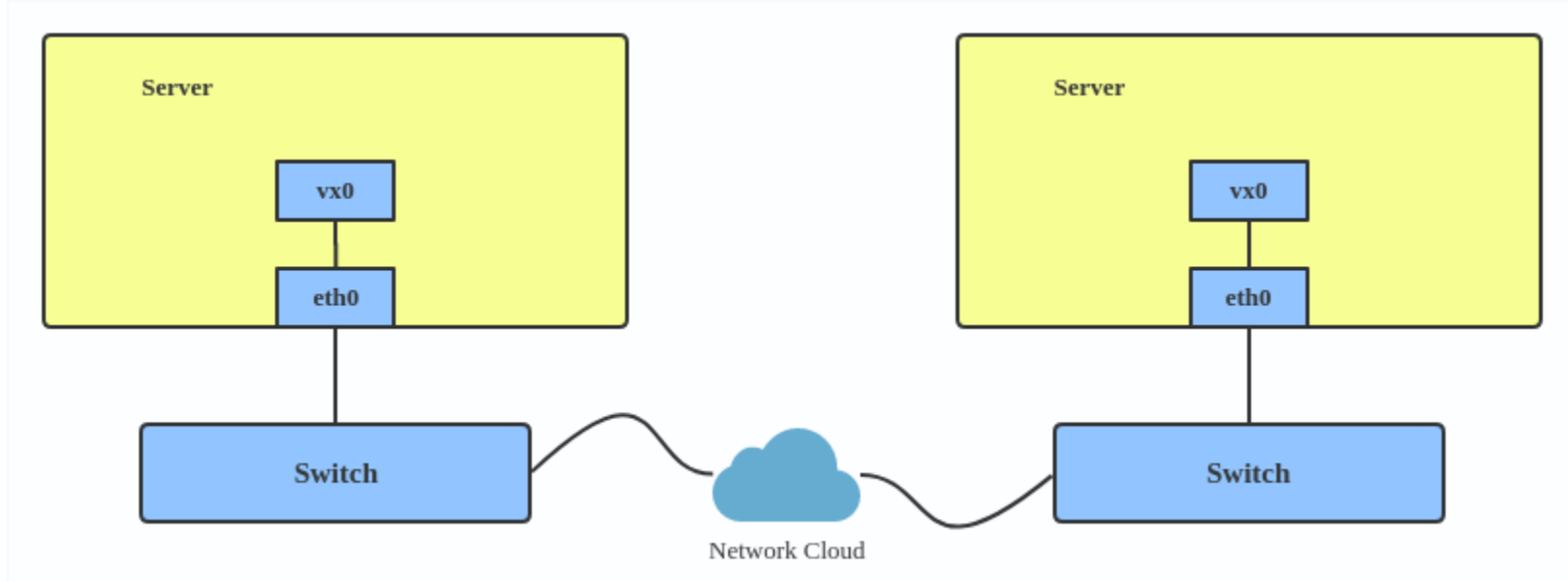
ip add vlan



ip add vxlan



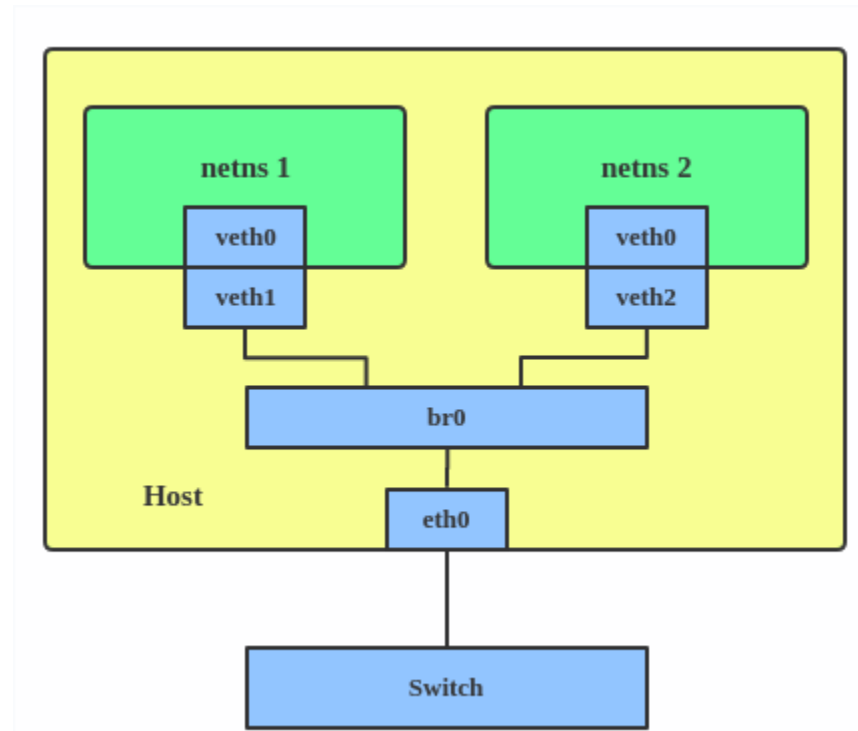
ip add vxlan



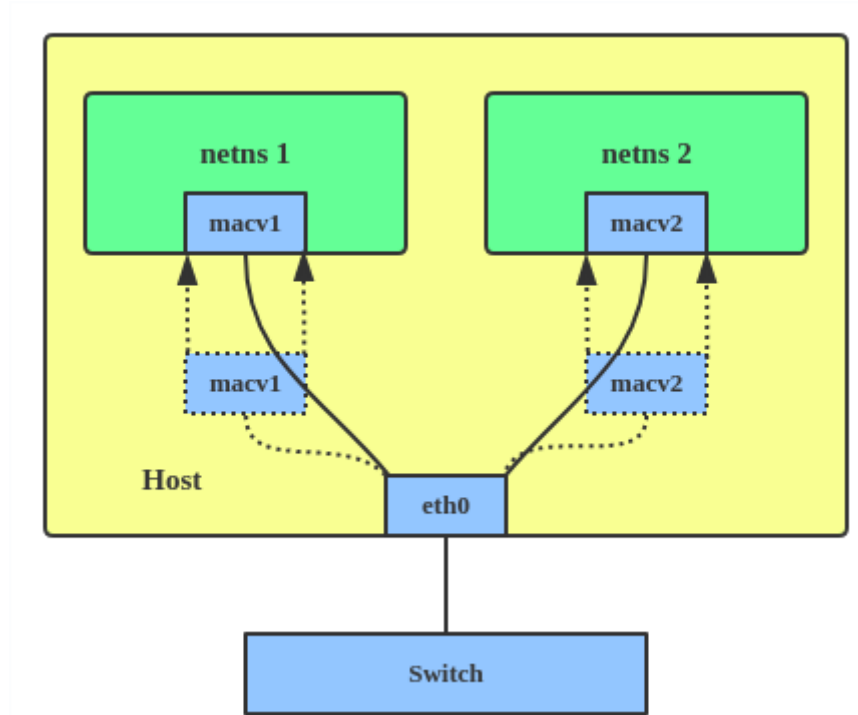
ip add vxlan

```
# ip link add vx0 type vxlan id 100 local 1.1.1.1 remote 2.2.2.2 dev  
eth0 dstport 4789
```

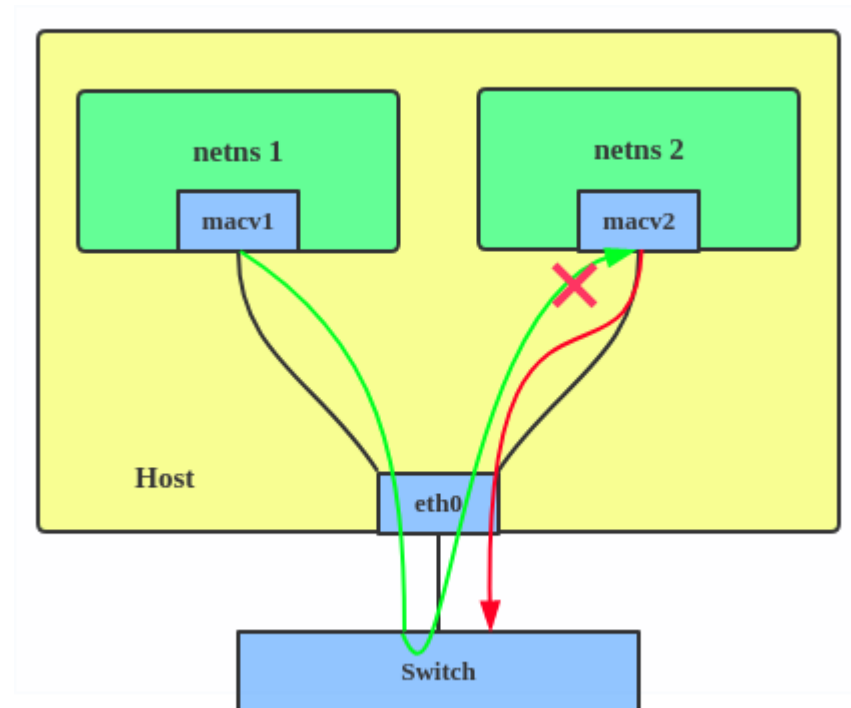
TAP/VETH(bridge side)



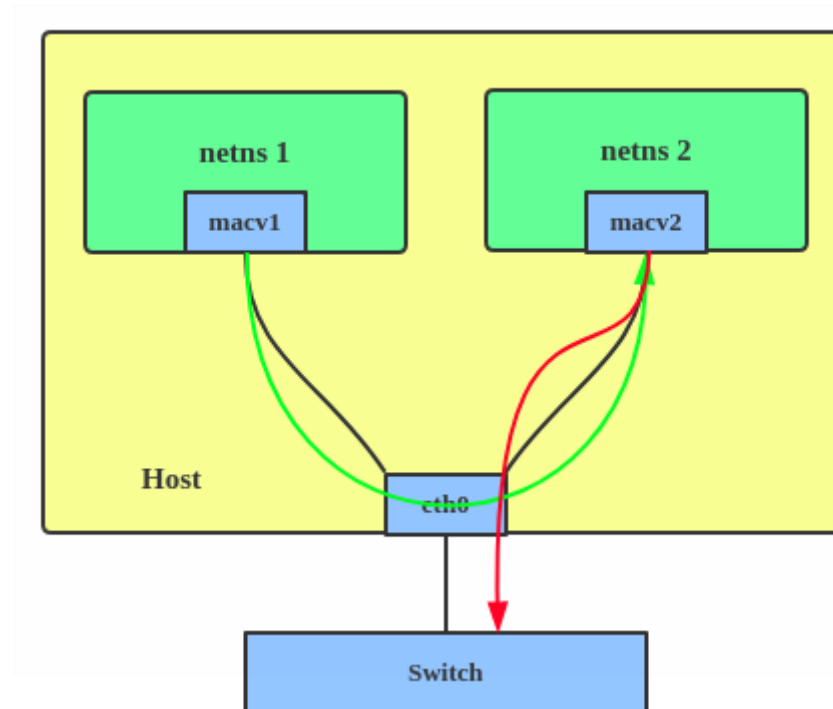
MACVLAN without Bridge(direct namespace)



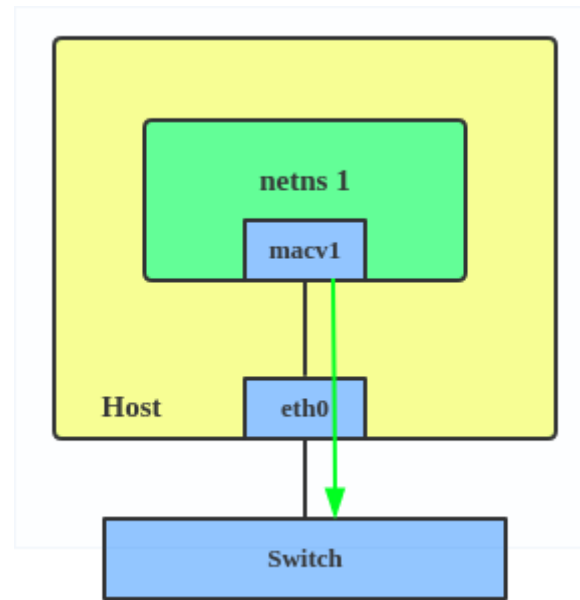
MACVLAN, VEPA



MACVLAN, Bridge



Passthru Bridge



MACVLAN

```
# ip link add macvlan1 link ens5 type macvlan mode bridge
```

```
# ip link add macvlan2 link ens5 type macvlan mode bridge
```

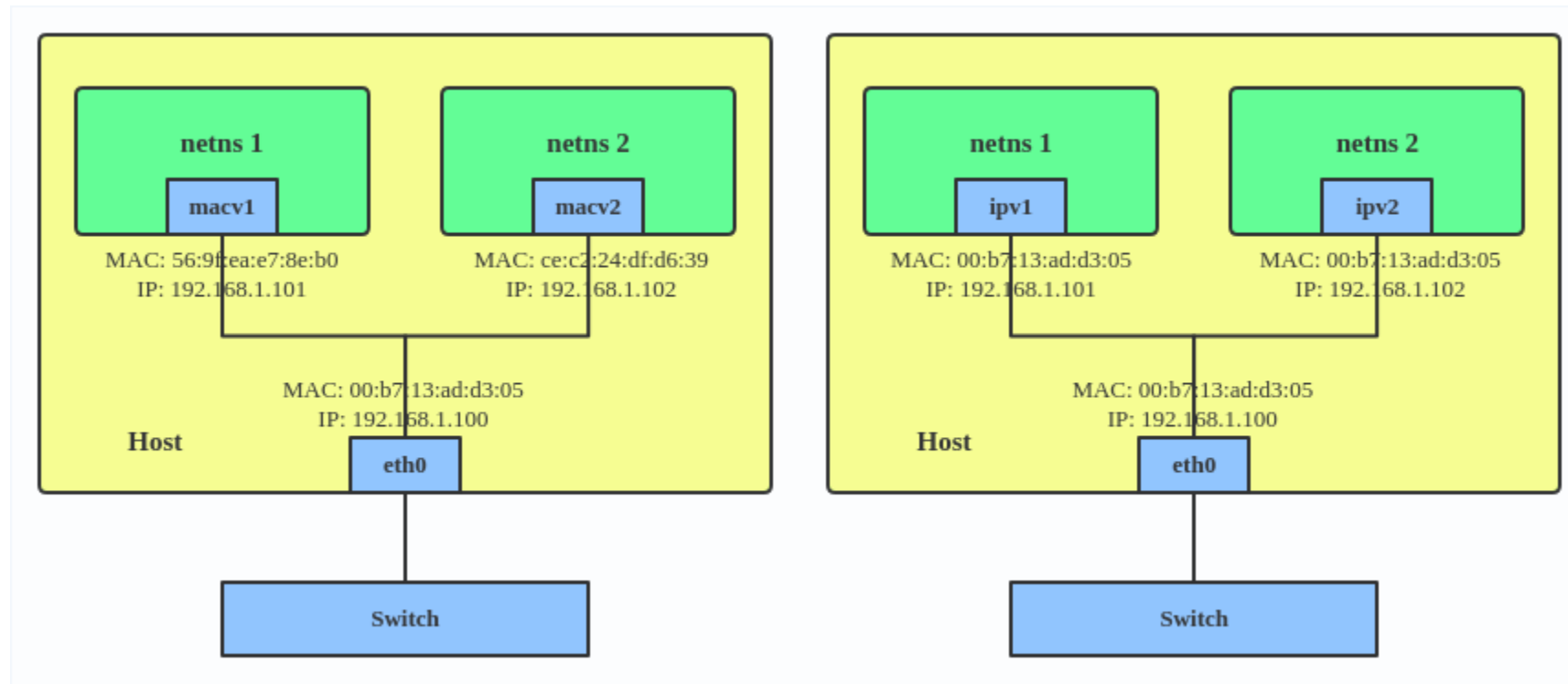
```
# ip netns add net1
```

```
# ip netns add net2
```

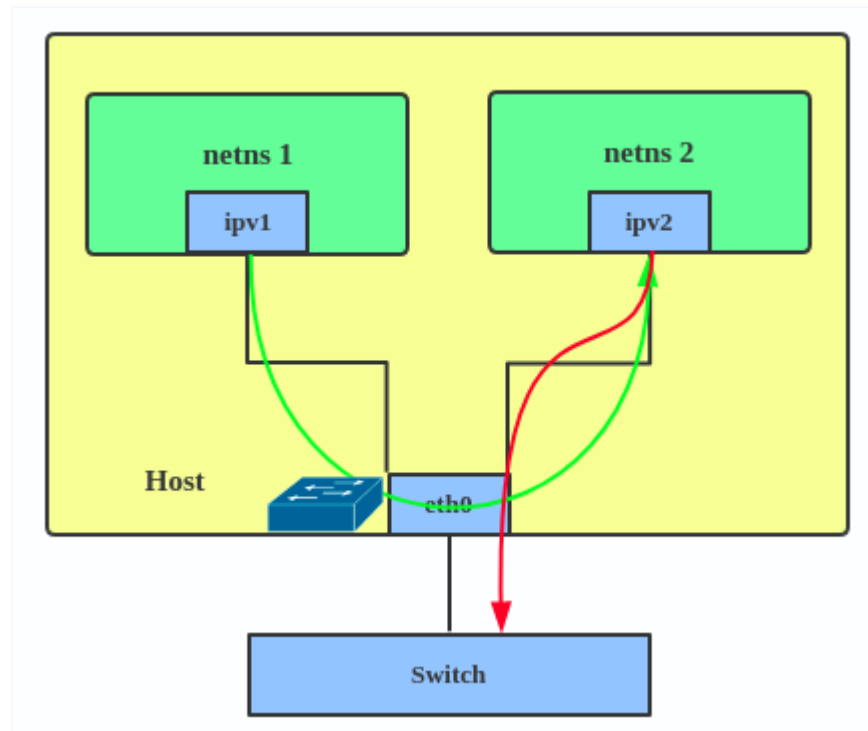
```
# ip link set macvlan1 netns net1
```

```
# ip link set macvlan2 netns net2
```

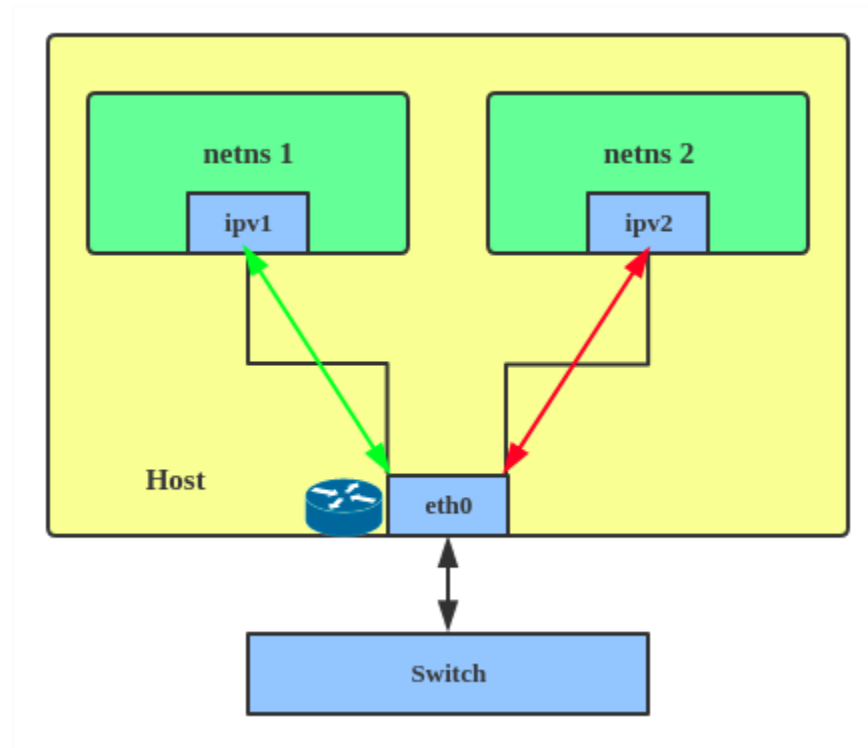
IPVLAN



IPVLAN



IPVLAN



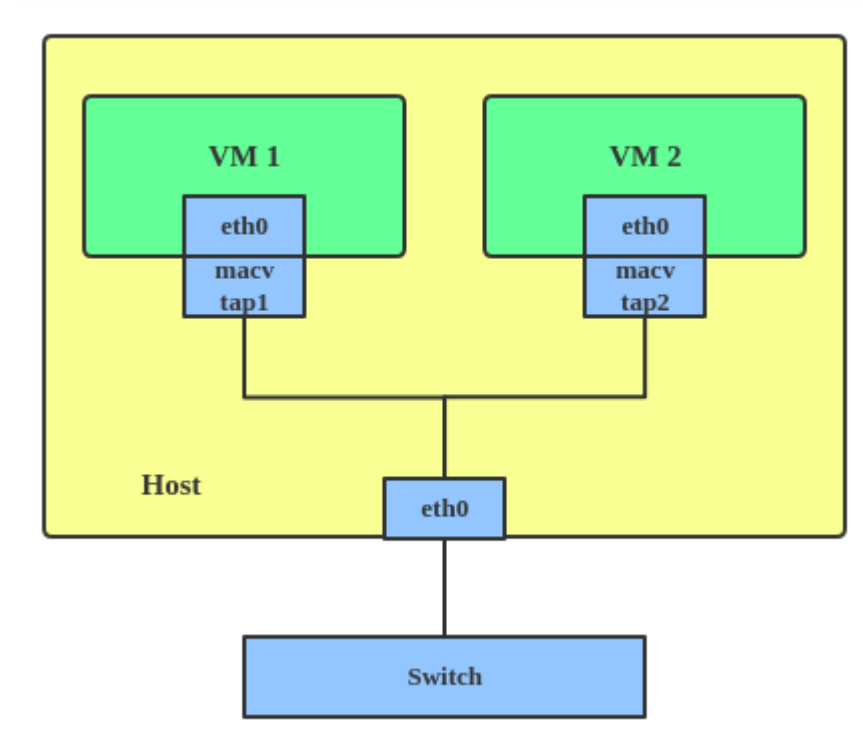
IPVLAN

```
# ip netns add ns0
```

```
# ip link add name ipvl0 link eth0 type ipvlan mode l2
```

```
# ip link set dev ipvl0 netns ns0
```

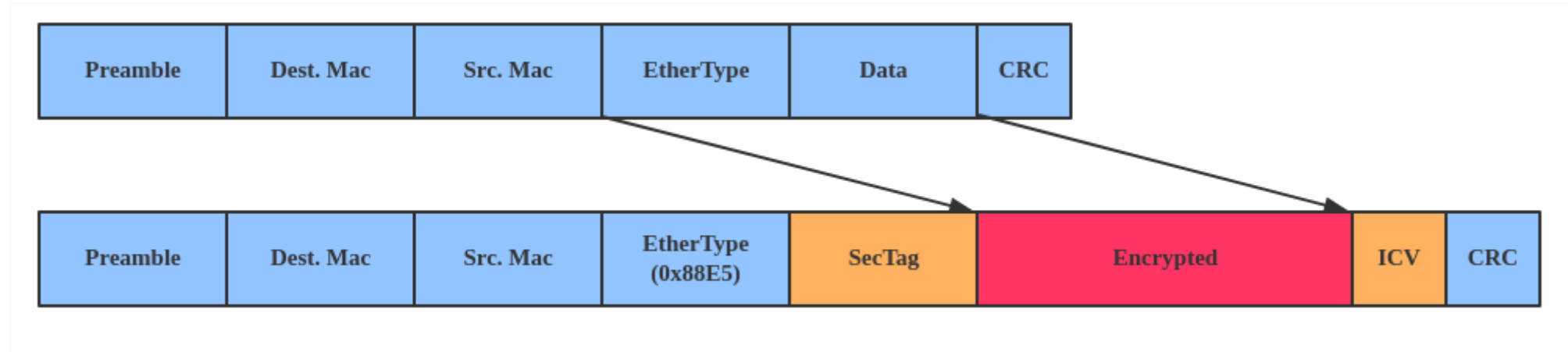
MACVTAP/IPVTAP



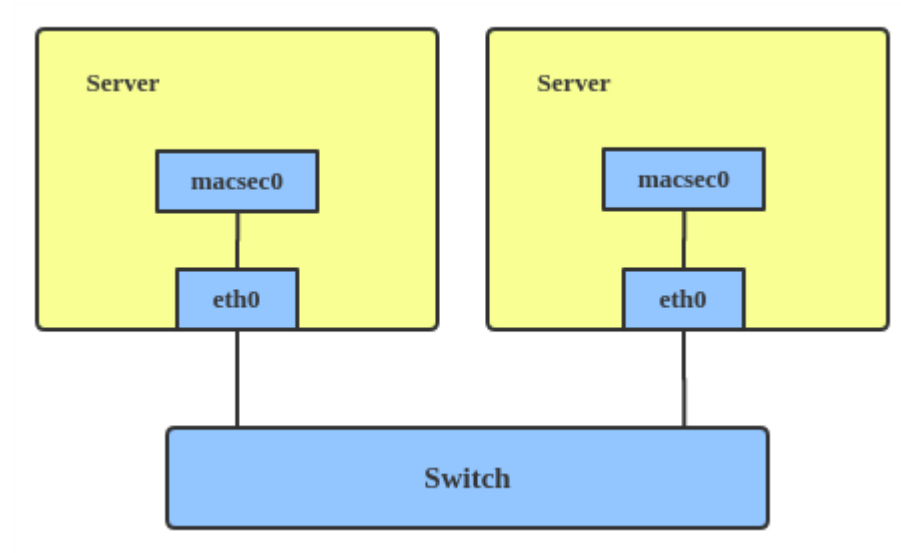
MACVTAP/IPVTAP

```
# ip link add link ens5 name macvtap0 type macvtap
```

MACsec



MACsec



MACsec

```
# ip link add macsec0 link ens5 type macsec
```

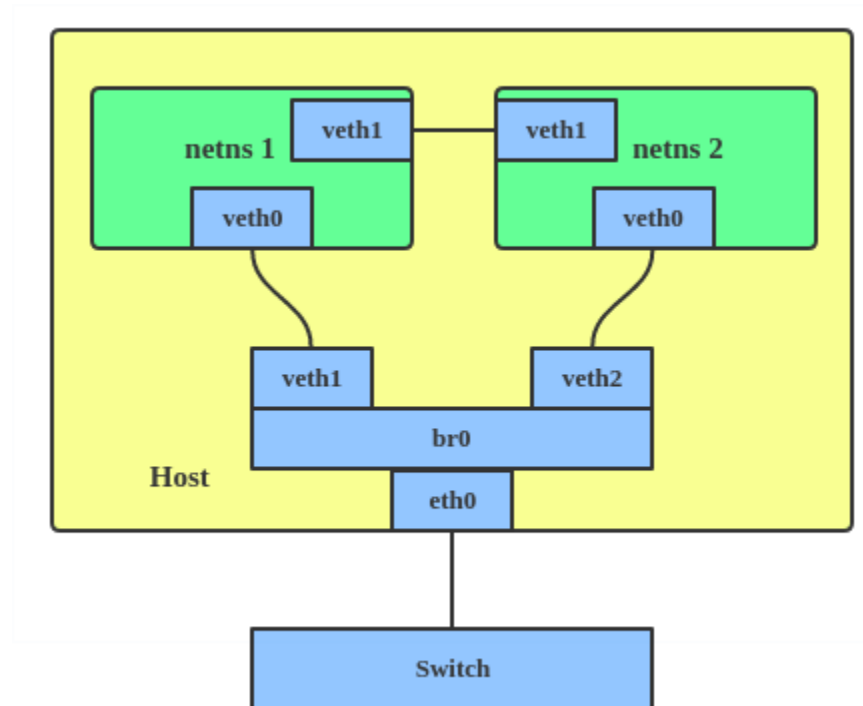
VETH

```
# ip netns add net1
```

```
# ip netns add net2
```

```
# ip link add veth1 netns net1 type veth peer name veth2 netns net2
```

VETH



VETH

```
# ip netns add net1
```

```
# ip netns add net2
```

```
# ip link add veth1 netns net1 type veth peer name veth2 netns net2
```

VXCAN

```
# ip netns add net1
```

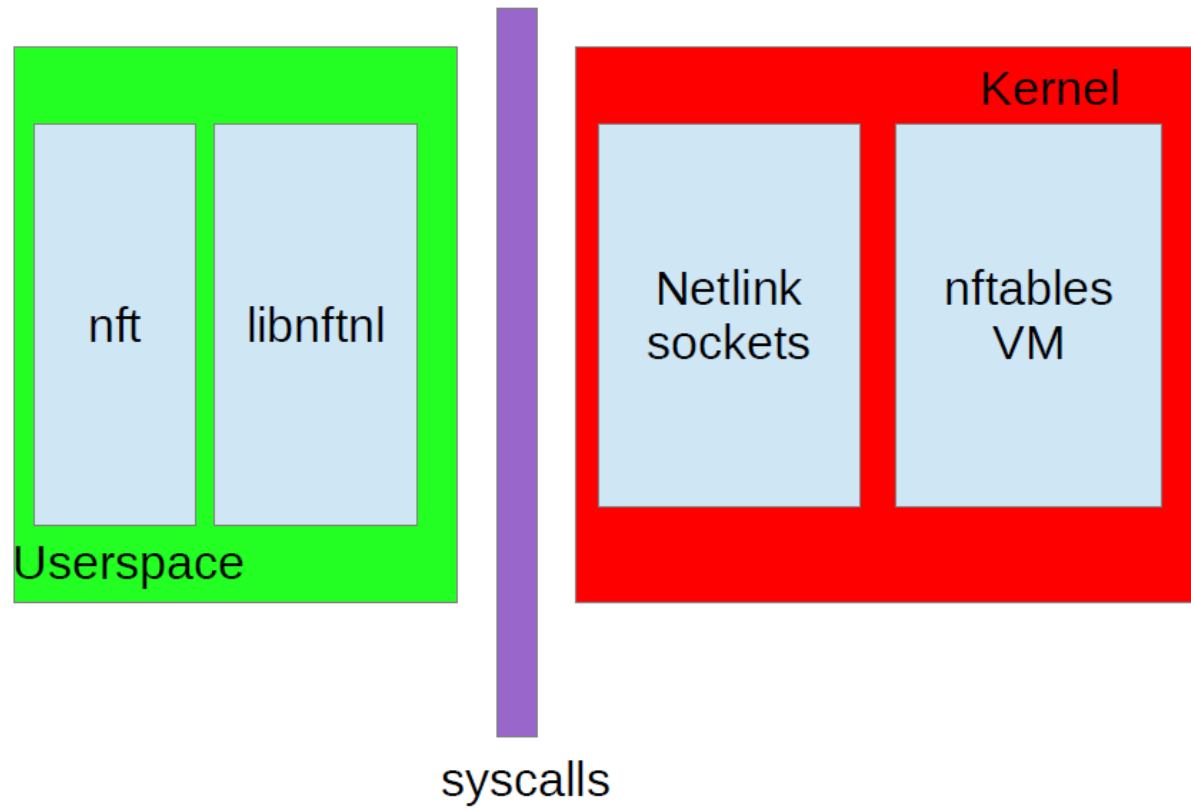
```
# ip netns add net2
```

```
# ip link add vxcan1 netns net1 type vxcan peer name vxcan2 netns  
net2
```

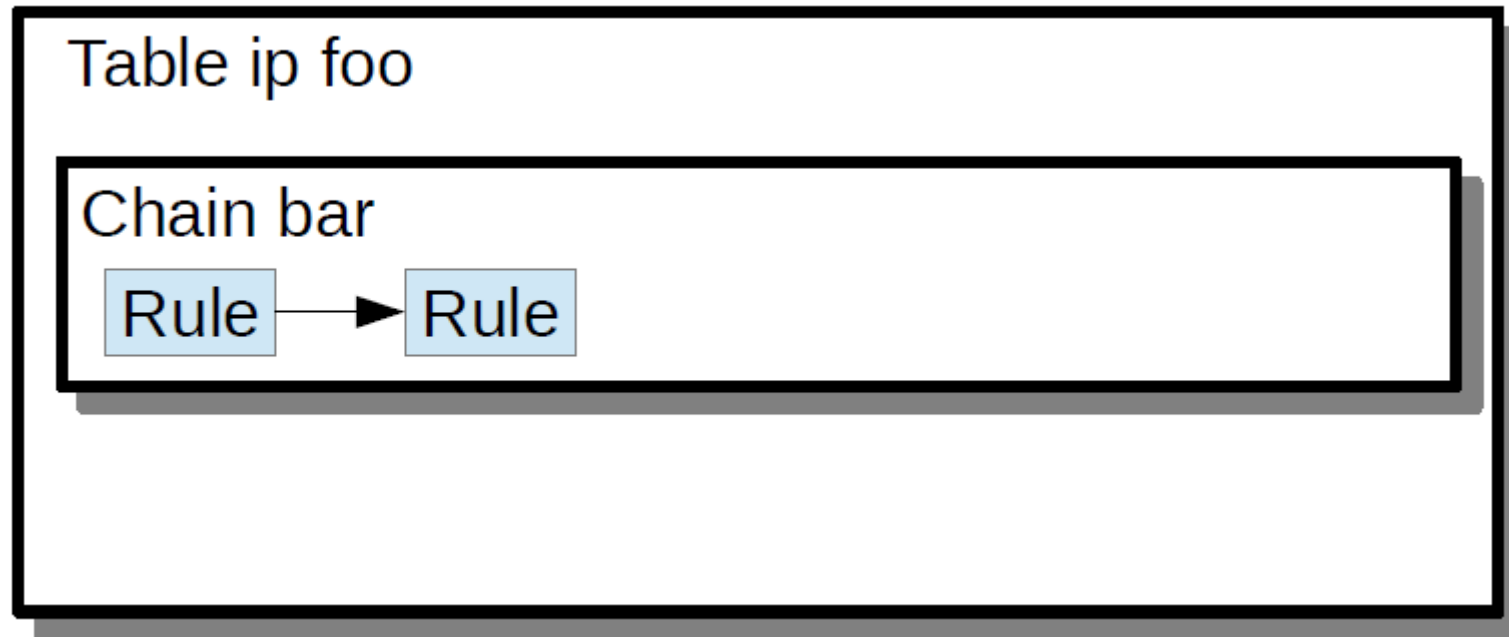
방화벽 및 네트워크 설정

nftables, firewalld, NetworkManager

nftables



nftables



nftables

목록 출력

nft list table filter

-> # nft list tables

nft list counters

특정 아이피 드롭

nft add rule ip filter OUTPUT ip daddr 1.2.3.4 drop

nftables

패키지

```
# yum install nftables
```

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

여전히 호환성 모드로 iptables 사용은 가능하나 가급적으면 nftables 기반으로 작업 권장. 혹은 firewalld로 작업.

nftables

nftables에서 필터(filter)생성은 다음과 같다. 자세한 명령어 설명은 아래에서 따로 자세히 다룬다.

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: 일반적인 IPv4프로토콜

arp: Address Resolution Protocol 제어 체인

ip6: IPv6

bridge: Linux Bridge 및 OVS Bridge

inet:

netdev: Container 및 VirtualMachine에서 사용하는 TAP(Test Access Point) 장치

counter 모듈 설명

자세한 설명은 아래 주소 참고

<https://lwn.net/Articles/631372/>

daddr: 도착 아이피

dport: 도착 포트

output/input: 입출력 패킷

nftables

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

-> 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
```

-> 필터리스트의 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

nftables

handle <NUMBER>

정책(rule)의 번호

nft add rule filter/ip <TYPE>

rule를 "filter"에 추가한다.

firewalld

firewalld

기존에 사용하는 iptables, nftables위에 고급 계층을 올려서 사용자가 쉽게 사용할 수 있도록 한다.

firewall-cmd명령어 기반으로 **영역(zone)**, **서비스(service)**, **포트(port)**와 같은 elements를 xml기반으로 쉽게 구성이 가능하다.

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=
```

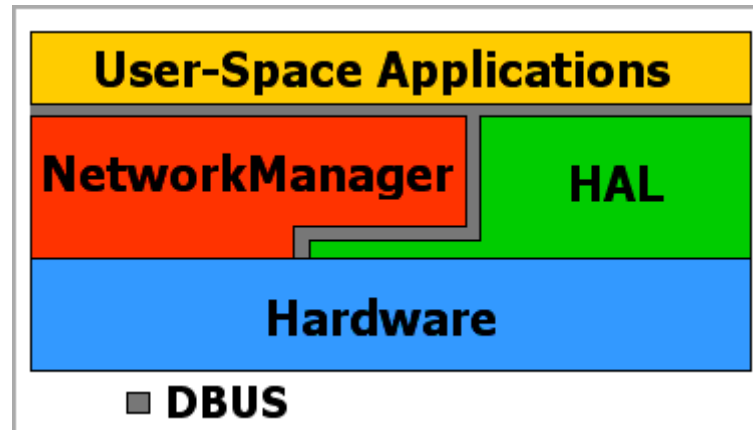
NetworkManager

네트워크 매니저는 **RHEL 7/8/9**에서 지원한다.

다만, "**RHEL 9**"에서는 "**7/8**"과 다르게 더 이상 "**ifcfg-***"를 지원하지 않는다.

모든 네트워크 파일은 INI형태로 **"/etc/NetworkManager/"**에 관리한다. 관리 파일은 '**\$ NetworkManager --print-config**' 명령어로 확인이 가능하다.

NetworkManager



NetworkManager

systemctl start NetworkManager

nmcli

nmtui

nm-connection-editor

NetworkManager

/var/run/NetworkManager

devices initrd no-stub-resolv.conf resolv.conf system-connections

/var/lib/NetworkManager/

internal-590765f7-ea80-49b3-aca2-8a570d85952b-ens3.lease

NetworkManager-intern.conf secret_key seen-bssids timestamps

NetworkManager

/run/NetworkManager/

devices initrd no-stub-resolv.conf resolv.conf system-connections

/etc/NetworkManager/

conf.d dispatcher.d dnsmasq.d dnsmasq-shared.d
NetworkManager.conf system-connections

NetworkManager

/run/NetworkManager/

devices initrd no-stub-resolv.conf resolv.conf system-connections

/etc/NetworkManager/

conf.d dispatcher.d dnsmasq.d dnsmasq-shared.d
NetworkManager.conf system-connections

패키지 관리 방법

dnf, yum, rpm and modules

dnf

dnf는 기존에 사용하던 yum명령어를 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 비슷한 기능을 제공하기도 함.

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 updat
-----------------	--------------	---

d] 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 month
-----------------	-----	---

d] s.

container-tools	2.0	common [Stable versions of podman 1.6, buildah 1.11, skopeo 0.1, runc, common, etc as well as dependencies such as container-selinux built and tested together, and supported as documented on the App
-----------------	-----	---

d] lication Stream lifecycle page.

container-tools	3.0	common [Stable versions of podman 3.0, buildah 1.19, skopeo 1.2, runc, common, etc as well as dependencies such as container-selinux built and tested together, and supported as documented on the App
-----------------	-----	---

d] lication Stream lifecycle page.

container-tools	4.0	common Stable versions of podman 4.0, buildah 1.24, skopeo 1.6, runc, common, CRIU, Udica, etc as well as dependencies such as container-selinux built and tested together, and supported as document
-----------------	-----	---

ed on the Application Stream lifecycle page.

rollback

yum 및 dnf는 rollback기능을 제공한다.

dnf history **rollback**

undo

redo

list

info

yum vs dnf-3

YUM과 DNF의 제일 큰 차이점은 yum은 순수하게 파이썬으로 작성이 되어 있다.

하지만, dnf는 libdnf3라는 C언어로 래핑 되어 있으며, 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.

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

ps 명령어는 어떤 옵션을 사용하느냐 따라서 시스템에 부하 수준이 다르다.

구동 중인 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/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

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

데모 및 설명

- 1, HUP, reload a process
- 9, KILL, kill a process without page down
- 15, TERM**, gracefully stop a process

pskill/pgrep

pskill은 프로세스의 이름 혹은 다른 속성을 확인하여 시그널 전달

```
$ pskill <signal> <process_name>
```

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

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

pskill/pgrep

```
$ pgrep -c ssh
```

```
10
```

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

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


pskill/pgrep

```
$ 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
```

pskill/pgrep

\$ 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 관리를 위한 명령어

useradd, usermod, chmod, chown

coreutils

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 useradd명령어 기반으로 사용자 추가한다.

useradd, passwd

adduser

이 명령어는 데비안 계열 및 BSD에서 많이 사용하는 명령어.

레드햇 계열 배포판은 useradd, adduser 둘 다 가지고 있지만 실제로는 **useradd**를 사용한다.

adduser는 링크로 구성이 되어 있다.

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>
```

DAY 2

디스크 관리를 위한 명령어

block device and disk recognized
management and naming of devices

디스크 관리를 위한 명령어

디스크 관리를 위해서 많이 사용하는 명령어는 다음과 같다.

lsblk

blkid

fdisk/gdisk

(k)partx, partprobe

block

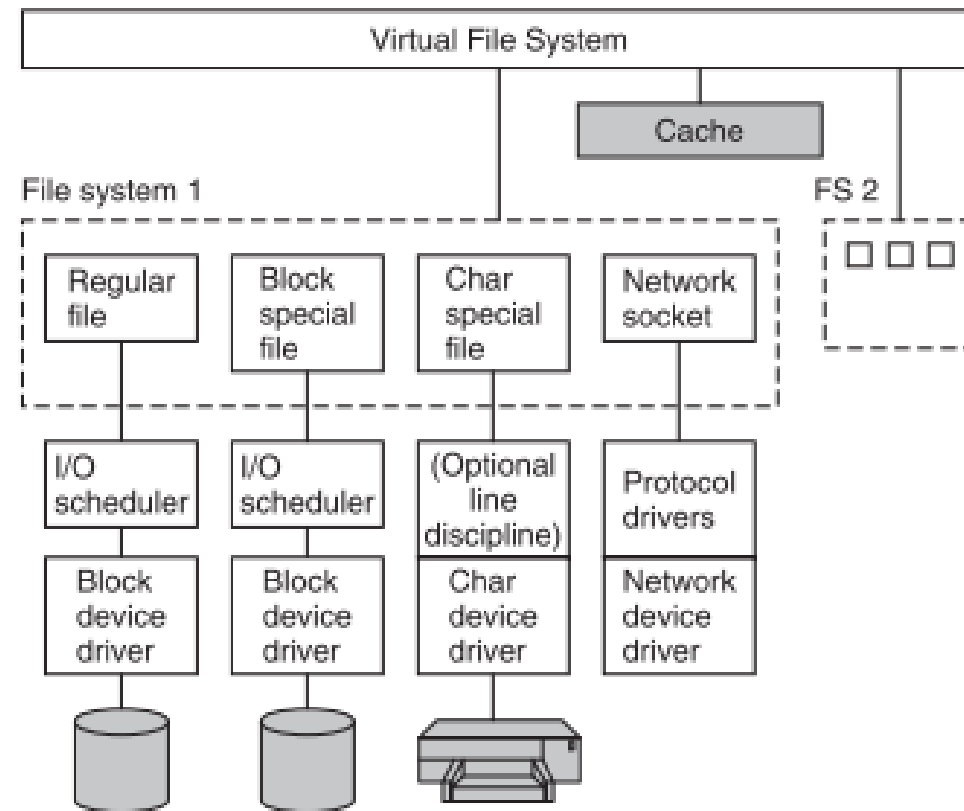


Figure 10-22. The Linux I/O system showing one file system in detail.

lsblk

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

lsblk

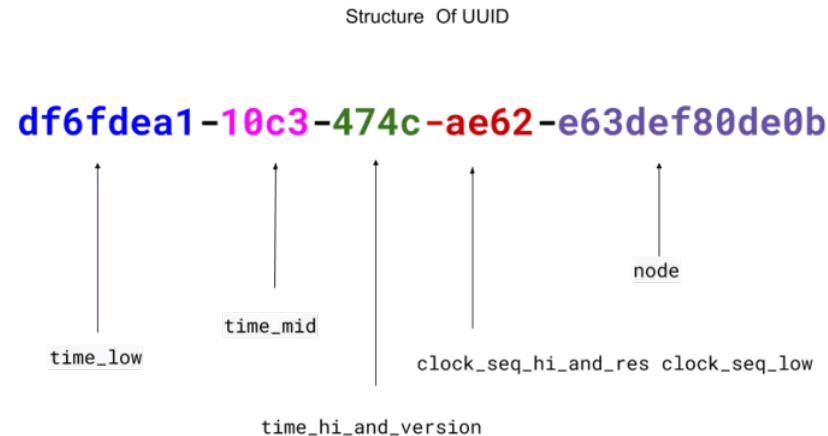
블록장치 즉, "vda", "sda", "hda" 같은 장치들은 사용이 가능한 이름으로 출력이 된다.

lsblk

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINTS
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	

blkid

슈퍼 블록에 기록이 되어 있는 파티션 혹은 파티션 정보를 읽어와서 UUID 정보를 출력한다. 여기서 말하는 UUID는 Universal Unique Identifier의 약자이며, 128비트로 구성되어 있는 정보이다.



blkid

이 정보들은 슈퍼블록에 저장되어 있다. UUID는 다음과 같이 정보를 사용자에게 제공한다.

```
uint8_t s_uuid[16] = {0xf3, 0x58, 0x6b, 0xaf, 0xb5, 0xaa, 0x49, 0xb5,  
                      0x8d, 0x6c, 0x05, 0x69, 0x28, 0x4c, 0x63, 0x9f};  
  
printf("%02x%02x%02x%02x%02x%02x-%02x%02x-%02x%02x-%02x%02x-  
%02x%02x%02x%02x%02x%02x\n",  
       s_uuid[0], s_uuid[1], s_uuid[2], s_uuid[3], s_uuid[4], s_uuid[5], s_uuid[6],  
       s_uuid[7],  
       s_uuid[8], s_uuid[9], s_uuid[10], s_uuid[11], s_uuid[12], s_uuid[13],  
       s_uuid[14], s_uuid[15]);
```

디스크 파티셔닝

gdisk/fdisk/cfdisk/parted

parted, partprobe

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: Master Boot Recode

설명: 최대 4개의 Primary 파티션 논리적 파티션으로 최대 32개 파티션

GPT: GUID Partition Table

GUID: globally unique identifier

설명: 최대 128개의 파티션을 제공함

fdisk/gdisk

강사 데모

cfdisk

강사 데모

partprobe

partprobe는 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 권장한다.

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

partx

'partx'는 'partprobe'와 비슷하지만, 조금 더 다양한 디스크 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 중지되는 경우가 있다(아주 잠깐).

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

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 "DeviceMapper"와 연동이 되기 때문에 멀티패스 설정 시, 인식이 잘 안되면 'kpartx'로 디스크 및 파티션 정보 갱신을 시도한다.

kpartx

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.

다중 스왑 및 파일 스왑

SWAP within Partition and File

다중 스왑 및 파일 스왑

리눅스에서 사용하는 스왑은 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
```

다중 스왑 및 파일 스왑

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"
```

```
# 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
```

가상머신 및 컨테이너 구성 및 구현

libvirt

podman

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.

Podman은 다음과 같은 서비스가 있다.

podman.service

io.podman.service

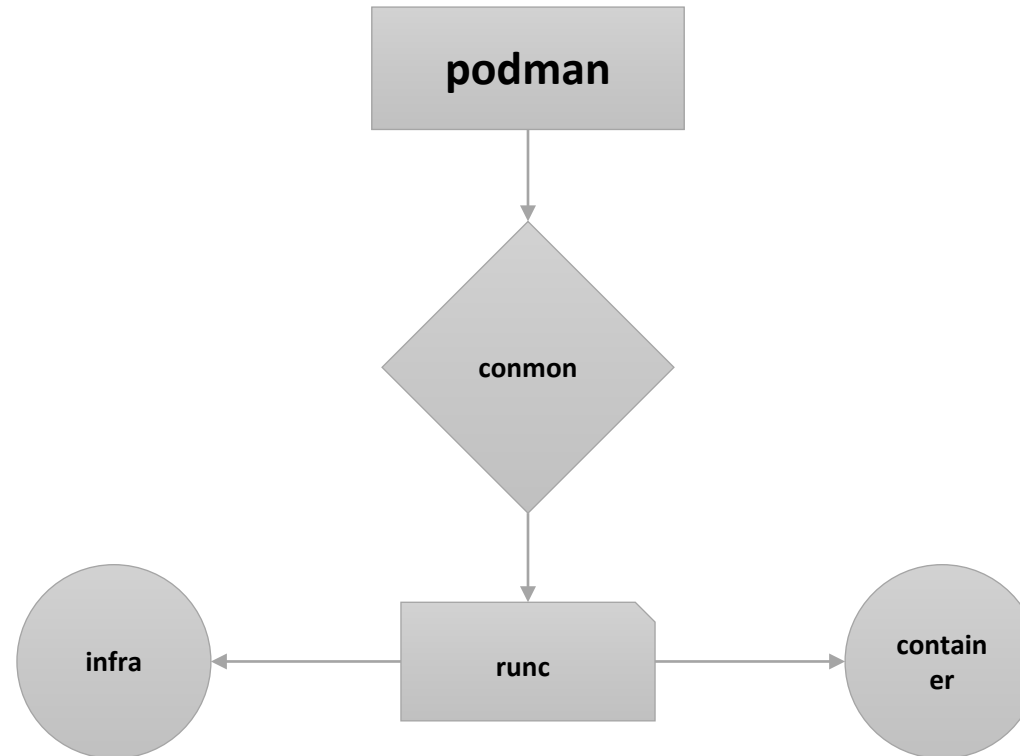
배포판 별로 다르지만 보통은 "**podman.service**"이름으로 운영이 된다.

레드햇 계열은 설치 시 다음과 같은 명령어로 설치한다.

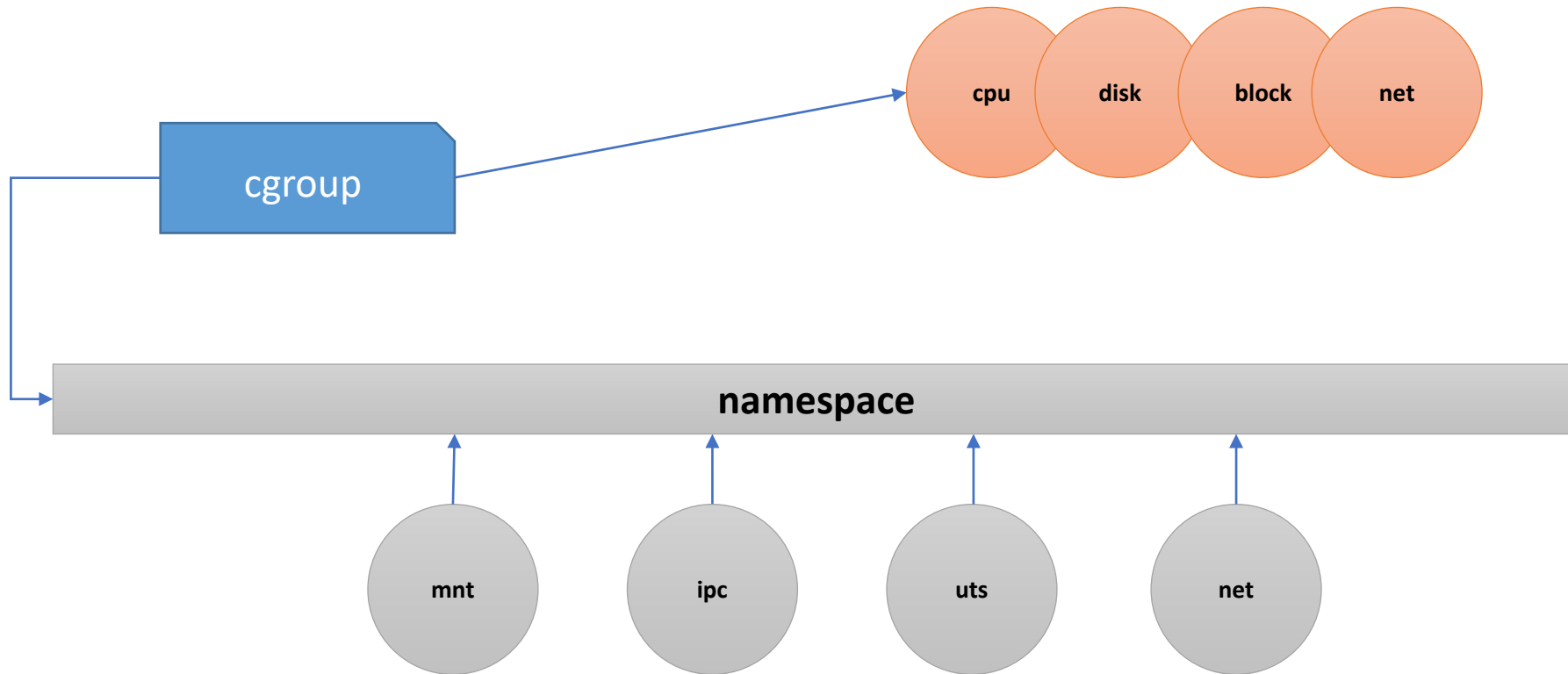
```
# dnf install podman
```

```
# dnf module list containers
```

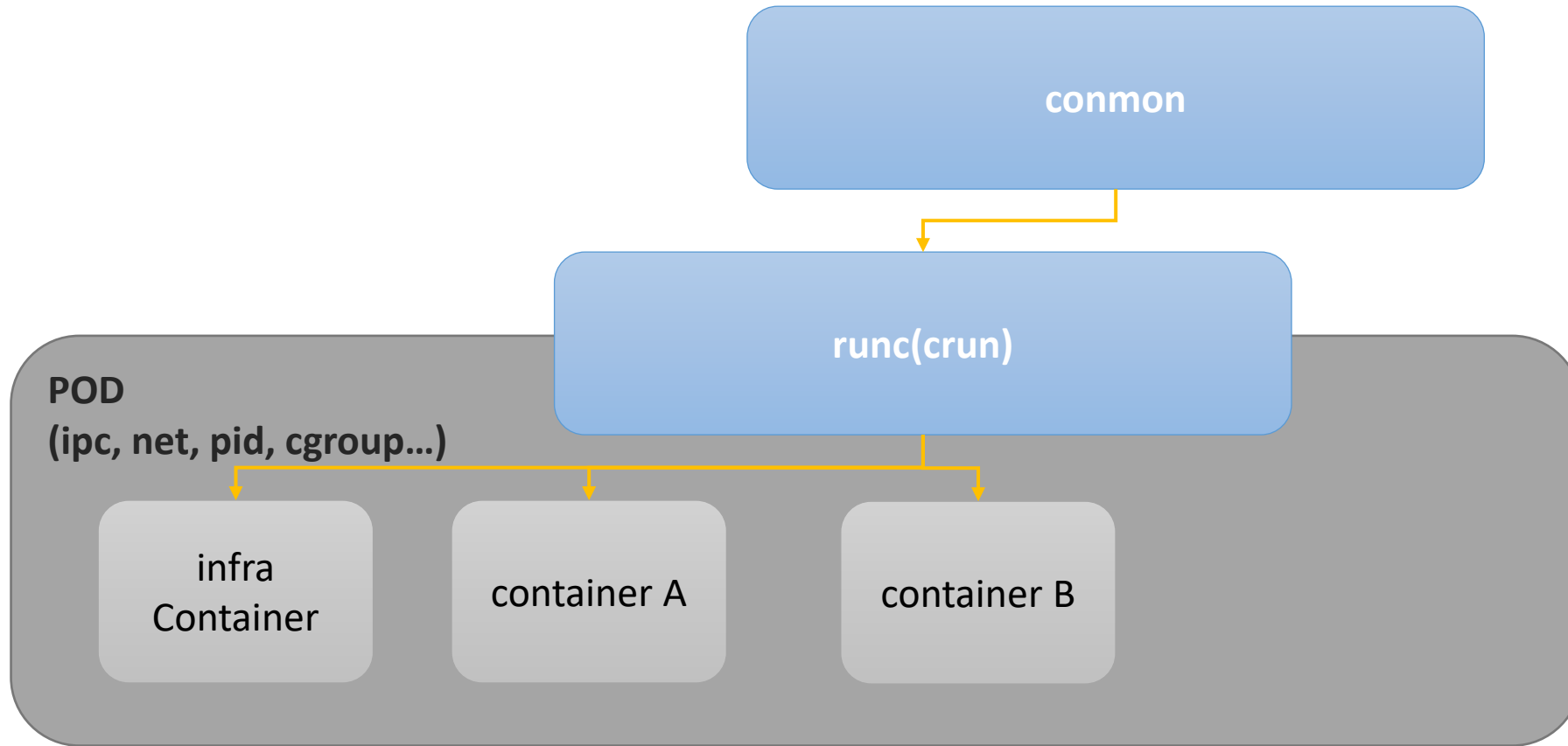

podman



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.

DAY 3

간단한 앤서블 문법

introduce of ansible

앤서블 기초

교육목적

이번 앤서블 교육은 다음 대상으로 제작 및 구성.

1. 앤서블 처음 사용하는 사용자
2. 실무까지는 아니어도 어떠한 방식으로 동작하는지 궁금하신 사용자
3. 간단하게 YAML 형태로 동작 학습
4. 전체적인 용어 및 기능 학습

앤서블

앤서블 소개

앤서블은 본래 Ansible Community, Company에서 제작하였고 릴리즈 도중에 레드햇이 인수하였음.

앤서블은 총 두 가지 릴리즈를 유지하고 있음.

- **Ansible Core**
- **Ansible Engine**

앤서블

앤서블 코어

코어는 앤서블 앤서블 핵심 모듈로 구성되어 있으며, 그 이외 확장으로 **posix**, collection, community로 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.

앤서블

앤서블 코어

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

앤서블

- ▶ 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

앤서블 소개

앤서블 타워

앤서블 타워는 두 가지 버전으로 구성이 되어있다.

첫 번째는 **앤서블 AWX 업 스트리밍 버전**이 있으며 레드햇에서 제고하는 서브스크립션 버전. 제일 큰 차이점은 버전 픽스가 되지 않는 롤링 업데이트 버전이다.

타워의 큰 특징은 웹 대시보드에서 API기반으로 **앤서블 호출, 구성 및 관리가 가능**하며 **사용자 단위로 작업 할당 및 추적**이 가능하다.

앤서블

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-playbook 명령어 사용 방법**

앤서블

editor

리눅스 콘솔에서 작성 시 사용하는 대표적인 에디터는 **vi/vim** 에디터가 있다.

거의 대다수 리눅스는 기본적으로 vi는 설치가 되어 있으나, vim설치가 되어 있지 않는 경우가 있다.

이러한 이유로 vim설치 이외 몇 가지 기능을 추가적으로 구성 및 활성화 해야 한다.

앤서블

editor

다른 편집기를 원하는 경우, 다른 에디터를 사용해서 교육을 진행 하셔도 됩니다. 😊

앤서블

YAML

YAML은 HTML처럼 사양만 지정이 되어 있고, 여기에서 필요한 방법을 가져와서 자신이 사용하는 언어 기반으로 **포매팅/파싱/프로세싱** 하면 된다.

앤서블

<https://yaml.org/>

```
%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/
```

앤서블 준비

SSH

앤서블은 기본적으로 두 가지 접근 방식을 제공한다.

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

이 둘 중 하나를 사용하면 된다. 일반적으로 첫번째 방식을 더 많이 사용한다.

앤서블 준비

SSH

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

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


앤서블 준비

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

앤서블 준비

vim editor

```
$ touch .vimrc
```

```
$ sudo yum install vim -y
```

혹은 centos 8버전 이후를 사용한다면

```
$ sudo yum install vim-ansible
```

앤서블 준비

vim

```
$ vi ~/.vimrc
```

```
" Load local vimrc from project directory (set this in your main vimrc)
" set exrc
```

```
autocmd FileType yml,yaml setlocal ts=2 sts=2 sw=2 expandtab et ai cuc
colorscheme torte
```

```
" Color column 80
highlight ColorColumn ctermbg=6 guibg=lightblue
set colorcolumn=80
```

vim editor

```
$ touch .vimrc
```

```
$ sudo yum install vim vim-enhanced neovim yamllint -y
```

혹은 centos 8버전 이후를 사용한다면

```
$ sudo yum install vim-ansible
```

vim editor

VIM

```
# mkdir -p ~/.vim/pack/git-plugins/start  
# git clone --depth 1 https://github.com/dense-analysis/ale.git  
~/.vim/pack/git-plugins/start/ale
```

NeoVim

```
# mkdir -p ~/.local/share/nvim/site/pack/git-plugins/start  
# git clone --depth 1 https://github.com/dense-analysis/ale.git  
~/.local/share/nvim/site/pack/git-plugins/start/ale
```

.vimrc

```
$ vi ~/.vimrc
```

```
set ts=2
```

```
set sts=2
```

```
set sw=2
```

```
set expandtab
```

```
set number
```

```
let g:indentLine_char = '⋮'
```

```
syntax on
```

```
filetype indent plugin on
```

.vimrc

```
set ruler
```

```
set cursorcolumn
```

```
set foldlevelstart=20
```

```
let g:ale_echo_msg_format = '[%linter%] %s [%severity%]'
```

```
let g:ale_sign_error = '✖'
```

```
let g:ale_sign_warning = '⚠'
```

```
let g:ale_lint_on_text_changed = 'never'
```

앤서블 준비

.vimrc

" General

set cuc # Set cursorcolumn

set exrc " Load vimrc from project
directory

set number " Show line numbers

set numberwidth=4

set showmatch " Highlight matching brace

set visualbell " Use visual bell (no beeping)

앤서블 준비

.vimrc

```
set hlsearch      " Highlight all search results
set smartcase     " Enable smart-case search
set ignorecase    " Always case-insensitive
set incsearch     " Searches for strings incrementally

set autoindent    " Auto-indent new lines
set expandtab      " Use spaces instead of tabs
set shiftwidth=2  " Number of auto-indent spaces
set smartindent   " Enable smart-indent
set smarttab      " Enable smart-tabs
set softtabstop=2 " Number of spaces per Tab
```

앤서블 준비

.vimrc

```
set autoindent " Auto-indent new lines
set expandtab  " Use spaces instead of tabs
set shiftwidth=2    " Number of auto-indent spaces
set smartindent " Enable smart-indent
set smarttab  " Enable smart-tabs
set softtabstop=2    " Number of spaces per Tab
```

앤서블 준비

.vimrc

```
set autoindent      " Auto-indent new lines
set expandtab        " Use spaces instead of tabs
set shiftwidth=2    " Number of auto-indent spaces
set smartindent     " Enable smart-indent
set smarttab        " Enable smart-tabs
set softtabstop=2   " Number of spaces per
Tab
```

앤서블 준비

.vimrc

"" Advanced

set ruler " Show row and column ruler
information

set undolevels=1000 " Number of undo levels

set backspace=indent,eol,start " Backspace
behaviour

set nowrap

앤서블 준비

.vimrc

짧은 옵션은 다음과 같다.

```
filetype plugin indent on  
autocmd FileType yaml  
set sw=2 et ts=2 indent=yaml st=2
```

앤서블 준비

nano

```
$ touch ~/.nanorc
```

```
$ sudo yum install nano
```

```
$ nano ~/.nanorc
```

앤서블 준비

nano

```
# Supports `YAML` files
syntax "YAML" "\.ya?ml$"
header "^(---|===)" "%YAML"
```

Keys

```
color magenta "^\\s*[$A-Za-z0-9_-]+\\:"
color brightmagenta "^\\s*@[\\$A-Za-z0-9_-]+\\:"
```

Values

```
color white ":\s.+ $"
```

앤서블 준비

nano

Booleans

```
icolor brightcyan " (y|yes|n|no|true|false|on|off)$"
```

Numbers

```
color brightred " [[:digit:]]+(\. [[:digit:]]+)?"
```

Arrays

```
color red "\[" "\]" ":\s+[|>]" "^\/s*- "
```

Reserved

```
color green
"(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str)
"
```


앤서블 준비

nano

Comments

color brightwhite "#.*\$"

Errors

color ,red ":\w.+\$"

color ,red ":' .+\$"

color ,red ":' .+\$"

color ,red "\s+\$"

앤서블 준비

nano

```
## Non closed quote  
color ,red "['\"]^['\"]]*$"
```

```
## Closed quotes  
color yellow "['\"].*['\"]"
```

```
## Equal sign  
color brightgreen ":( |$)"  
set tabsize 2  
set tabtospacs
```

앤서블 문법

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

기본 명령어 키워드

YAML

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

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

기본 명령어 키워드

YAML

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

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

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

기본 명령어 키워드

YAML

그래서 YAML상단에는 다음과 같은 형태로 키워드 명령어를 사용한다.

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

기본 명령어 키워드

YAML

- name: 이 플레이북은 웹 서버 설치 및 구성을 합니다.

hosts: all

tasks:

- name: install httpd package

package:

name: httpd

모듈

모듈 변수

작업 이름

함수(function)

기본 명령어 키워드

YAML

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

전역 키워드 선언

예제

become:

remote_user:

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.

기본 명령어 키워드

tasks:

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

- name

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

YAML

sample-yaml.yaml 파일 참고

좀 더 고급 자동화를 위한 앤서블 구성

intermedia of ansible

앤서블 ad-hoc 구성 및 linux-system-role 기반으로 설정 구현

ansible system role and roles

앤서블 모듈 및 패키지

MODULE CASE

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

## MODULE DOCUMENT

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**



**ad-hoc**

# 앤서블 문법

## ad-hoc

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 문법

## ad-hoc

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

애드혹 명령어

```
$ 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

## INVENTORY

앤서블 인벤토리는 다음과 같은 형식을 가지고 있다.

**[인벤토리 이름]**

<호스트>

**[인벤토리 이름:children]**

<그룹이름>

# 앤서블 인벤토리

## INVENTORY

인벤토리는 위의 내용을 기준으로 다음과 같은 내용을 가지고 있다.

- 호스트 이름
- 아이피 주소
- 호스트에서 사용하는 변수

# 앤서블 인벤토리

## **ansible.cfg**

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

## INVENTORY

```
mkdir ansible
```

```
cd ansible
```

```
nano hosts
```

```
[web]
```

```
localhost
```

```
[db]
```

```
localhost
```

# 앤서블 인벤토리

## INVENTORY

인벤토리 파일 기반으로 실행 시 다음과 같이 실행한다.

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

# 앤서블 설정

Example of the ansible.cfg

```
$ cat ansible.cfg
```

```
[defaults]
```

```
inventory = hosts
```

# 앤서블 플레이북

ping.yaml

---

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

# 앤서블 플레이북

## 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

## 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

## 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

## 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

hostname.yaml

```
- hosts: all
 tasks :
 - name: change to a hostname
 hostname:
 name: www.example.com
```

```
ansible-playbook hostname.yaml
hostnamectl
```

# 통합

## www\_service.yaml

- hosts: all  
  tasks :
  - name: change to a hostname  
  hostname:  
    name: [www.example.com](http://www.example.com)
- name: install the postfix package  
  yum:  
    name:
  - postfix
  - httpd
  - mariadb-server  
  state: present

# 통합

`www_service.yaml`

- name: make a text file for httpd service
- copy:
- content: "Hello World My Ansible Service"
  - dest: /var/www/html/index.html

# 통합

## www\_service.yaml

- name: start and enable to a httpd, mariadb-server and postfix service  
systemd:
  - name: "{{ item }}"
  - state: started
  - enabled: yes
- loop:
  - httpd
  - mariadb
  - postfix

# 통합

## www\_service.yaml

혹은 아래와 방식으로 가능. 다만, "- hosts: all", "tasks:"를 제거해야 한다.

- hosts: all
- tasks:
  - include\_tasks: hostname.yaml
  - include\_tasks: package.yaml
  - include\_tasks: html.yaml
  - include\_tasks: service.yaml