# 표준 리눅스 장애처리

엠토스

# DAY 1

# DAY 1

systemd에 통합된 시스템 영역

# systemd

현재 모든 리눅스 배포판은 systemd기반으로 구성이 되어 있다. Systemd는 2010년 그때 당시 레드햇 직원인 Lennart Poettering, Kay Sievers, 두 명이 PID 1번에 대해서 다시 생각을 하게 되었고, 이를 통해서 systemd가 기존의 init, up-start를 대신하게 되었다.

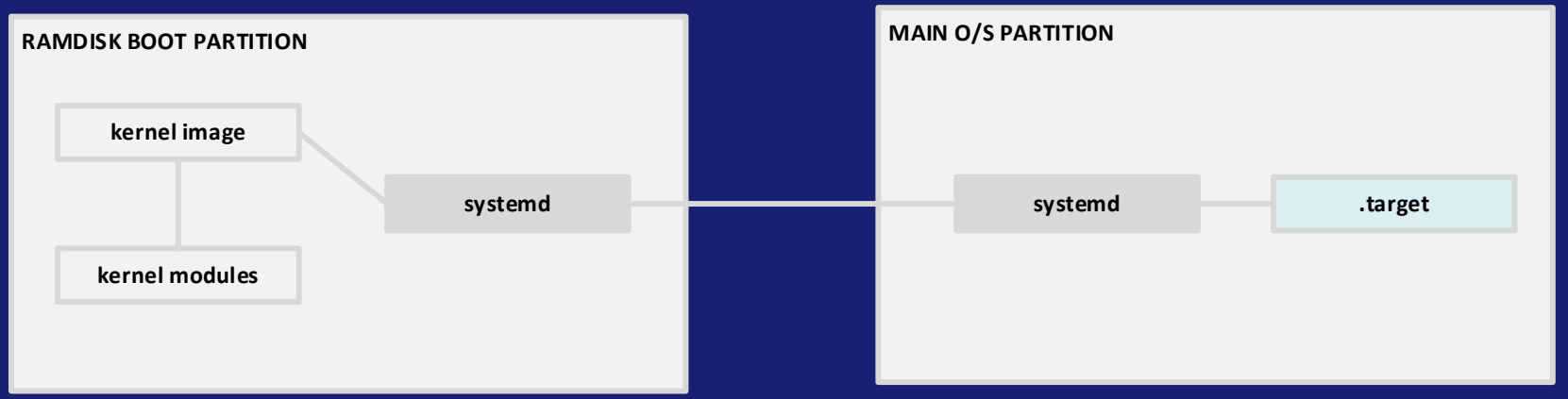
초기 도입은 레드햇이 지원하고 있는 페도라 프로젝트를 통해서 도입하게 되었고, 현재는 모든 오픈소스 배포판 및 사용 운영체제에서 사용하고 있다. systemd에서는 다음과 같은 부분은 통합이 되어 있다.

1. 시스템 부팅 영역
2. 서비스 관리 영역
3. 시스템 유틸리티

# 시스템 부팅 영역

시스템 부팅 영역은 이전에 initramfs혹은 ramdisk라고 부르던 영역이 부팅 영역에 완전히 통합이 되었다. 이전, LILO혹은 GRUB에서는 램 디스크가 없어도 bzimage형태(모듈 포함)로 부팅이 가능하였다. 하지만, systemd에서 더 이상 **작은 이미지(vmlinuz), 큰 이미지(bzimage)**를 구별없이 사용을 하고 있다.

결국, 부팅 영역은 램 디스크를 통해서 기본적인 초기화를 하고, 그 이후 비벗팅(pivoting)를 통해서 정상적으로 O/S부팅이 진행이 된다.



# 서비스 영역

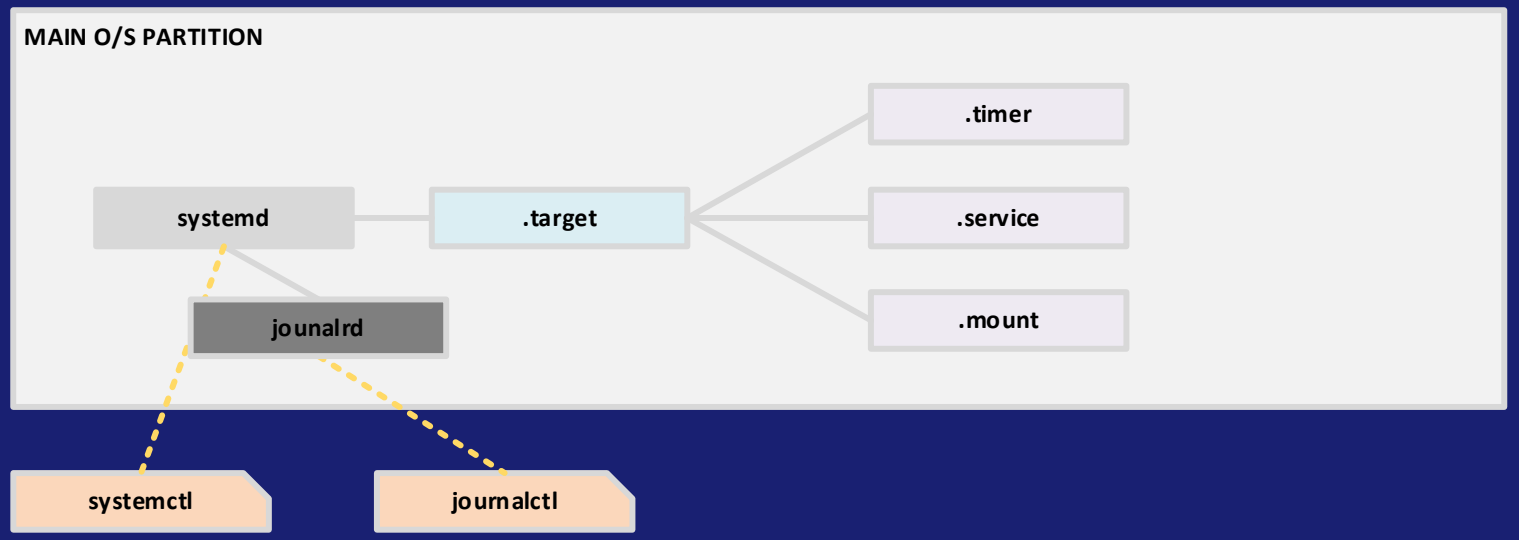
이전 init에서는 서비스 영역은 다음처럼 분리가 되어 있었다.

1. 서비스를 위한 서비스 스크립트 및 유틸리티
2. 시스템 로그를 확인하기 위한 syslogd 및 텍스트 프로세싱 도구

기존 시스템에서 로그를 확인하기 위해서는 cat, grep, awk, sed와 같은 도구를 사용하여 정보를 가공하였지만, 지금은 systemd에서 제공하는 systemd-journald를 통해서 조회가 가능하다. 또한, 앞으로 모든 리눅스 배포판은 호환성으로 syslogd를 제공하지만, 더 이상 주요 시스템 로깅 데몬으로 사용하지 않는다.

서비스 부분은 이전에 쉘 스크립트로 관리가 되었던 init.d는 INI형태로 변경이 되었으며, 자원 및 유닛 관리는 systemd의 관린 명령어인 systemctl명령어를 통해서 관리하게 된다.

# 서비스 영역


MAIN O/S PARTITION
.timer
systemd
.target
.service
jounalrd
.mount
systemctl
journalctl

# 시스템 부분

몇몇 시스템 명령어는 systemd에 통합이 되었다.

- shutdown
- reboot
- halt
- poweroff

위의 명령어들은 systemd에 함수로 통합이 되었다. 이는 아래 깃헙 소스코드 주소에서 확인이 가능하다.

https://github.com/systemdaemon/systemd/blob/master/src/linux/kernel

# 서비스 영역 확인하기

서비스 영역을 확인하기 위해서 다음과 같이 확인한다.

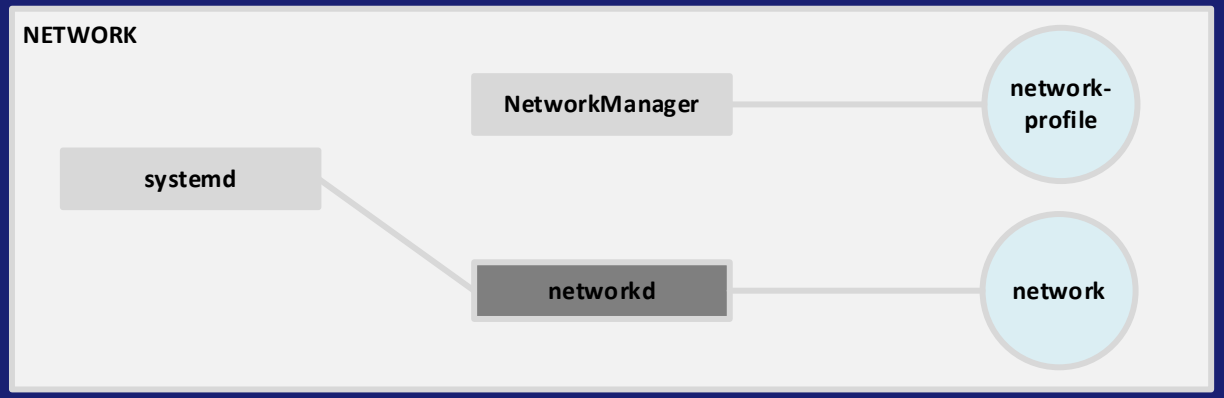
```
# systemctl list-unit-files
# systemctl status
# /usr/lib/systemd/
# /usr/lib/systemd/system
# /usr/lib/systemd/user
# /etc/systemd/system
# /etc/systemd/user
```

# 네트워크

모든 리눅스 배포판은 다음과 같은 기능 기반으로 네트워크 설정 기능을 제공하고 있다.

1. NetworkManager
2. systemd-networkd
3. LSB Network Script

이전 모든 리눅스는 각기 네트워크 관리자 시스템 예를 들어 wicked, NetworkManager, netplan를 사용 하였지만, 현재는 NM으로 이전 및 systemd-networkd으로 통합이 되고 있다. systemd기반에서는 모든 네트워크 구성 및 설정은 systemd-networkd기반으로 구성이 된다.



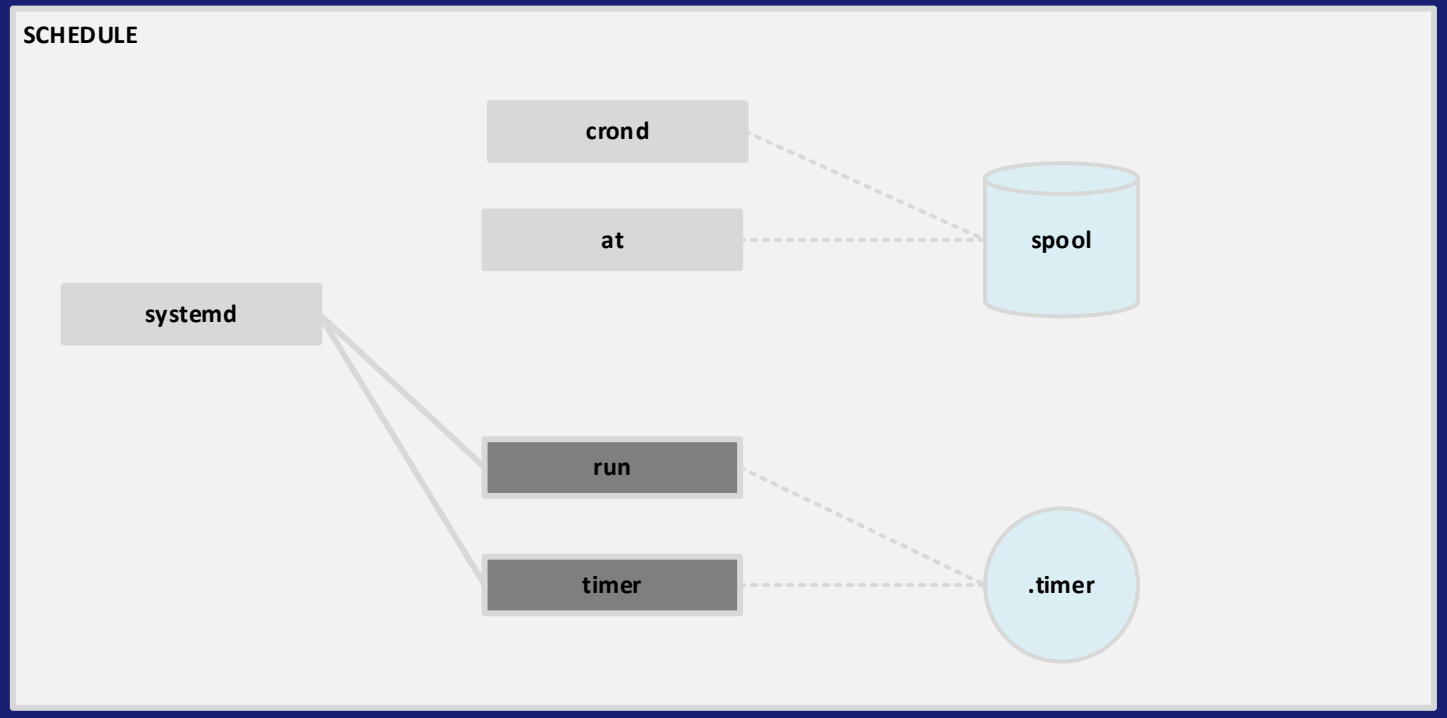
# 네트워크

현재 대다수 리눅스가 지원하는 네트워크 설정은 대략 다음과 같이 된다.

1. NetworkManager
2. /etc/sysconfig/network-scripts
3. /etc/hostname
4. /etc/systemd/networkd

# 예약작업

모든 리눅스는 현재 at, crond(anacrond)기반으로 예약 작업을 수행하고 있다. 하지만, systemd로 시스템 블록이 통합이 되면서 대다수 기존 스크립트 혹은 작업들은 systemd-timer 자원으로 통합이 되었다.



# 예약작업

systemd-timer는 아래 명령어로 간단하게 확인이 가능하다. 현재 모든 리눅스 배포판은 더 이상 crond, anacrond를 사용하지 않는다.

```
# systemctl list-unit-files -t timer
# ls -l /etc/cron.*
```

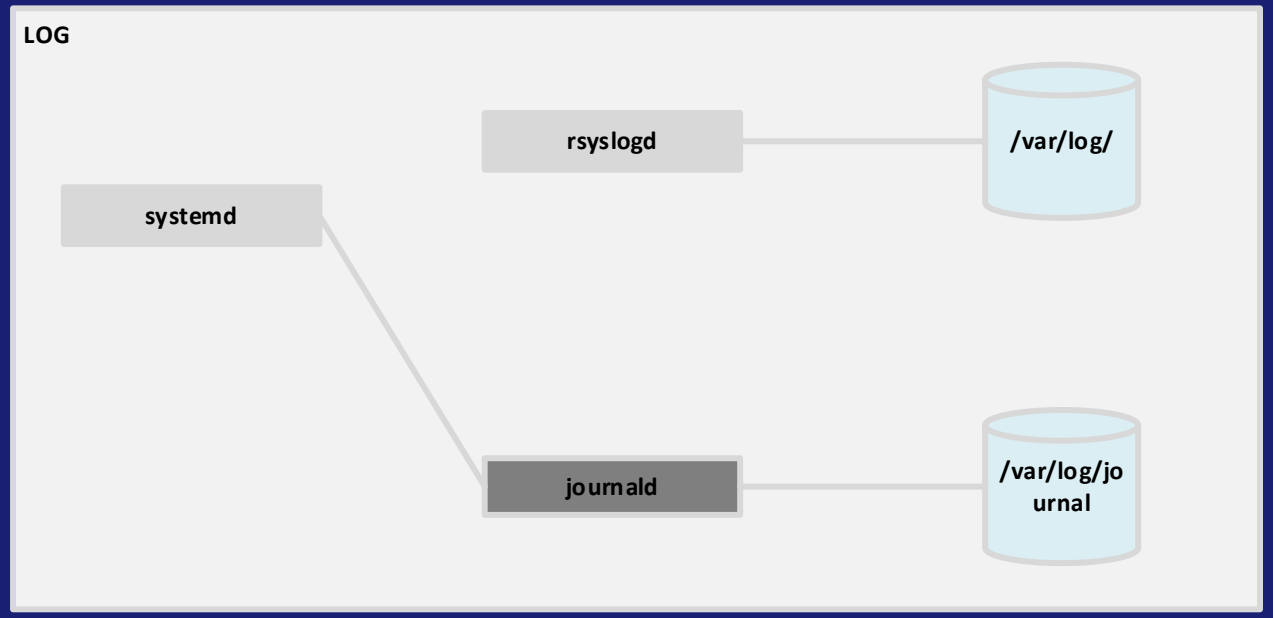
자주 사용하지는 않지만, at명령어로 사용하던 부분은 다음처럼 변경이 되었다.

```
# systemd-run date; systemd-run --on-active=30 --timer-property=AccuracySec=100ms
/bin/touch /tmp/foo
```

# 로깅 및 로그

로깅 대몬 및 로그 서비스는 **syslogd**를 그대로 사용하고 있으나, systemd 기반을 사용하는 시스템은 systemd-journald가 주요 로깅 시스템으로 동작하고 있으며, 호환성으로 syslogd로깅도 같이 지원하고 있다.

레드햇 리눅스 기준으로 RHEL 9버전부터는 더 이상 모든 로그를 syslogd로 지원하지 않으며,  journald기반으로 로깅을 제공한다. 이를 관리 및 조회하기 위해서 journalctl명령어를 통해서 확인이 가능하다.



# 로깅 확인하기

바이너리 로깅 기록은 아래 명령어로 확인이 가능하다.

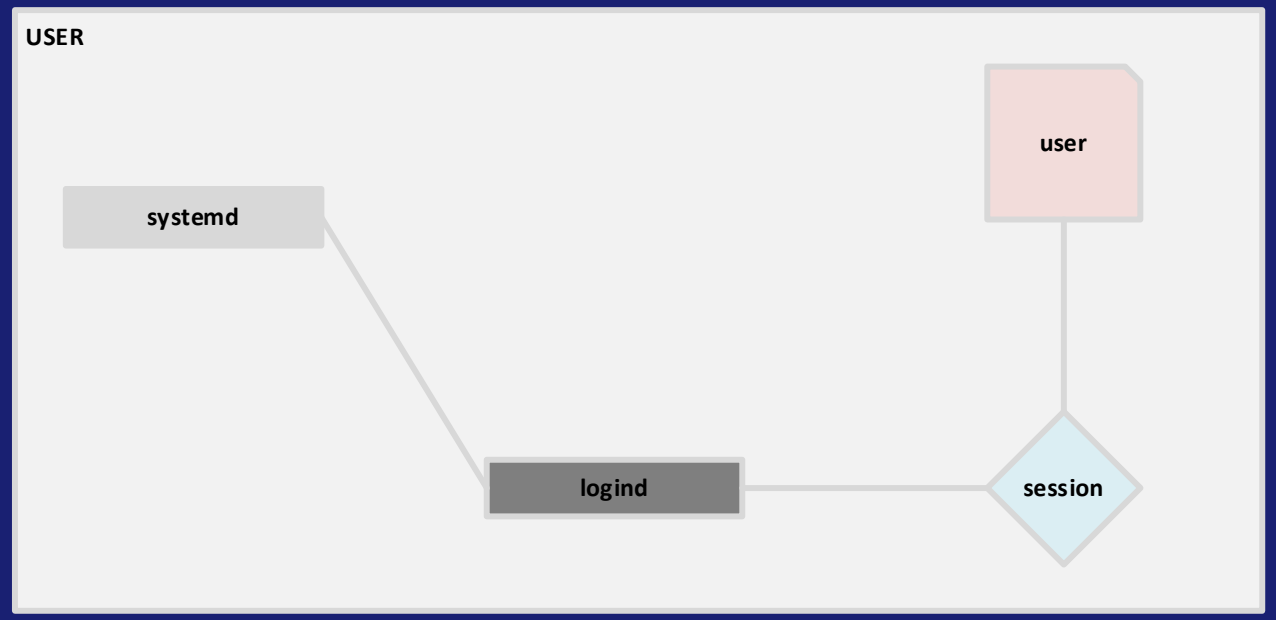
```
# journalctl -u httpd.service -fl -perr
```

커널 및 부팅 로그도 이미 데이터베이스 전환이 되어 있기 때문에 명령어로 조회 및 확인이 가능하다.

```
# journalctl -b
# journalctl -k
# journalctl --list-boots
```

# 사용자 관리

사용자는 이전과 동일하게 useradd, userdel, usermod를 통해서 관리가 가능하다. 다만, systmed에서는 systemd-logind를 통해서 세션을 관리하게 된다. 사용자 관련된 세션 관리 및 제어는 logind 대몬을 통해서 수행해야 한다.



# DAY 1

램 디스크를 통한 운영체제 복구

# 램 디스크

systemd로 변경이 되면서, 이전에 사용하던 램 디스크의 기능이 확장이 되었다. 먼저, 이 기능을 이해하기 전에 램 디스크(ramdisk)에 대해서 잠깐 이야기 한다.

램 디스크는, 본래 리눅스 커널이 부팅 시 모든 커널 오브젝트, 즉 모듈을 커널 이미지에 가지고 있으면 부팅이 느리기 때문에 좀 더 빠르게 하기 위해서 램 디스크 이미지를 사용하게 되었다. 램 디스크는 기본적으로 커널에서 추가적으로 필요한 기능을 메모리 영역에 불러와서, 커널 이미지가 빠르게 실행 후, 추가 기능을 램 디스크를 통해서 불러온다.

| 커널 형식 | 설명 |
| --- | --- |
| vmlinuz | 제일 작은 크기의 커널. 보통 램 디스크 기반으로 구성 시 많이 사용한다. 빠르게 부팅이 되지만, 램 디스크와 같은 기능을 통해서 추가적인 모듈을 제공하지 않으면, 올바르게 부팅이 되지 않을 수 있다. |
| bzimage | 모든 기능을 커널 이미지에 포함 시킨다. 부팅은 느리지만, 모든 기능이 커널에 포함이 되어 있기 때문에, 부팅 시 문제 발생 가능성이 낮다. 다만, 디버깅이 어려운 부분이 있다. |

# 램 디스크

램 디스크는 보통 두 가지 형식으로 이름을 많이 사용한다.

1. ramdisk
2. initramfs

두 개의 차이점은 크게 없지만, 구체적으로 다음과 같이 역할이 나누어 진다. 현재 사용중인 systemd는 initramfs를 통해서 부팅을 진행한다.

| 램 디스크 | 설명 |
| --- | --- |
| ramdisk | 램 디스크는 말 그대로 메모리에 디스크를 만들어서 사용한다. 이전에 vmlinuz커널 이미지에서 추가 기능을 불러 올 때 사용 하였다. ramdisk의 다른 이름은 initrd라는 이름을 사용한다. |
| initramfs | 램 디스크와 동일하나, 기존 램 디스크에 부팅 기능이 포함된 디스크 이미지를 말한다. 이전 System V는 init, systemd는 systemd를 가지고 있다. 또한, initramfs는 ramdisk와 다르게 cpio arhcive형태로 구성이 되어 있다. |

# 램 디스크

램 디스크 디버깅은 보통 다음과 같은 용도로 사용한다.

1. 시스템 부팅 영역에 문제가 발생 하였을 때.
2. 커널 모듈이 올바르게 동작하지 않을 때.
3. 커널 이미지가 올바르게 동작하지 않을 때.

일반적으로 램 디스크 사용 시 rd.break 옵션을 많이 사용한다. 하지만, systemd에서는 더 많은 옵션을 디버깅 옵션으로 제공한다.

https://man7.org/linux/man-pages/man7/dracut.cmdline.7.html

```
[root@rocky boot]# file initramfs-5.14.0-427.13.1.el9_4.x86_64.img
initramfs-5.14.0-427.13.1.el9_4.x86_64.img: ASCII cpio archive (SVR4 with no CRC)
[root@rocky boot]# file vmlinuz-5.14.0-427.13.1.el9_4.x86_64
vmlinuz-5.14.0-427.13.1.el9_4.x86_64: Linux kernel x86 boot executable bzImage, version 5.14.0-427.13.1.el9_4.x86_64 (mockbuild@iad1-prod-build001.bld.equ.rockylinux.org) #1 SMP PREEMPT_DYNAMIC Wed May 1 19:11:28 UT, RO-rootFS, swap_dev 0xC, Normal VGA
```

# 램 디스크

램 디스크에서 많이 사용하는 옵션은 다음과 같다.

| 옵션 | 설명 |
| --- | --- |
| rd.break | 램 디스크에서 디버깅이나 혹은 다른 작업이 필요한 경우 이 옵션을 사용한다. 이 옵션 이외 다양한 옵션을 제공하고 있다. 하위 램 디스크 옵션으로 다음을 지원하고 있다. *cmdline\|pre-udev\|pre-trigger\|initqueue\|pre-mount\|mount\|pre-pivot\|cleanup* |
| rd.udev.info | 램 디스크에서 동작중인 udev의 정보를 출력한다. 보통 일반적으로 디스크 및 네트워크 장치 정보를 출력한다. |
| rd.lvm=0 | 램 디스크에서 사용하는 LVM2기능을 중지한다. |
| rd.luks=0 | 램 디스크에서 사용하는 luks기능을 중지한다. |
| rd.multipath=0 | 램 디스크에서 사용하는 멀티패스 기능을 중지한다. |
| biosdevname=0 | 델 바이오스 이름 기능을 중지한다. |

# 램 디스크

램 디스크는 직접 O/S영역에 접근은 불가능하며, 두 가지 지점에서 접근이 가능하다.

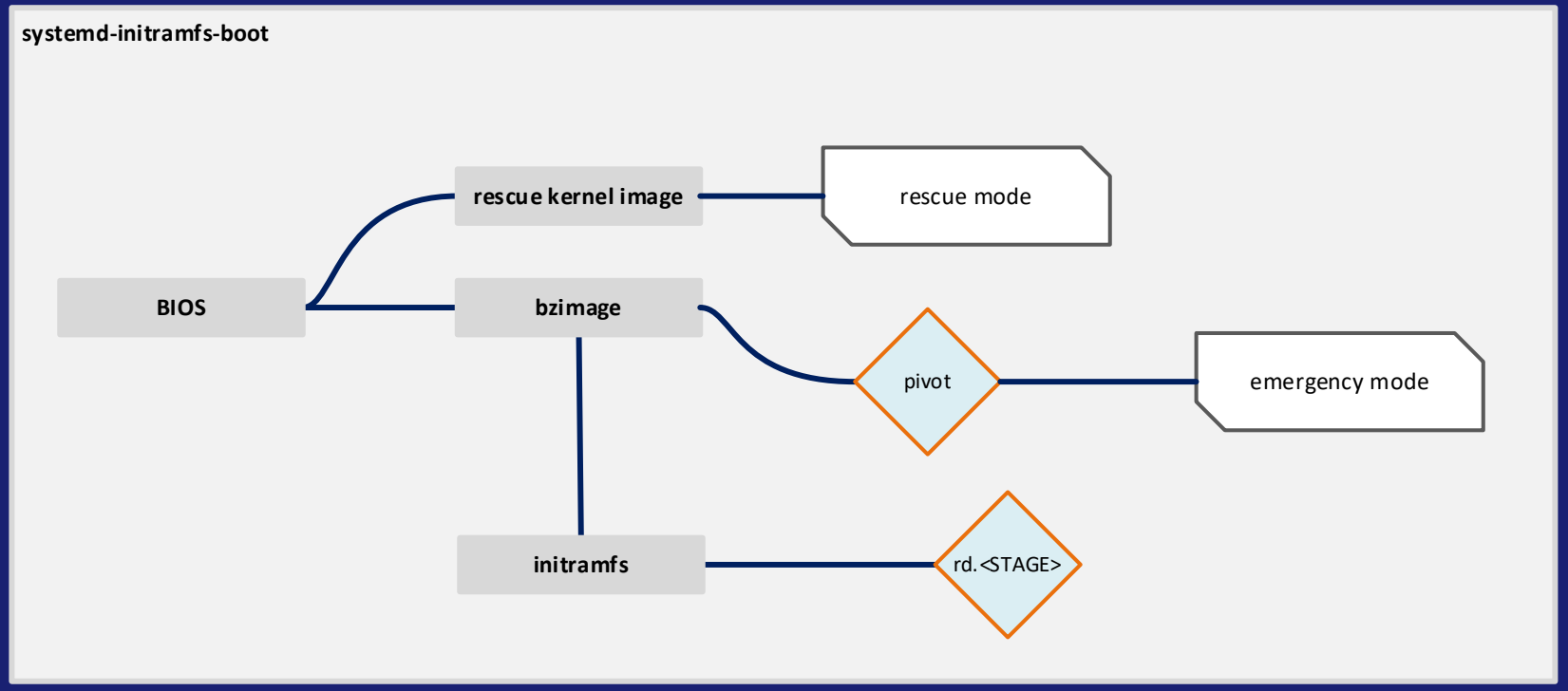
1. break
2. rescue mode
3. emergency mode

break는 램 디스크에서 중지 및 OS영역을 읽기 전용으로 마운트 한다. 하지만, 옵션에 따라서 pivot하기 전후로 달라진다. rescue mode는 램 디스크에서 OS영역에 발생한 문제를 해결을 위해서 사용한다. 기본적으로 break와 동일하지만, rescue mode에서 사용하는 커널 이미지는 좀 더 많은 커널 모듈을 램 디스크에 포함이 되어 있다.

emergency mode는 ramdisk에서 OS영역으로 전달 후, 특정한 문제가 발생하여 root shell로 drop이 된 상태이다.

이런 경우는, 램 디스크에서 OS영역으로 올바르게 pivoting이 되었지만, 특정한 문제로 시스템 부트업이 중지가 된 상태이다. 보통 이 경우는 간단한 패키지 문제나 systemd의 유닛 설정 문제로 많이 발생한다.

# 램 디스크


systemd-initramfs-boot
BIOS
rescue kernel image
rescue mode
bzimage
pivot
emergency mode
initramfs
rd.<STAGE>

# 램 디스크 기능 확인

램 디스크 내용은 다음과 같이 구성이 되어 있다.

```
# mkdir ramdisk
# cp /boot/initramfs-5.14.0-427.13.1.el9_4.x86_64.img .
# cpio -ivF initramfs-5.14.0-427.13.1.el9_4.x86_64.img
# lsinitrd --unpack initramfs-5.14.0-427.13.1.el9_4.x86_64.img
```

램 디스크는 시스템 블록에서 장치 및 설정파일이 변경이 되면, 램 디스크에 반영이 된다. 반영은 systemd에서 확인 후 램 디스크에 추가하는데, 이때 사용하는 도구가 dracut이라는 도구이다.

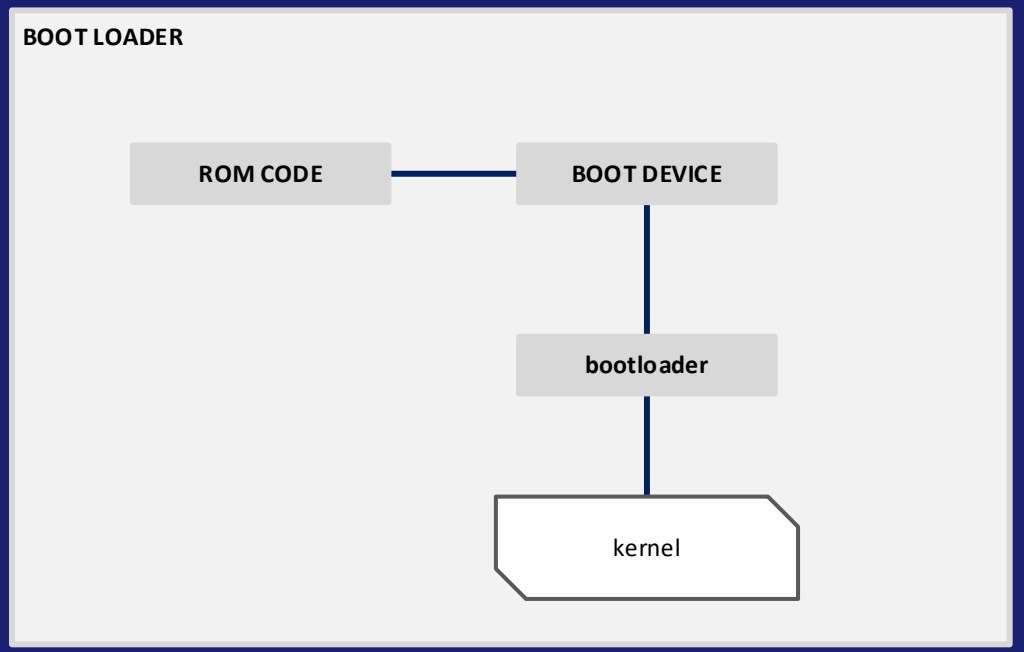
```
# ls -al /etc/dracut.conf
-rw-r--r--. 1 root root 117 Apr  8 10:06 /etc/dracut.conf
# ls /lib/dracut/
```

# DAY 1

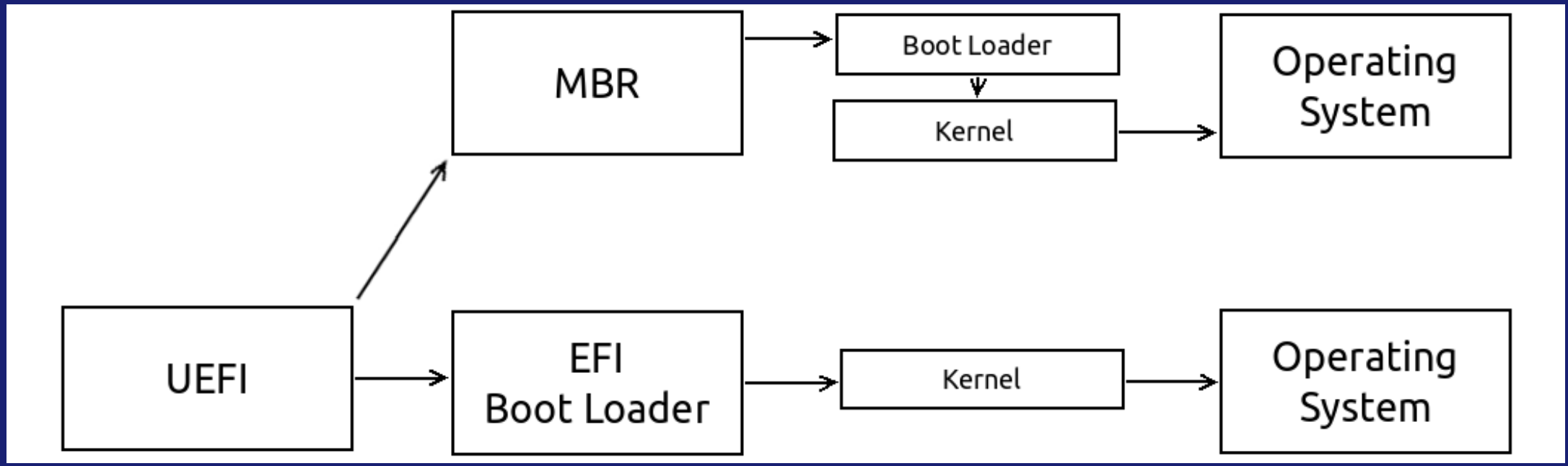
grub2, bootrec

# GRUB2

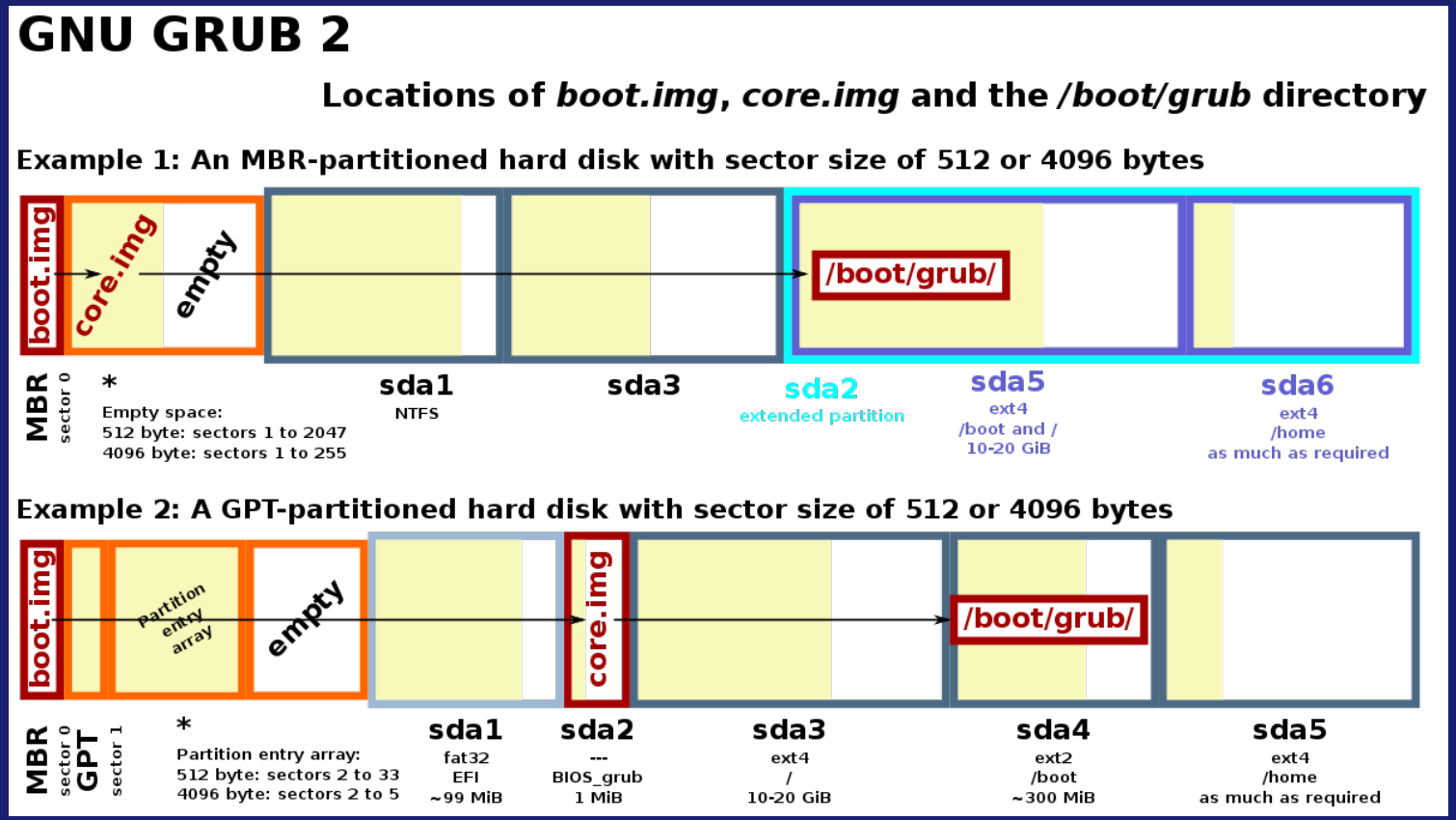
GRUB2은 현재 대다수 리눅스에서 사용하는 커널 부트로더(kernel bootloader)이다. 부트로더의 역할은 바이오스에 간단한 시작 프로그램을 올린 후, 메모리에 시스템에서 사용할 커널 이미지를 불러와 A.OUT형태로 프로그램을 실행한다.



# BOOTLOADER


MBR
Boot Loader
Kernel
Operating System
UEFI
EFI Boot Loader
Kernel
Operating System

# BOOTLOADER


GNU GRUB 2
Locations of boot.img, core.img and the /boot/grub directory
Example 1: An MBR-partitioned hard disk with sector size of 512 or 4096 bytes
boot.img
core.img
empty
/boot/grub/
MBR
sector 0
*
Empty space:
512 byte: sectors 1 to 2047
4096 byte: sectors 1 to 255
sda1
NTFS
sda3
sda2
extended partition
sda5
ext4
/boot and /
10-20 GiB
sda6
ext4
/home
as much as required
Example 2: A GPT-partitioned hard disk with sector size of 512 or 4096 bytes
boot.img
Partition entry array
empty
core.img
/boot/grub/
MBR
sector 0
GPT
sector 1
*
Partition entry array:
512 byte: sectors 2 to 33
4096 byte: sectors 2 to 5
sda1
fat32
EFI
~99 MiB
sda2
---
BIOS_grub
1 MiB
sda3
ext4
/
10-20 GiB
sda4
ext2
/boot
~300 MiB
sda5
ext4
/home
as much as required

# GRUB2

grub2는 본래 LILO를 대체하기 위한 차세대 부트로더 이었으나, 점점 grub2도 기존 LILO와 동일하게 기능이 추가가 되면서 점점 복잡하게 되었다. 특히 MBR에서 EFI로 변경이 되면서, 사용하기가 더 어렵게 되었다. 이러한 이후로 systemd에서는 더 이상 grub2를 사용하지 않고 bootrec로 전환을 하고 있다.

현재 사용중인 grub2의 관리 명령어는 다음과 같다.

grubby

grub2에서 사용하는 설정파일을 구성하는 명령어. 생성되는 파일 위치는 /boot/grub2/grub2.cfg, /boot/efi/EFI/Redhat/grub.cfg에 생성이 된다. 이 파일이 생성되는 위치는 리눅스 배포판 별로 조금식 다를 수 있기 때문에 배포판 별 매뉴얼 확인이 필요하다.

# GRUB2-GRUBBY

grubby를 통해서 커널을 제어하기 위해서 아래와 같이 명령어를 사용한다.

```
# grubby --update-kernel=/boot/vmlinuz-5.14.0-427.13.1.el9_4.x86_64
# grubby --remove-kernel=/boot/vmlinuz-5.14.0-427.13.1.el9_4.x86_64
# grubby --set-default=/boot/vmlinuz-5.14.0-427.13.1.el9_4.x86_64
# gruuby --info=ALL | grep -E "^kernel|^index"
# grubby --set-default-index=1
# grubby --default-title
```

수동으로 커널을 추가하는 경우, 다음 명령어로 추가가 가능하다. 생성된 앤트리(entry)는 다음 디렉터레에서 확인이 가능하다.

```
# grubby --add-kernel=new_kernel --title="entry_title" --initrd="new_initrd" --copy-default
# ls /boot/loader/entries/
2609d9fb46664d0c8ee8cb7d2bdab610-0-rescue.conf  2609d9fb46664d0c8ee8cb7d2bdab610-5.14.0-427.13.1.el9_4.x86_64.conf
```

# GRUB2-GRUBBY

커널에 등록된 변수를 변경하기 위해서 다음과 같이 설정한다. 기본적인 명령어는 다음과 같다.

```
# grubby --update-kernel=current_kernel --remove-args="kernel_args"
# grubby --update-kernel=current_kernel --args="kernel_args"
```

위의 명령어를 커널에 적용하면 다음과 같이 사용이 가능하다.

```
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --args="ipv6.disable=1"
# grep ipv6 /boot/loader/entries/2609d9fb46664d0c8ee8cb7d2bdab610-$(uname -r).conf
> /root rd.lvm.lv=rl/swap rhgb quiet ipv6.disable=1
# grubby --update-kernel=/boot/vmlinuz-$(uname -r) --remove-args="ipv6.disable=1"
# grep ipv6 /boot/loader/entries/2609d9fb46664d0c8ee8cb7d2bdab610-$(uname -r).conf
```

모든 인자 값을 제거하기 위해서 아래와 같이 명령어를 실행한다.

```
# grubby --update-kernel=ALL --args="kernel_args"
```

# GRUB2-GRUBBY

SELinux경우에도 커널 인자 값을 통해서 동작 여부에 대해서 설정이 가능하다. 다만, disabled으로 옵션 변경 시 사용을 권장한다.

```
# grubby --update-kernel ALL --args selinux=0
# grubby --update-kernel ALL --remove-args selinux
```

# grub2-install

부트로더가 손상이 되거나 혹은 올바르게 동작하지 않는 경우  grub2-install명령어를 통해서 재구성이 가능하다.

```
# lsblk
# grub2-install /dev/sda
```

이미 구성된 시스템에 부트로더가 이미 구성이 되었는지 확인이 어려운 경우, 다음과 같이 명령어를 실행한다.

```
# dnf install -y grub2-efi-x64
# grub2-install --target=x86_64-efi --efi-directory=/boot/efi --removable --boot-directory=/boot/efi/EFI --bootloader-id=grub /dev/sda
```

# grub2-mkconfig

grub2에서 사용하는 설정파일을 다시 재생성을 원하는 경우 다음과 같이 명령어를 사용한다.

```
# find / -name grub.cfg -type f -print
> /boot/efi/EFI/rocky/grub.cfg
> /boot/grub2/grub.cfg
# grub2-mkconfig -o /boot/grub2/grub.cfg
```

# bootrec

앞으로 grub2를 대신하여 사용할, systemd으로 통합된 부트로더 시스템. 레드햇 및 대다수 배포판은 이 기능을 활성화가 안되어 있으며, 레드햇 계열 배포판에서 사용하려면 아래와 같이 몇몇 작업을 수행해야 한다.

```
# mkdir -p /usr/lib/systemd/boot/efi
# cp /boot/efi/EFI/rocky/* /usr/lib/systemd/boot/efi/
# dnf --enablerepo=devel install systemd-boot -y
# bootctl install
# reboot
```

앞으로 사용할 부트로더 시스템이기 때문에 미리 학습을 권장한다.

# DAY 1

bootrec

# XFS

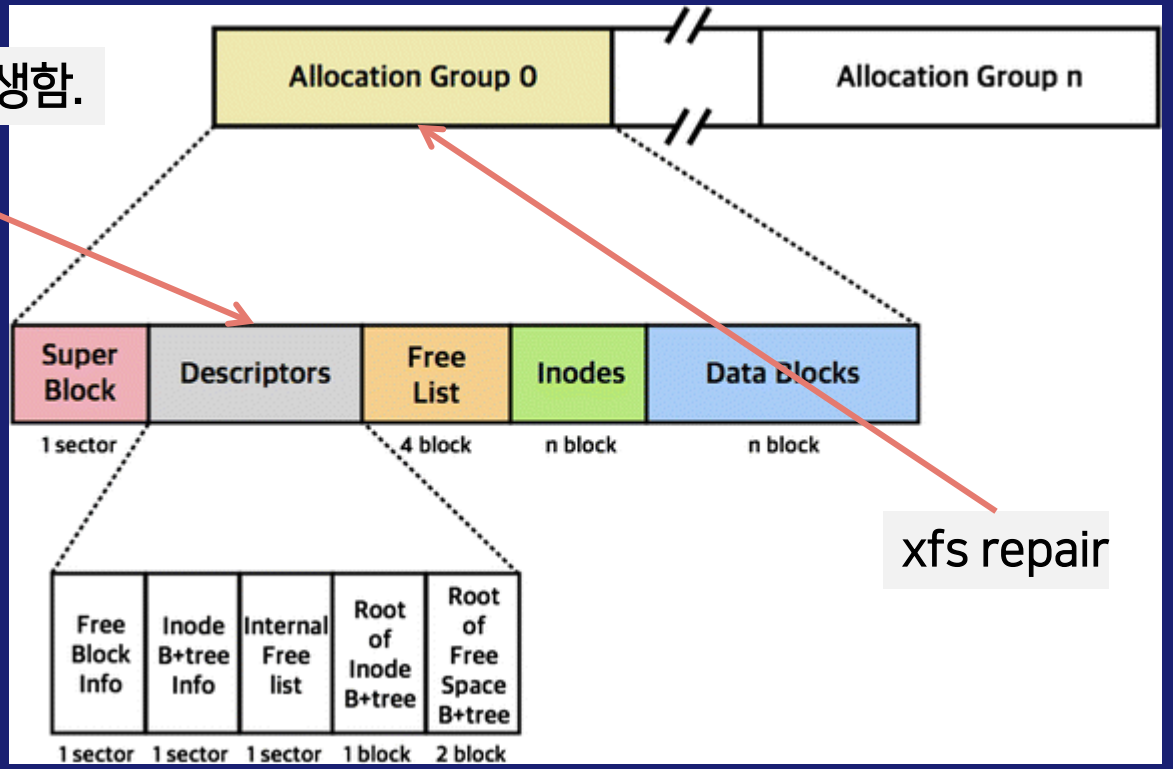
XFS파일 시스템은 이전 ext3/4과 다르게 복구 과정이 쉬우면서 상당히 까다로운 파일 시스템이다. XFS는 본래 SGI IRIX에서 사용하던 파일 시스템을 리눅스로 마이그레이션 하였으며, IRIX 파일 시스템은 고성능에 맞추어서 디자인이 되었기 때문에 다중 접근이 발생하는 경우 파일 손상이 종종 발생 하였다.

이러한 이유로, SGI에서는 XFS의 공유 파일 시스템 버전인 GFS를 만들었으며, 현재 레드햇이 GFS2라는 이름으로 사용하고 있다.

XFS를 복구하기 위해서 다음과 같은 구조에 대해서 간단하게 이해가 필요하다.

# XFS 구조

이 부분에서 자주 오류가 발생함.



# XFS META BACKUP

XFS 파일 시스템 작업을 진행 하기 전, 가급적이면 메타 정보를 백업 후 진행한다. 백업 및 복구하는 방법은 다음과 같다.

```
# xfs_metadump /dev/sdb /root/sdb_block_meta.backup
```

복구 하는 방법은 다음과 같이 실행한다. 문제 없이 복구가 되면, 별도 메시지는 출력하지 않는다.

```
# xfs_mdrestore /root/sdb_block_meta.backup /dev/sdb
```

위의 작업이 완료 후, 파일 시스템에 강제로 손상을 만든다.

# XFS TROUBE MAKER

XFS 파일 시스템에 다루기 위해서 사용중인 OS영역에서는 위험하기 때문에, 가급적이면 블록 장치를 하나 추가 후 작업을 진행한다.

- /dev/sdb

테스트 하기 위해서 아래와 같이 명령어를 수행한다.

```
# mkdir -p /mnt/sdb
# mkfs.xfs /dev/sdb
# mount /dev/sdb /mnt/sdb
# umount /mnt/sdb
# xfs_db -x -c blockget -c "blocktrash -s 1000 -n 300" /dev/sdb
> blocktrash: 0/5 btcnt block 5 bits starting 1609:6 flipped
> blocktrash: 0/18 inode block 1024 bits starting 1748:4 flipped
# mount /dev/sdb /mnt/sdb
> mount: /mnt/sdb: mount(2) system call failed: Structure needs cleaning.
```

# XFS REPAIR

위의 작업이 완료가 되면, 파일 시스템은 잘못된 메타 정보로 인하여 장애가 발생한다.

```
# xfs_db -x -c blockget -c "blocktrash -s 512109 -n 1000" /dev/sdb
```

마지막으로 발생한 이벤트를 확인하기 위해서 다음 명령어를 실행한다.

```
# xfs_logprint /dev/sdb
> Oper (0): tid: b0c0d0d0  len: 8  clientid: LOG  flags: UNMOUNT
Unmount filesystem
```

# XFS REPAIR

파일 시스템 메타 정보가 올바르게 구성이 되어 있는지 다음 명령어로 확인한다.

```
# xfs_info /dev/sdb
Metadata CRC error detected at 0x5618dd7849f0, xfs_agf block 0x8/0x1000
xfs_info: cannot init perag data (74). Continuing anyway.
meta-data=/dev/sdb               isize=512    agcount=4, agsize=8323072 blks
         =                       sectsz=4096  attr=2, projid32bit=1
         =                       crc=1        finobt=1, sparse=1, rmapbt=0
         =                       reflink=1    bigtime=1 inobtcount=1 nrext64=0
data     =                       bsize=4096   blocks=33292288, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0, ftype=1
log      =internal log           bsize=4096   blocks=16384, version=2
         =                       sectsz=4096  sunit=1 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

# XFS REPAIR

메타 정보를 덤프 받아서 확인하려면 다음과 같이 명령어를 실행한다.

```
# xfs_metadump /dev/sdb /root/sdb.metadump
Metadata CRC error detected at 0x55b5689459f0, xfs_agf block 0x8/0x1000
xfs_metadump: cannot init perag data (74). Continuing anyway.
Metadata CRC error detected at 0x55b56895fb80, xfs_agi block 0x10/0x1000
Metadata CRC error detected at 0x55b56895fb80, xfs_agi block /usr/sbin/xfs_metadump:
line 34:  1787 Segmentation fault      (core dumped) xfs_db$DBOPTS -i -p
xfs_metadump -c "metadump$OPTS $2" $1
```

# 복구

복구를 수행하기 위해서 다음 명령어로 실행한다. 처음에는 진짜 문제가 있는지 아래 명령어로 xfs메타 디비를 확인한다.

```
# xfs_repair -n /dev/sdb
```

실제로 문제가 있다고 판단이 되면, 아래와 같이 명령어를 실행한다.

```
# xfs_repair /dev/sdb
```

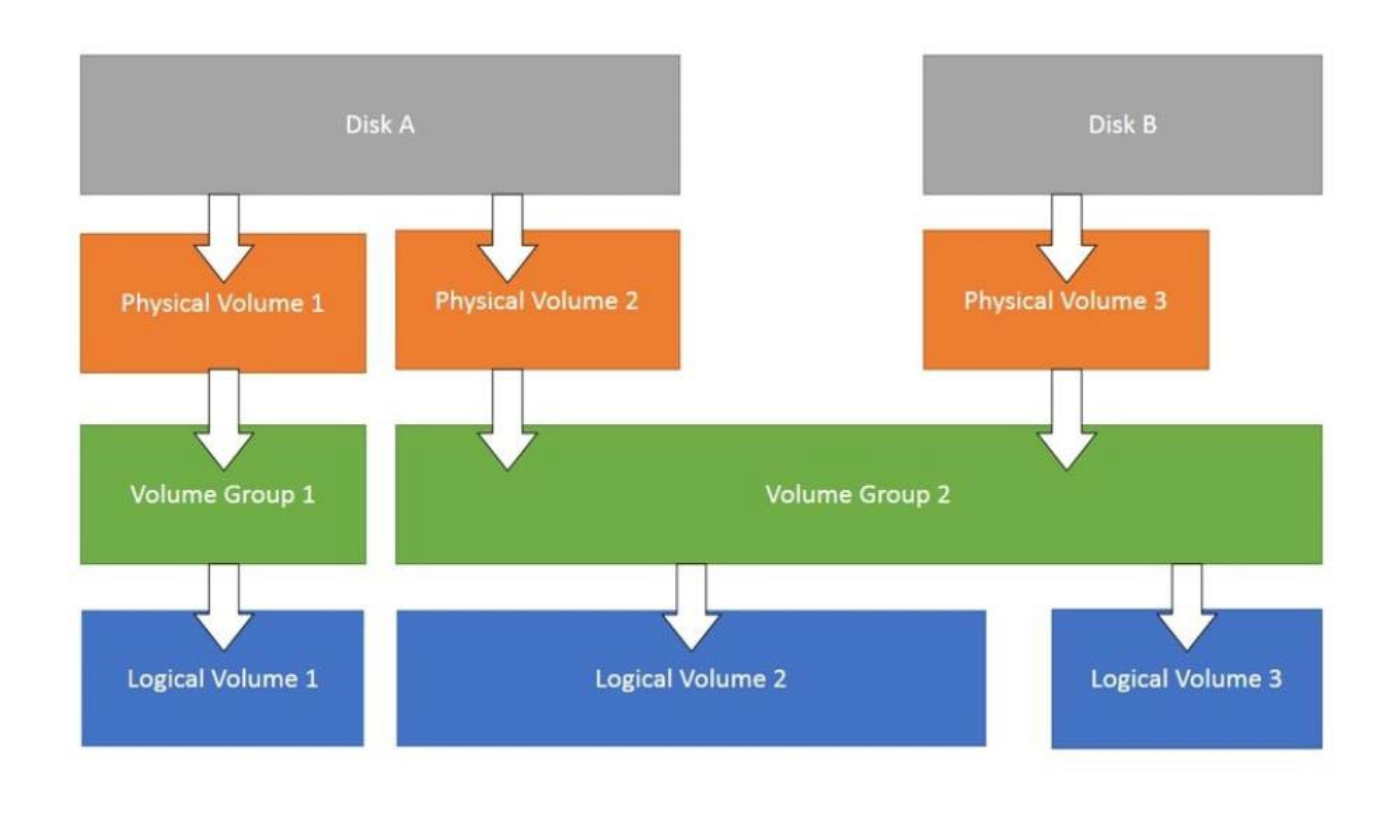
문제 없이 실행이 되면, 마운트가 정상적으로 동작하는지 확인한다.

```
# xfs_repair /dev/sdb
```

# DAY 1

블록장치 용량 확장
LVM2/Stratis

# LVM2


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

# LVM2 명령어

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

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

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

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

# LVM2 볼륨 및 논리 장치 생성

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

1기가 파티션 생성

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

VG에 1기가 전부 할당

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

논리적 디스크 생성

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

# LVM2 볼륨그룹 확장

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

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

추가로 만든 파티션 혹은 디스크를 test-vg에 추가한다.

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

확장된 볼륨 크기만큼 논리 디스크를 확장한다.

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

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

# LVM2 BACKUP

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

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

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

# LVM2 BACKUP/RESTORE

PE영역은 별도로 백업이 되지 않는다. 백업을 하기 위해서 다음과 같이 수동으로 작업을 수행한다.

```
# dd if=/dev/sdc of=/root/lvm_backup_pv.pv bs=1024 count=1
# dd if=/root/lvm_backup_pv.pv of=/dev/sdc bs=1024 count=1
# systemctl restart lvm2-lvmpolld.socket
# pvs
```

혹은 다음과 같이 복구가 가능하다. 다만, PV메타 영역이 완전이 손상이 된 경우, 올바르게 동작이 안될 수 있다.

```
# pvcreate --uuid "FmGRh3-zhok-iVI8-7qTD-S5BI-MAEN-NYM5Sk" --restorefile
/etc/lvm/backup/rhel /dev/sdb
```

# LVM2 PV METADATA

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

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

# LVM2 VG METADATA

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

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

# LVM2 LV METADATA

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

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

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

# LVM2의 미래

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

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

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

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

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

# STRATIS

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

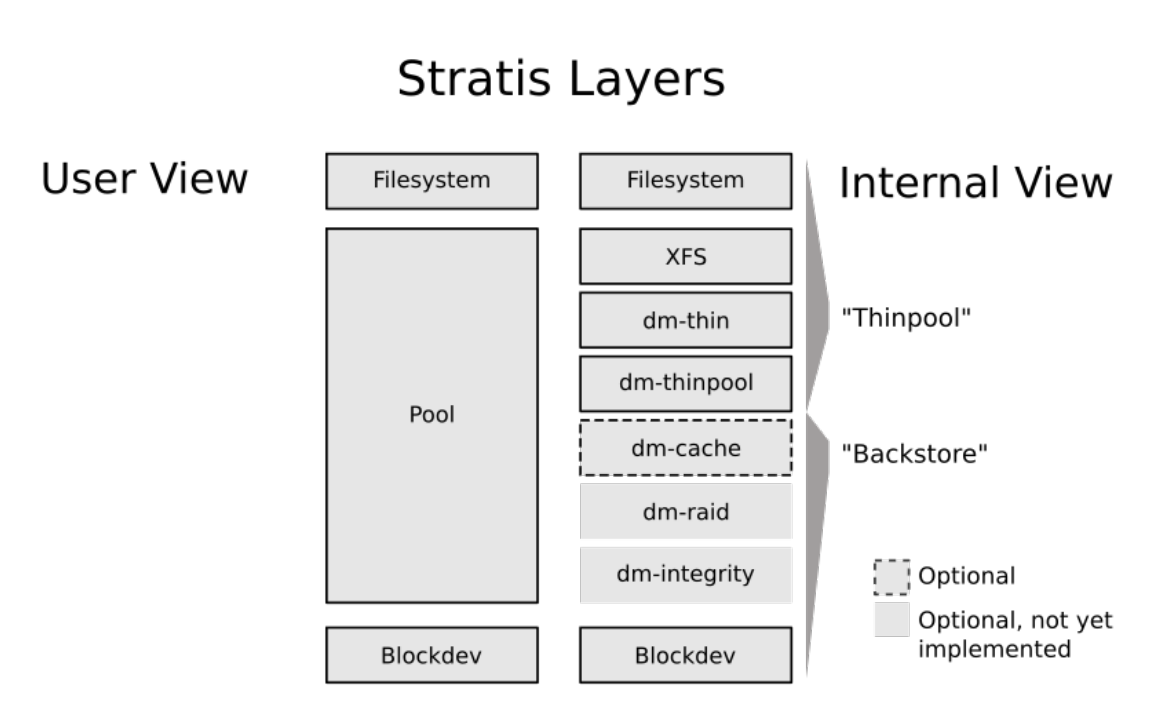
레드햇 계열 배포판에서는 Stratis라는 이름으로 제공하고 있으며, 이를 사용하기 위해서는 레드햇 계열 기준 7버전 이상을 권장한다. 또한, 파일 시스템 및 "Startisd"데몬 버전에 따라서 기능이 다르기 때문에, 가능한 최신 버전 사용을 권장하며, 8버전 이후로 상용 시스템에 사용하기가 적합하다.

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

XFS 기능 링크

# STRATIS

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



# STRATIS POOL

스토리지 풀 도구를 설치한다. Stratis는 Strarisd데몬으로 관리가 되기 때문에, 반드시 데몬 서비스가 동작이 되어야 한다.

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

# STRATIS FILESYSTEM

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

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

# STRATIS POOL EXTEND

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

# 진행 전 준비사항

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

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

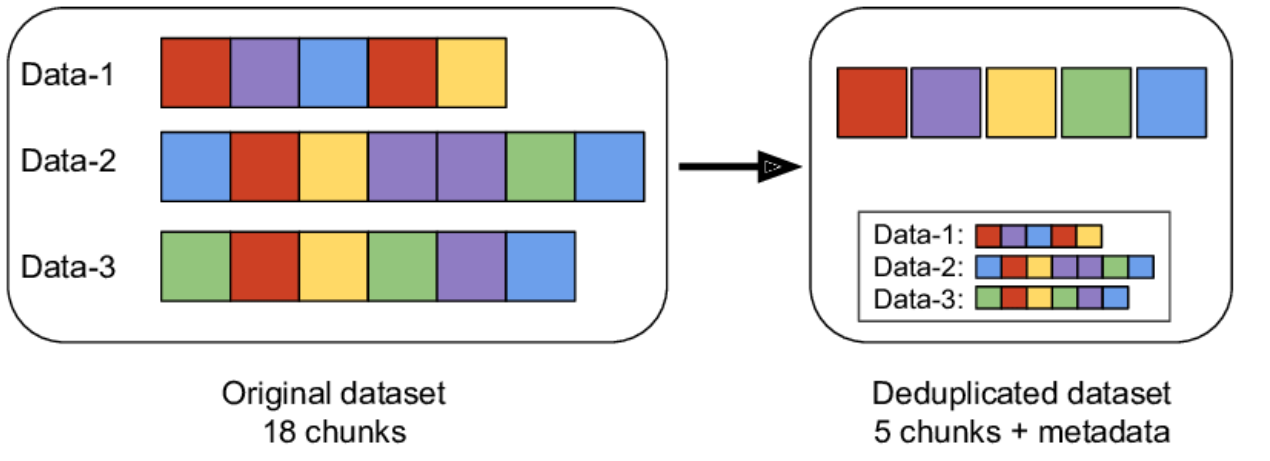
# DUP/RAID/SNAPSHOT

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

# VDO(DE-DUPLICATE OPTIMIZATION)

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

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



# VDO

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

아래 배포판은 "vdo.service"가 필요하다.

- RHEL 7
- RHEL 8

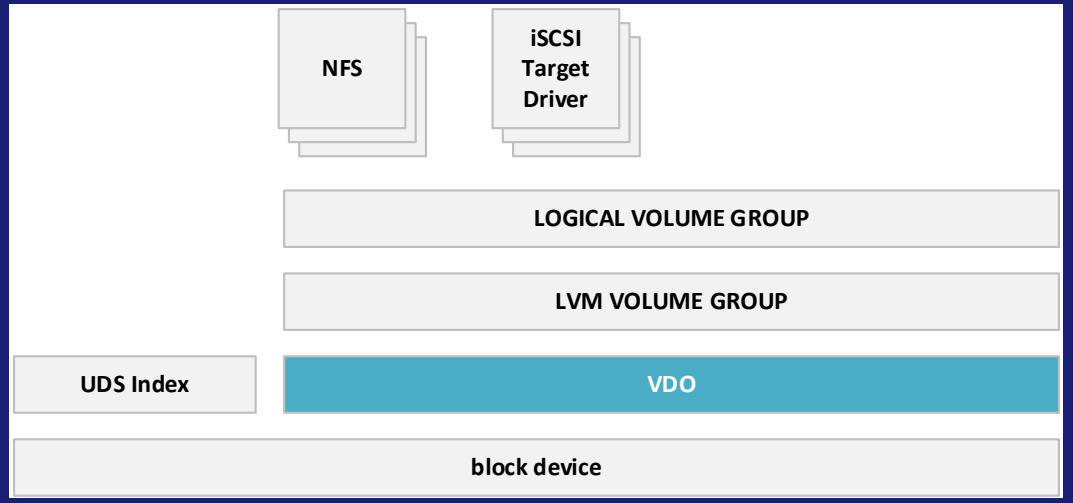
아래 배포판은 "vdo.service"가 필요하지 않는다.

- RHEL 9

# VDO

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

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



# VDO(LVM2)

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

# VDO

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

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

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

# VDO

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

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

이 이후 내용은 강사의 지시에 따라서 확인 및 검증한다.

# 연습문제

다음과 같이 발생한 스토리지 문제를 해결한다.

# DAY 1

블록장치 장애 확인 및 처리

# 블록장치 문제 확인

블록 장치 문제 해결을 하기 위해서 다음과 같은 부분을 확인해야 한다.

1. 커널에서 발생하는 드라이버 메시지
2. 파일 시스템에서 발생하는 장애 메시지
3. 배드 섹터(bad sector) 혹은 컨트롤러 장애 메시지

1,2번은 보통 커널 메시지에서 확인이 가능한 부분이며, 3번 경우에는 배드 섹터를 확인하기 위해서 몇가지 도구를 사용한다. 먼저, 실시간으로 배드 섹터를 확인하기 위해서 smartmontools를 사용한다.

```
# dnf install smartmontools -y
# smartctl -h
# smartctl -H /dev/sdb
```

# 배드 블록 생성

배드 블록을 명령어로 생성한다. 일단, 가상으로 사용할 파일 장치를 생성한다.

```
# dd if=/dev/urandom of=/tmp/file bs=512 count=32768 status=progress
# sha256sum /tmp/file
# loopdev=$(losetup -f --show /tmp/file)
# echo $loopdev
> /dev/loop0
# dmsetup create file1 << EOF
    0  2048 linear $loopdev 0
 2048  4096 error
 6144  26624 linear $loopdev 6144
EOF
# dmsetup create file2 << EOF
    0  30720 linear $loopdev 0
 30720  2048 error
EOF
```

# 배드 블록(smartctl)

smartctl를 통해서 특정 블록 디스크에 대해서 배드 블록 상태 확인이 가능하다.

```
# smartctl -H /dev/sdb
smartctl 7.2 2020-12-30 r5155 [x86_64-linux-5.14.0-427.13.1.el9_4.x86_64] (local
build)
Copyright (C) 2002-20, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF READ SMART DATA SECTION ===
SMART Health Status: OK
```

만약, smart기능을 제공하지 않는 블록장치(지금은 거의 없음) 경우에는 다음과 같이 확인이 가능하다.

```
# smartctl -x /dev/sdb -T permissive
Device does not support Self Test logging
Device does not support Background scan results loggingSMART Health Status: OK
```

# 배드 블록 생성

다음과 같이 명령어를 실행하면, 파일 기반 블록 장치에 배드 블록을 강제로 임의로 생성한다.

```
# ls /dev/mapper/
control  file1  file2  rl-home  rl-root  rl-swap
# dd if=/dev/mapper/file1 of=/dev/null count=2048
2048+0 records in
2048+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00632769 s, 166 MB/s
```

# 배드 블록 생성

블록에 크기를 조정하면서 가상의 배드블록 생성을 지속적으로 시도한다.

```
# dd if=/dev/mapper/file1 of=/dev/null count=2049
dd: error reading '/dev/mapper/file1': Input/output error
2048+0 records in
2048+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00660798 s, 159 MB/s
# dd if=/dev/mapper/file2 of=/dev/null count=30720
30720+0 records in
30720+0 records out
15728640 bytes (16 MB, 15 MiB) copied, 0.0802671 s, 196 MB/s
# dd if=/dev/mapper/file2 of=/dev/null count=30721
dd: error reading '/dev/mapper/file2': Input/output error
30720+0 records in
30720+0 records out
15728640 bytes (16 MB, 15 MiB) copied, 0.0890455 s, 177 MB/s
```

# 배드 블록 생성

파일 기반으로 생성된 블록장치에 강제로 배드 블록을 생성 및 복제한다.

```
# ddrescue -B -v -n /dev/mapper/file1 /tmp/file1 /tmp/log
# ddrescue -B -v -c 16 -r 2 /dev/mapper/file2 /tmp/file1 /tmp/log
# ddrescue -B -v -c 16 -r 2 /dev/mapper/file1 /tmp/file1 /tmp/log
```

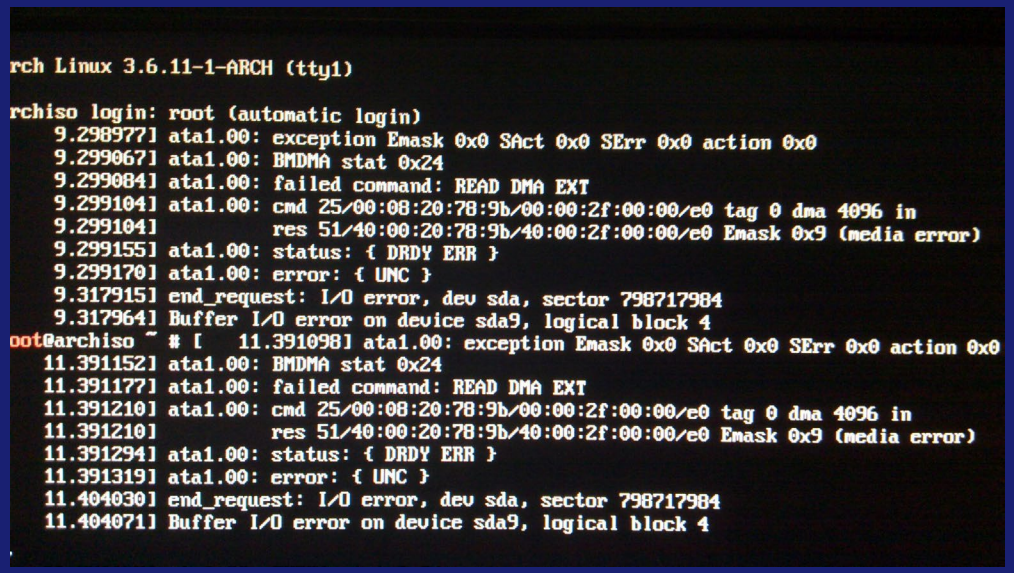
# 배드 블록 검사

배드 블록이 발생한 경우 해당 블록 영역을 사용할 수 없도록 표시를 해야 한다. badblock이라는 명령어로 블록에 마킹이 가능하다. 정확히는 파일 시스템 슈퍼블록에 해당 블록을 사용할 수 없도록 표시한다.

```
# badblocks -n /dev/mapper/file1
1024
1025
1026
1027
1028
# badblocks -v /dev/mapper/file2 3000 1024
Checking blocks 1024 to 3000
Checking for bad blocks (read-only test): done
Pass completed, 0 bad blocks found. (0/0/0 errors)
# badblocks -v /dev/sdb 40000 1024
Checking blocks 1024 to 40000
Checking for bad blocks (read-only test): done
Pass completed, 0 bad blocks found. (0/0/0 errors)
```

# 커널 메시지 확인

아래에서 사용할 journald에서 좀 더 학습하겠지만, 블록장치 및 모든 PCI장치들은 문제가 발생하면 커널에서 드라이바를 통해서 오류 메시지를 출력한다.



# 커널 메시지 확인

위의 메시지를 커널에서 실시간으로 확인하기 위해서 다음과 같이 명령어를 수행한다.

```
# journalctl list-boots
> 8cb2af897e9a482594cfb53b94603c65
# journalctl -b 8cb2af897e9a482594cfb53b94603c65 -p err -p warning
# journalctl -k -p err -p warning
```

```
hv_storvsc 443d9029-ebf8-4c7b-a883-925d398bb46e: tag#68 cmd 0x85 status: scsi
hv_storvsc 443d9029-ebf8-4c7b-a883-925d398bb46e: tag#69 cmd 0x85 status: scsi
hv_storvsc 443d9029-ebf8-4c7b-a883-925d398bb46e: tag#616 cmd 0x85 status: scs
hv_storvsc 443d9029-ebf8-4c7b-a883-925d398bb46e: tag#511 cmd 0x85 status: scs
hv_storvsc 443d9029-ebf8-4c7b-a883-925d398bb46e: tag#448 cmd 0x85 status: scs
block dm-0: the capability attribute has been deprecated.
TCP: eth0: Driver has suspect GRO implementation, TCP performance may be comp
XFS (sdb): Filesystem needs repair.  Please run xfs_repair.
XFS (sdb): Metadata CRC error detected at xfs_agi_read_verify+0xd9/0x110 [xfs
XFS (sdb): Unmount and run xfs_repair
XFS (sdb): First 128 bytes of corrupted metadata buffer:
00000000: 58 41 47 49 00 00 00 01 00 00 00 00 00 7f 00 00  XAGI............
00000010: 00 00 00 40 00 00 00 06 00 00 00 01 00 00 00 3d  ...@...........=
00000020: 00 00 00 80 ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000030: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000060: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000070: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ................
XFS (sdb): metadata I/O error in "xfs_read_agi+0x8f/0x140 [xfs]" at daddr 0x1
XFS (sdb): xfs_imap_lookup: xfs_ialloc_read_agi() returned error -117, agno 0
XFS (sdb): Failed to read root inode 0x80, error 117
```

```
buffer_io_error: 32 callbacks suppressed
Buffer I/O error on dev dm-3, logical block 256, async page read
Buffer I/O error on dev dm-3, logical block 256, async page read
Buffer I/O error on dev dm-3, logical block 256, async page read
Buffer I/O error on dev dm-3, logical block 256, async page read
Buffer I/O error on dev dm-3, logical block 257, async page read
Buffer I/O error on dev dm-3, logical block 257, async page read
Buffer I/O error on dev dm-3, logical block 257, async page read
Buffer I/O error on dev dm-3, logical block 257, async page read
Buffer I/O error on dev dm-3, logical block 258, async page read
Buffer I/O error on dev dm-3, logical block 258, async page read
buffer_io_error: 2038 callbacks suppressed
```

# 연습문제

다음과 같이 발생한 스토리지 문제를 해결한다.

# DAY 2

# systemd-journald

DAY 2

# journald

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

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

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

# journalctl 명령어

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

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

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

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

# 저널 중앙서버

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

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

# 저널 클라이언트

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

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

# 저널 로그확인

아래 명령어로 올바르게 동작하는지 확인한다. 이 명령어는 node1번에서 실행한다.

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

# 저널 서버 인증키

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

# 저널 서버 인증키 생성

```
# openssl req -newkey rsa:2048 -days 3650 -x509 -nodes -out ca.pem -keyout ca.key -subj '/CN=Certificate authority/'
ʁㅓČł 'nㅓὴἐGñ] ㅓㅓČł ɴČ빌 ╒Ţ
[ ca ]
default_ca = this
[ this ]
new_certs_dir = .
certificate = ca.pem
database = ./index
private_key = ca.key
serial = ./serial
default_days = 3650
default_md = default
policy = policy_anything
[ policy_anything ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
EOF
```

# 저널 서버 인증키 생성

아래 명령어를 순서대로 진행한다.

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

# journalctl boot

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

# journald persistent logging

```
# cp -a /run/log/journald /var/log/
# vi /etc/systemd/journald.conf
[Journal]
Storage=persistent
# systemctl restart systemd-journald
# systemctl is-active systemd-journald
# journalctl -b -1
# journalctl -b <BOOT_ID>
# killall -USR1 <SYSTEMD_JOURNALD>
# kill -USR1 <SYSTEMD_JOURNALD>
```

# .service logging

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

# podman journald

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

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

$ podman info | grep logDriver
logDriver: journald

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

# journal for kernel and boot logging

journald에서 커널에서 발생한 메시지 확인하기 위해서 아래와 같이 명령어를 사용한다.

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

# journald priority

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

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

# journalctl

별다른 수정없이 로그를 보고 싶은 경우, 다음과 같이 명령어 실행.

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

출력 방법은 변경하고 싶으면 다음과 같이 실행한다.

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

# journalctl

최근 메시지를 출력하고 싶으면 다음과 같이 한다.

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

로그 메시지가 얼마나 디스크 용량을 사용하는지 확인하려면 다음과 같이 한다.

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

로그 사이즈를 줄이기 위해서 다음과 같이 명령어를 실행한다.

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

# COREDUMP

DAY 2

# COREDUMP

코어 덤프를 생성하기 위해서 아래와 같이 간단하게 C코드 작성 및 컴파일 한다.

```
# vi core.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  printf("\n");
  printf("Process is aborting\n");
  abort();
  printf("Control not reaching here\n");
  return 0;
}
# gcc -o core core.c
```

# COREDUMP

바이너리를 실행하면 올바르게 코어 파일 생성이 되지 않는다. 코어파일을 생성하기 위해서 다음과 같이 설정한다.

```
# ulimit -a
# ulimit -c unlimited
```

위와 같이 설정이 완료가 되면, 코어 파일 생성이 가능하다. 코어 파일은 /var/lib/systemd/coredump에 생성이 된다.

```
# ./core
# ls /var/lib/systemd/coredump/
> core.core.0.b4b843f4b3584218b7e6aa311626f7cf.8089.1719753676000000.zst
# dnf install gdb -y
# gdb --args ./core
> r
```

# COREDUMP

영구적으로 coredump를 적용하기 위해서 아래와 같이 설정한다.

```
# vi /etc/systemd/system
DefaultLimitCORE=infinity
# systemctl daemon-reexec
```

# 연습문제

# SELINUX/ADUIT

DAY 2

# SELINUX

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

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

대다수 리눅스 시스템은 MAC 둘 중 하나를 사용하고 있다.

AppArmor

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

SELinux

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

# SELINUX

SELinux사용 상태 확인.

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

1: selinux 일시적으로 사용

0: selinux 일시적으로 중지

일시적으로 SELinux 사용 상태를 중지 혹은 사용으로 변경.

# SELINUX

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

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

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

# SEMANAGE

SELinux에서 사용하는 모든 컨텍스트에 대해서 관리 및 수정이 가능.

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

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

# BOOLEAN

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

`getsebool`

프로그램에서 사용하는 특정 기능(콜) 목록을 확인.

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

# BOOLEAN

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

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

변경된 내용에 대해서 확인하기 위해서는 다음과 같이 실행한다.

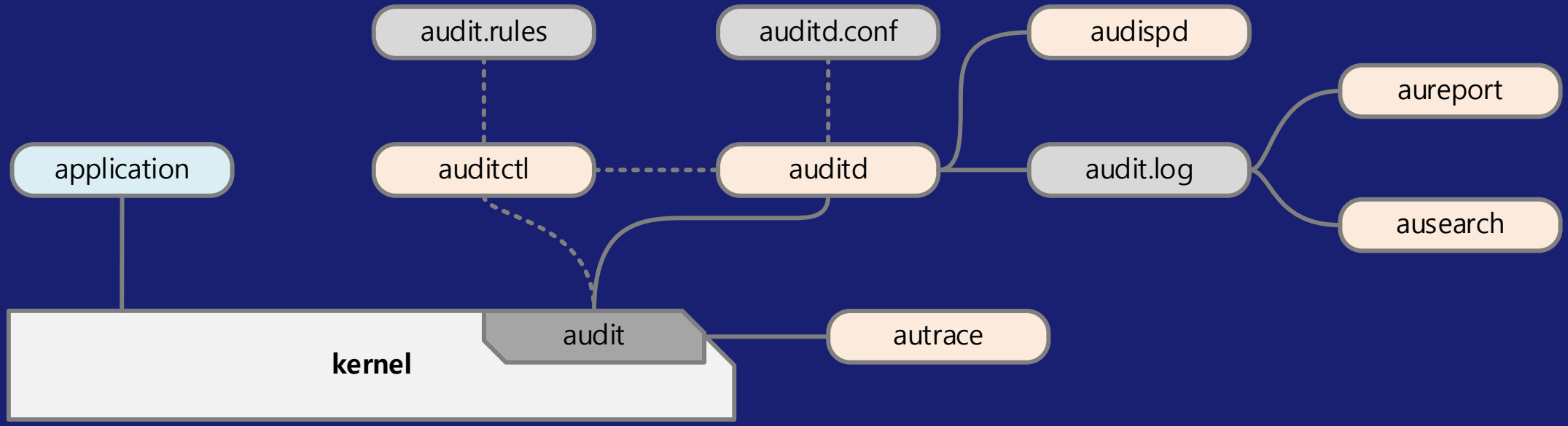
```
# semanage boolean -lC
```

`-l: list`

`-C: Customized`

# AUDIT

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



# AUDIT

특정 프로그램에서 실행하는 모든 콜에 대해서 등록 후 확인한다.

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

특정사용자가 파일 접근(open)에 대해서 등록 후 확인한다.

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

성공적으로 파일을 접근한 콜에 대해서 기록한다.

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

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

# AUDIT

특정 파일에 변경사항 확인하기

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

디렉터리 퍼미션 확인 및 감사

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

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

# AUDIT(search/report)

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

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

# DAY 2

IP utility

networkd

NetworkManager

# 네트워크 관리 명령어

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

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

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

# 기본 네트워크 관리 명령어

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

구성된 NIC의 아이피 정보 및 NIC상태 확인

```
# ip address show
```

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

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

# 기본 네트워크 관리 명령어

연결이 되어 있는 NIC카드 정보.

```
# ip link
```

현재 구성이 되어있는 라우팅 테이블 정보.

```
# ip route
```

# 네트워크 관리 명령어

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

# 네트워크 관리 명령어

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

# 네트워크 관리 명령어

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

# 더미 장치 생성하는 방법

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

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

# 네임 스페이스 장치

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

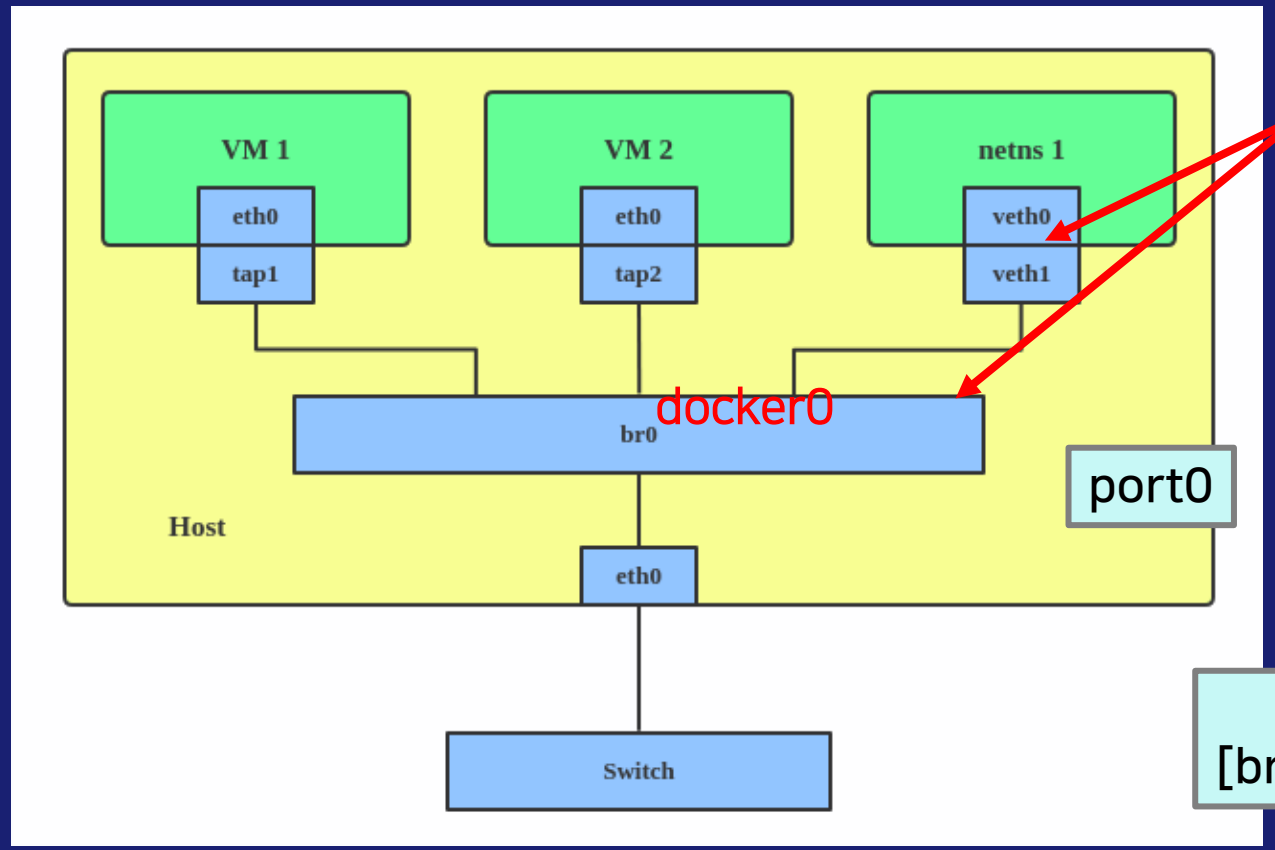
```
# ip netns add test
# ip netns list
test
# ls /var/run/netns/
test
# ip netns exec test ip link
```

# 네임 스페이스 장치

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

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

# 컨테이너 브리지 구현 예




application

"port 0"생성

veth1

port0

port1

port2

podman0
[bridge, switch]

# 컨테이너 브리지 구현 예

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

# 네트워크

네트워크 네이밍 및 마이그레이션

# 네트워크 네이밍

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

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

# 네트워크 네이밍

혹은 아래 방법으로 전환 가능.

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

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

dell bios naming

# 네트워크 네이밍

systemd-networkd를 통해서 장치 이름 변경이 가능하다.

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

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

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

# ifcfg-rh

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

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

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

# ifcfg-rh

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

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

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

# 네트워크

NetworkManager

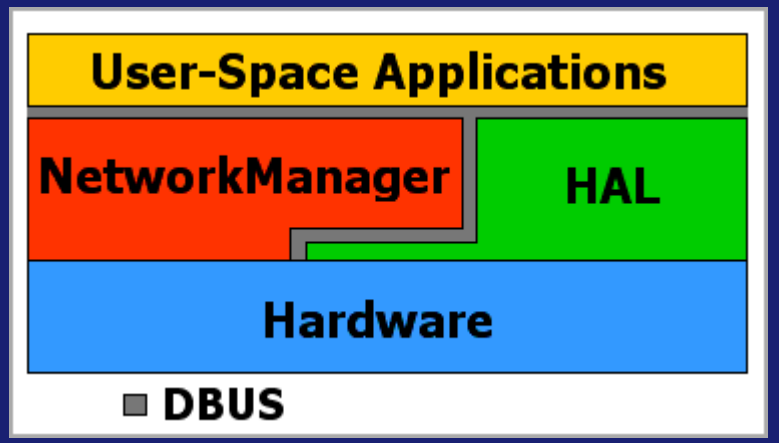
# 네트워크매니저

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

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

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

# NetworkManager



# 네트워크매니저

네트워크 매니저를 사용하기 위한 명령어는 아래와 같다.

```
# systemctl start NetworkManager
# nmcli
# nmtui
# nm-connection-editor
```

# 네트워크매니저

기존에 사용하던 network configuration은 init기반의 "shell scrip(/etc/init.d/)"로 되어 있었다.

ifcfg-rh는 "/etc/sysconfig/network-scripts"와 "/etc/sysconfig/network"를 통해서 관리 하였다. 현재는 RHEL 7이후로는 "Network Manager"기반으로 변경. RHEL 7에서는 호환성을 위해서 스크립트 플러그인을 지원하고 있다.

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

**네트워크 매니저 관리 명령어**

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

# 네트워크매니저

## NMTUI

TUI기반으로 네트워크 설정한다. 자동화 용도로 사용하기는 어렵다.

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

## NM-CONNECTION-EDITOR

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

# 네트워크매니저

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

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

## /etc/sysconfig/network-scripts/

RHEL 7/8까지는 지원. RHEL 9부터는 더 이상 지원하지 않는다. 하지만, 임시적으로 사용이 가능한 방법은 있다. 기존 rh-ifcfg와 호환을 위해서 다음과 같이 작업을 수행한다.

# 네트워크매니저

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

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

# 네트워크매니저

영구적으로 적용시키기 위해서 다음과 같이 작업을 수행한다.

```
# vi /etc/NetworkManager/NetworkManager.conf
[main]
plugins=ifcfg-rh,keyfile
rc-manager=auto
migrate-ifcfg-rh=true
# systemctl restart NetworkManager
# nmcli con sh
# nmcli connection migrate --plugin ifcfg-rh eth0 --file
# ls -l /etc/sysconfig/network-scripts/
```

자동화 기반으로 배포하기 위해서 가급적이면 NetworkManager보다는, systemd-networkd기반으로 구성 및 배포를 권장한다.

# 네트워크매니저 기존 방식

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

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

# systemd-networkd

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

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

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

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

# systemd-networkd

systemd기반으로 네트워크 구성하기 위해서 다음과 같이 작업을 수행한다.

```
# dnf install epel-release -y
# dnf install systemd-networkd
# systemctl enable --now systemd-networkd
# systemctl is-active systemd-networkd
> active
# networkctl list
IDX LINK TYPE     OPERATIONAL SETUP
  1 lo   loopback carrier     unmanaged
  2 eth0 ether    routable    unmanaged
```

# systemd-networkd

eth1에 고정 아이피로 적용 및 반영한다. 다만, 관리를 위해서 alternative name를 적용한다.

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

# systemd-networkd

네트워크 이름을 추가하기 위해서 링크 파일을 생성한다.

```
# /etc/systemd/network/70-eth-static.link
[Match]
MACAddress=00:15:5d:00:88:06
OrignalName=eth1
[Link]
AlternativeName=storage
```

위와 같이 작성 및 수정하면, 아래와 systemd-networkd에 반영한다.

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

# DAY 3

# DAY 3

kernel parameter

# 커널

리눅스 커널에서 사용하는 값들은 보통 두 가지 영역에서 조회가 가능하다.

1. `sysctl`
2. `/sys/`
3. `lsmod, modules-load.d/`

대다수 리눅스 배포판은 바닐라 커널 기반에서 각기 배포판에 맞게 커널 값을 빌트인 형태로 수정 후 컴파일 및 배포를 한다. 이러한 차이점을 확인하기 위해서 보통 커널 설정파일(.config)를 통해서 확인 한다.

# 커널 값 확인

커널 값 확인을 위해서 다음과 확인이 가능하다. 먼저 빌트인 모듈에 대해서 수정이 필요한 경우, 현재 사용중인 커널의 파라메터를 확인해야 한다. 예를 들어서 cdrom모듈에서 사용하는 버퍼 크기로 인하여 처리속도가 늦으면 다음과 같이 작업을 수행한다.

```
# grep -i -e buffer -e cdrom /boot/config-5.14.0-427.13.1.el9_4.x86_64
> CONFIG_CDROM_PKTCDVD_BUFFERS=8
```

해당 값을 변경하기 위해서 kernel config를 변경하지 않으며, 모듈 파라메터를 수정해야 한다.

```
# vi /etc/modprobe.d/dvd-buffer.conf
> CONFIG_CDROM_PKTCDVD_BUFFERS=12
# lsmod | grep cdrom
cdrom                  90112  1 sr_mod
# modprobe -r cdrom
# modprobe cdrom
```

# 테스트 프로그램

파이썬 기반 서버.

```
import time
import socket
import errno

daemon_port = 2425
payload = b'a' * 1448

listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_sock.bind(('0.0.0.0', daemon_port))

# listen backlog
listen_sock.listen(32)
listen_sock.setblocking(True)

while True:
    mysock, _ = listen_sock.accept()
    mysock.setblocking(True)

    # do forever (until client disconnects)
    while True:
        try:
            mysock.send(payload)
            time.sleep(0.001)
        except Exception as e:
            print(e)
            mysock.close()
            break
```

# 테스트 프로그램

파이썬 기반 클라이언트.

```
import socket
import time

def do_read(bytes_to_read):
    total_bytes_read = 0
    while True:
        bytes_read = client_sock.recv(bytes_to_read)
        total_bytes_read += len(bytes_read)
        if total_bytes_read ≥ bytes_to_read:
            break

server_ip = "192.168.2.139"
server_port = 2425

client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_sock.connect((server_ip, server_port))
client_sock.setblocking(True)

while True:
    do_read(1)
    time.sleep(0.001)
```

# 커널 파라메터

앞서 다룬 부분은 모듈에 대한 파라메터를 다룬 부분이며, **파일 시스템/네트워크/블록 장치**와 같은 영역은 커널 파라메터와 스케줄러를 조정해야 한다.

```
# sysctl -a | grep -i net\.ipv4\.tcp_[r,w]mem
> net.ipv4.tcp_rmem = 4096        131072  6291456
> net.ipv4.tcp_wmem = 4096        16384   4194304
```

일시적으로 장애를 처리하기 위해서 메모리 버퍼 크기 수정이 가능하다. 아래는 페이지 크기(4k)로 계산이 되어 있다. 일시적으로 크기를 8기가로 확장한다.

```
# sysctl -w net.ipv4.tcp_mem="8388608 8388608 8388608"
```

일시적으로 확장하여 문제 부분에 대해서 해결이 되었으면, 다음과 같이 영구적으로 적용이 가능하다.

# 커널 파라메터

영구적으로 적용하기 위해서 아래와 같이 작성한다.

```
# vi /etc/sysctl.d/tcp.conf
net.ipv4.tcp_mem="8388608 8388608 8388608"
```

문제 없이 적용이 되었으면, sysctl명령어로 다시 확인한다.

```
# sysctl -a | grep -i net\.ipv4\.tcp_[r,w]mem
```

# 디스크 스케줄러

디스크 스케줄러는 아래 위치에서 확인이 가능하다.

```
# cd /sys/block/sda/queue
# ls scheduler
scheduler
# cat scheduler
[none] mq-deadline kyber bfq
```

보통 레이드가 구성된 디스크는 별도로 스케줄러 사용이 필요 없으며, 특히나 NvMe/SSD와 같은 장치는 더더욱 스케줄러 활성화 되어 있으면, 블록장치 접근이 많이 늦을 수도 있다. 올바르게 영구적으로 변경하기 위해서 다음과 같이 수정한다.

# 디스크 스케줄러

udev에 다음과 같이 설정한다.

```
# vi/etc/udev/rules.d/60-schedulers.rules
ACTION=="add|change", KERNEL=="sdc", ATTR{queue/scheduler}="deadline"
```

위와 같이 추가 및 변경 후 다음과 같이 시스템에 반영한다.

```
# udevadm settle
# systemctl daemon-reload
```

# DAY 3

memory paging

swap(general swap, zram)

4/1/2024

# 다중 스왑 및 파일 스왑

리눅스에서 사용하는 스왑(SWAP)은 보통 2가지 형태로 사용한다.

- 파티션 형태의 raw block
- 파일 형태의 file block

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

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

# 파일기반 스왑 구성

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

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

# 파일기반 다중 스왑

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

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

# ZSWAP(일반)

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

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

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

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

# ZSWAP(특정위치)

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

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

# ZSWAP(확인)

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

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

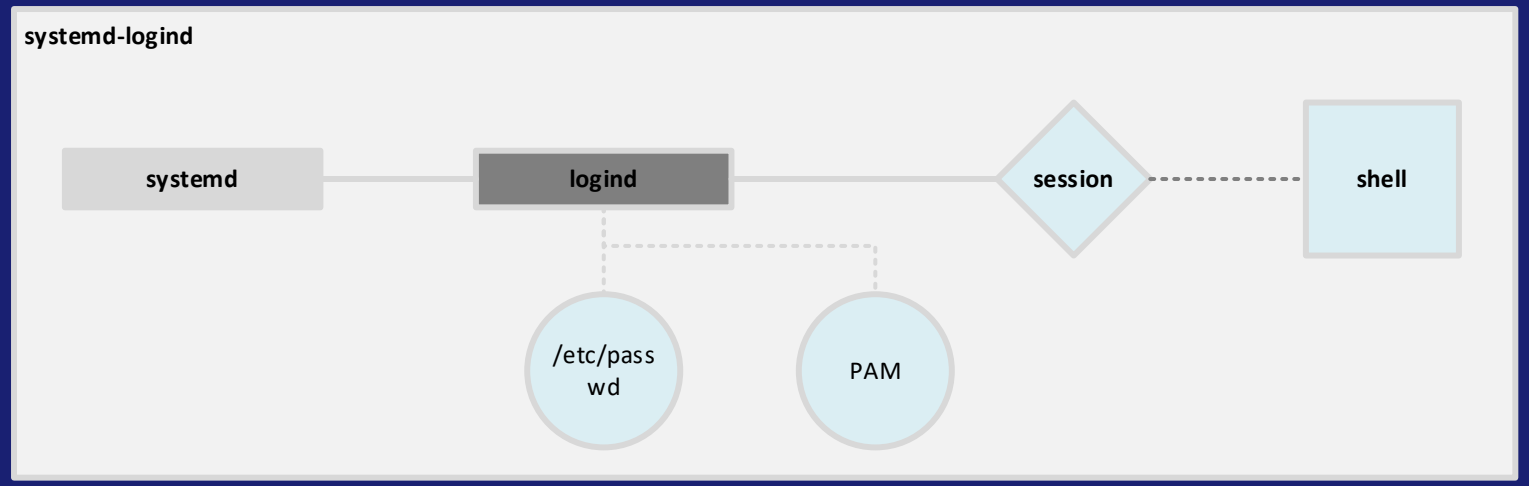
# DAY 3

user session recording

# 사용자

리눅스 사용자는 여전히 패스워드 파일 및 쉐도우 파일로  생성 및 관리된다. 다만, 사용자가 로그인 하였을 때, 이전에는 logind라는 프로세스가 담당하였지만, systemd로 전환이 되면서 systemd-logind라는 프로세스가 관리한다.

크게 다른 부분은 사용자 로그인 세션은 systemd에서 담당한다.



# 관리 명령어

┌╝ţ δ ╔Čn ┌

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

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

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

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

1 sessions listed.
```

# 관리 명령어

## ┌╝ţ δ ╔Čn ┌

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

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

# SESSION RECODING

세션 레코딩에는 두 가지 방법이 있다.

1. 사용자 쉘 기반
2. sssd기반의 session 레코딩

사용자 세션 레코딩을 원하는 경우, 아래와 같이 tlog를 설치한다.

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

# 세션 활동 기록 활성화(간단)

활동을 기록하기 위해서 아래와 같이 도구를 설치한다.

```
# adduser -s /usr/bin/tlog-rec-session recode_user
# ssh recode_user@localhost
> top
> ls
> exit
```

# 세션 활동 기록 활성화(간단)

재생을 위해서 아래와 같이 작업을 수행한다.

```
# journalctl -fl _COMM=tlog-rec-session
> "rec":"b4b843f4b3584218b7e6aa311626f7cf-18ba-fa324"
# tlog-play -r journal -M TLOG_REC=b4b843f4b3584218b7e6aa311626f7cf-18ba-fa324
```

# PAM

로그인 접근에 대해서 기본적으로 레드햇 계열은 7/8/9별도로 설정이 되어 있지 않다. 몇몇 서버들은 보안 이유로 로그인 차단을 해두는 경우가 있다. 이런 상황에서 어떻게 해야 하는지 혹은 로그인 차단 제한을 어떻게 구성하는지 확인한다.

```
# authselect enable-feature with-faillock
Make sure that SSSD service is configured and enabled. See SSSD
documentation for more information.
# ls -l cat /etc/security/faillock.conf
# cat /etc/pam.d/password-auth
# cat /etc/pam.d/system-auth
```

이 부분에 다음과 같이 수정한다.

# FAILCOUNT

아래와 같이 설정 후, 사용자 락 및 초기화를 테스트 한다. 대다수 레드햇 계열 8/9배포판은 이 기능으로 인하여 로그인이 안되는 경우들이 있다.

```
# authselect current
# less /etc/security/faillock.conf
# vi /etc/security/faillock.conf
unlock_time=1200
silent
# faillock --user lockuser
# faillock --user lockuser --reset
```

# DAY 3

system report and collection

system monitoring

4/1/2024

# 시스템 리포트

시스템에 장애 발생 시, 원격에서 진단이 필요한 경우가 있다. 이를 위해서 시스템 관련 정보 수집이 필요하다. 보통, 시스템 엔지니어는 다음과 같은 내용들을 가지고 분석 및 판단한다.

1. 시스템 로그
2. 장치 설정 및 디바이스
3. 소프트웨어 및 바이오스 혹은 펌웨어 정보
4. 메모리 덤프

위와 같은 내용 기반으로 정보를 수집한다. 모든 부분을 특정 도구를 통해서 수집이 불가능 이기 때문에 사용이 가능한 도구에 대해서 이야기 한다.

# sosreport

초기 리눅스는 시스템 정보를 수집하기 위해서 특정 스크립트 혹은 링크로 정보를 복제 및 복사를 하였다. 하지만, 리눅스가 점점 기업에서 많이 사용하면서 이러한 부분에 대해서 자동화가 필요하였고, 이를 위해서 sosreport라는 도구를 만들었다.

이 도구는 레드햇 리눅스(RHEL 아님)에서 사용하기 시작하였고, 현재는 모든 배포판에서 다 같이 사용하고 있다. 리눅스 배포판 별로 각기 다른 플러그인을 제공하고 있지만, 기본 플러그인은 거의 비슷하게 사용하기 때문에, 수집되는 내용은 거의 동일하다.

설치 및 사용 방법은 다음과 같다.

# sosreport

설치 및 구성 내용. 버전별로 다르다. 레드햇 계열 8버전부터 sos라는 이름으로 패키지가 변경 되었다. 사용방법도 배포판 버전마다 다르기 때문에, 각 버전에 알맞게 사용한다. 8버전 에서는 다중 서버에서 sosreport수집이 가능하다.

다만, 플레이북 기반이기 때문에, 반드시 모든 서버에 SSH키가 미리 배포가 되어 있어야 한다.

```
# dnf install sos -y
# sos report
# sos clean sosreport-rocky-2024-06-29-ruvseav.tar.xz
# sos report -l
# tar xJf sosreport-rocky-2024-06-29-ruvseav.tar.xz
```

# DEVICE

사용자 영역에서 다음과 같은 커널 혹은 장치 영역에 대해서 조회가 가능하다.

1. 바이오스
2. 슬롯 정보
3. 메모리
4. CPU
5. 디스크

일반적으로 하드웨어 장애가 발생하면, 위의 영역에서 많이 발생하기 때문에 해당 영역에 대해서 조회 및 확인 하는 방법에 대해서 확인한다.

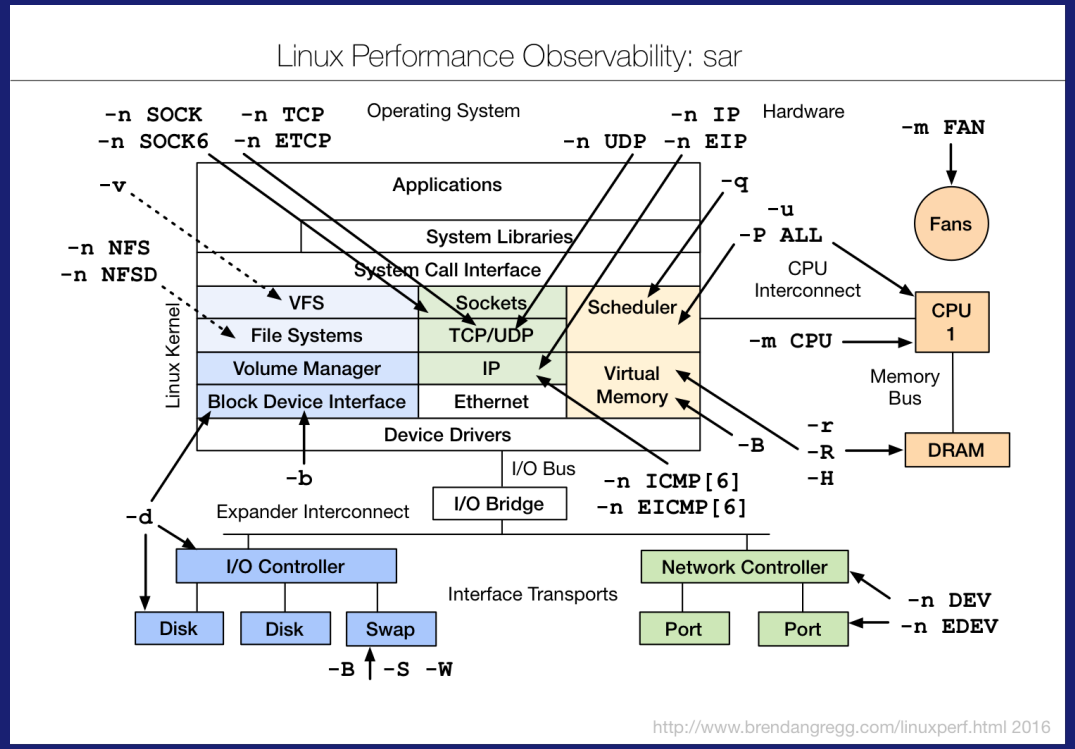
# DEVICE

일반적으로 sysstat(sar)를 통해서 수집된 정보 확인이 가능하다. sysstat는 실시간 서버 정보가 아니라 시스템에서 수집한 정보 기반으로 내용을 출력한다. 현재 SAR는 systemd로 변경이 되면서 아래와 같이 정보 수집 위치가 변경이 되었다.

```
# systemctl -t timer
sysstat-collect.timer
sysstat-summary.timer
```

# SAR 검증 범위

SAR도구를 통해서 다음과 같은 부분 확인이 가능하다. 다만, SAR는 성능 확인이기 때문에 장애 처리 용도로는 적절하지 않으며, 문제가 시스템에 어떠한 영향을 주는지 확인 용도로 사용한다.




**sar 1 3:** CPU 상태에 대해서 출력한다.
**sar -u 1 3:** 위의 옵션과 동일.
**sar -r 1 3:** 메모리 상태 확인.
**sar -S 1 3:** 스왑 상태 확인.
**sar -b 1 3:** 모든 블록에 대한 상태 확인.
**sar -d 1 3:** 독립적인 블록에 대한 상태 확인.
**sar -w 1 3:** 컨텍스트 스위칭 상태 확인.
**sar -q 1 3:** 실행 큐 상태 확인.
**sar -n 1 3:** 네트워크 리포트 확인.
**sar -s 1 3:** 이전 데이터 확인

# 분석을 위한 로깅

트러블 슈팅을 위해서 SAR기반으로는 충분하지 않다. 그 이유는 SAR는 사용자 영역에서 cron이나 timer를 통해서 특정 시간에 정보를 수집하기 때문에, 수집이 안되는 데이터 영역이 발생 할 수 있다. 또한, 이를 통해서 분석이나 혹은 그래프가 필요한 경우, 적절하지 않는다.

이러한 이유로 systemtap 그리고 PCP를 통해서 시스템의 자원 상태를 실시간으로 수집한다. 이 목차에서는 다루는 도구는 systemtap이 아닌, PCP를 통해서 수집 및 분석에 대해서 간단하게 확인한다.

# SYSTEMTAP

System TAP은 기본적으로 C언어 구조를 가지고 있으며, 사용자는 이 구조를 통해서 모니터링 영역 추가가 가능하다. 먼저, 사용하기 위해서 패키지 관리자를 통해서 설치한다.

```
# dnf install systemtap kernel-debug kernel-debug-devel -y
```

설치가 완료가 되면, 올바르게 동작하는지 아래와 같이 예제 파일을 만들어서 실행한다.

```
# vi hellword.stp
probe begin
{
  print ("hello world\n")
  exit ()
}
# stap helloworld.stp
```

# STAP

특정 inode에 대해서 모니터링이 필요한 경우 아래와 같이 확인이 가능하다.

```
probe kernel.function ("vfs_write"),
      kernel.function ("vfs_read")
{
  if (@defined($file->f_path->dentry)) {
    dev_nr = $file->f_path->dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_path->dentry->d_inode->i_ino
  } else {
    dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_dentry->d_inode->i_ino
  }

  if (dev_nr == ($1 << 20 | $2) # major/minor device
      && inode_nr == $3)
    printf ("%s(%d) %s 0x%x/%u\n",
      execname(), pid(), ppfunc(), dev_nr, inode_nr)
}
```

# STAP

위의 코드를 적용하기 위해서 다음과 같은 단계가 필요하다.

```
# stat -c "%D, %i" /etc/crontab
> fd00 201888768
# stap inode-watch.stp
```

# DAY 4

# DAY 4

QEMU/KVM Libvirtd

# 가상화

리눅스 가상화는 기본적으로 다음과 같은 도구로 구성이 된다.

- QEMU/KVM
- Libvirt

이를 통해서 가상 머신 라이프 사이클 관리 및 가상머신에서 사용하는 이미지를 디버깅 하기도 한다.

# libvirt

대다수 가상머신은 libvirt기반으로 라이프 사이클을 관리한다. 대표적으로 다음과 같은 대표적인 소프트웨어는 libvirtd기반으로 가상머신 라이프 사이클을 지원한다.

1. QEMU/KVM
2. OpenStack
3. Xen
4. oVirt

# virsh

가상머신 조회 혹은 디버깅을 위해서 virsh명령어를 사용해야 한다.

```
# virsh list
# virsh info <ID>
# virsh dom[RESOURCE] <ID>
# virsh domifaddr <ID>
# virsh domblklist <ID>
# virsh domfsinfo
```

# virsh

네트워크 장치 확인은 다음과 같이 진행한다.

```
# virsh net-list
# virsh net-info
# virsh net-dumpxml
# virsh net-define
# virsh net-port-list
```

# virt tools

추가적인 가상머신 관리도구 및 디버깅 도구는 아래 패키지를 설치한다.

```
# dnf install guestfs-tools
# dnf install virt-install
# dnf install libguestfs
```

가상머신 이미지 디버깅 시 보통 libguestfs를 통해서 실제로 가상머신을 실행하지 않고 이미지 디버깅이 가능하다. 예를 들어서 가상머신의 root 비밀번호 변경이 필요한 경우 아래와 같이 가능하다. 아래와 같이 임의로 올바르지 않는 비밀번호를 구성한다.

```
# virt-builder --list
# virt-builder --size 10G --root-password password:iswrong --format qcow2 --output /var/lib/libvirt/images/cos-9.qcow2 centosstream-9
```

# virt tools

구성이 완료가 되면 다음과 같이 비밀번호를 임시로 생성하고 변경한다. 아래는 버전에 따라서 명령어가 다르기 때문에 몇가지 명령어를 혼용해서 적었다.

```
# openssl passwd -1 test
$1$iDIqL/9K$r8VrzLyKoH7OaePjrZEUX.
# guestfish --rw -a /var/lib/libvirt/images/cos-9.qcow2
> add cos-9.qcow2
> pvs
> lvs
> list-filesystem
> mount /dev/rl/rl-root /
> vi /etc/shadow
> flush
> quit
```

# 가상머신 재배포

기존에 사용하던 가상머신 이미지를 다른 서비스 용도로 배포하기 위해서 다음과 같이 작업을 수행한다.

1. 디스크 이미지 초기화
2. 공통으로 사용하는 파일 제거
3. 임시 파일 및 민감한 인증 파일 초기화

```
# virt-sysprep -a /var/lib/libvirt/cos-9.qcow2
# virt-sparesify /var/lib/libvirt-/cos-9.qcow2
```

이를 통해서 이미지 초기화 후, 다시 재 배포 및 재사용이 가능하다. 만약, 가상머신 이미지 초기화가 필요한 경우 아래 명령어로 빠르게 초기화(포맷)가 가능하다.

```
# virt-format -a /var/lib/libvirt/cos-9.qcow2
```

# DAY 4

Container

# 컨테이너

리눅스 컨테이너는 Podman 및 runc, crun 그리고 OCI이미지 기반으로 운영 및 구성이 된다. 고수준 컨테이너 런타임, 저수준 컨테이너는 OCI 및 CRI인터페이스를 따르는 경우 사용하는 디렉터리 위치는 동일한다.

일반적으로 표준 컨테이너가 사용하는 디렉터리 위치는 다음과 같다.

- /var/lib/containers/storage
- /run/containers
- /run/pods

또한, 네트워크는 CNI라는 플러그인을 통해서 구성하기 때문에, 네트워크 관련 설정은 아래 위치에서 구성이 된다.

- /etc/cni.d/

# 컨테이너 생성

컨테이너가 올바르게 생성이 되는지, 혹은 문제가 있는지 확인하기 위해서 다음과 같은 방법으로 접근 및 확인한다.

```
# dnf install podman -y
# podman run -d --name test --rm nginx
# podman inspect nginx
```

# 컨테이너 디버깅

위와 같이 고수준 컨테이너 런타임 및 엔진 설치 후 생성하면 문제 없이 동작한다. 생성 및 동작한 컨테이너 관련된 로그를 확인하기 위해서 다음과 같이 접근한다. 저널에서 컨테이너 프로세스 확인은 레드햇 계열 기준 8버전부터 지원한다.

```
# podman logs nginx
# podman inspect test --format '{{.State.Pid}}'
# podman info --format '{{ .Host.LogDriver }}'
# journalctl -fl _PID=$(podman inspect test --format '{{.State.Pid}}'
# journalctl CONTAINER_NAME=test
```

정확히, Podman에서 systemd-journald를 지원하지 않으면 CONTAINER_*시작되는 지시자 사용은 안된다.

# 컨테이너 SELINUX

컨테이너에서 사용하는 블록 장치는 보통 호스트에서 사용하는 파일 시스템 영역을 backingFsBlockDev 형태로 구성한다. 다른 말로 블록 장치는 존재하지 않지만 overlay2모듈을 사용해서 마치 블록장치가 존재하는 것 마냥 구성한다.

이러한 이유로 SELinux가 활성화가 되어 있는 시스템이며, 반드시 SELinux Context를 구성해야 한다.

```
# mkdir /root/disk
# echo "hello world" > /root/disk/index.html
# podman run -d --name httpd-selinux --rm --volume /root/disk/:/var/www/html/ -p
8080:8080 docker.io/library/httpd:latest
# curl $(hostname -I)/index.html
# podman stop httpd-selinux
# podman run -d --name httpd-selinux --rm --volume
/root/disk/:/usr/local/apache2/htdocs:Z -p 8080:80 docker.io/library/httpd:latest
# curl localhost/index.html
```

# 컨테이너 로깅

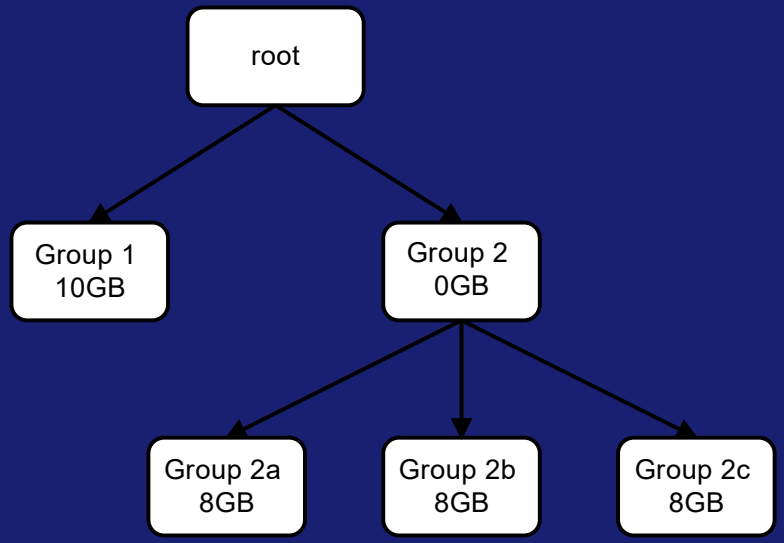
# DAY 4

MEMORY OOM

4/1/2024

# OOM

OOM(Out Of Memory)의 약자이며, 주로 하는 역할은 OS의 구동에 필요한 메모리가 부족할 때 메모리에서 동작하는 프로세스를 특정 조건에 따라서 중지를 한다. 모든 프로세스는 각각 메모리 그룹이 있으며, 각 부모들은 자식의 프로세스를 관리한다. OOM이 동작하면, 자신의 그룹에 포함된 자식 프로세스를 우선 순위 기준으로 메모리에서 중지 및 제거를 수행한다.



# OOM

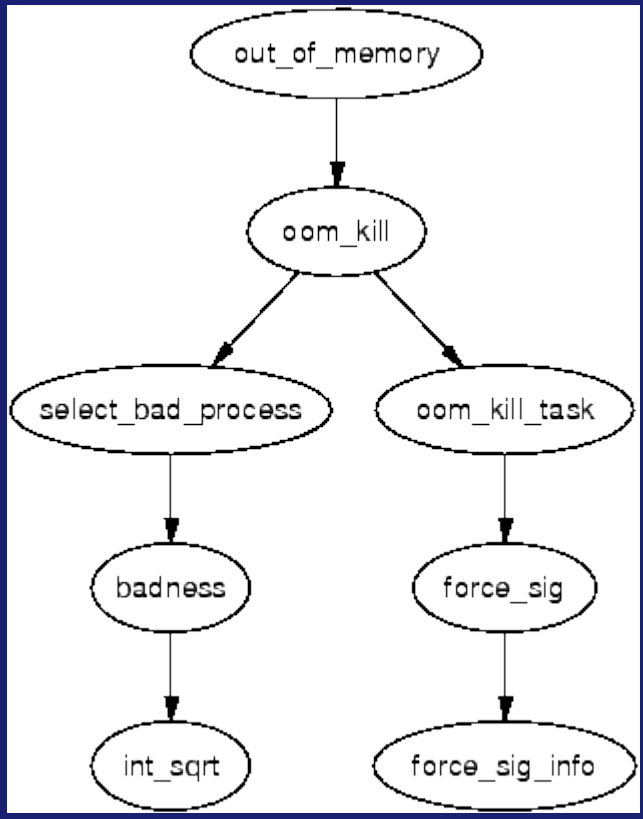
OOM동작 우선 순위 설정이 가능하다. 기본적으로 모든 프로세스는 OOM SCORE를 가지고 있으며, 이를 통해서 어떠한 프로세스를 먼저 중지할지 결정한다.

```
# cat oom_score
667
```

OOM은 프로세스를 중지하기전에 다음과 같은 조건을 확인 후 커널에서 OOM작업을 수행한다. 리눅스 커널은 기본적으로 루트 프로세스(root process)는 최소 3% 메모리를 보장이 되어야 한다.

**TotalRam * (OverCommitRatio/100) + TotalSwapPage,OverCommitRatio**

그래서, 이러한 조건으로 페이징을 진행하다가 더 이상 메모리가 할당하기 어려운 상황이면 OOM이 동작하면서 프로세스 중지가 시작이 된다.



# OOM프로세스

OOM 조건 확인은 아래와 같다.

1. Is there enough swap space left (nr_swap_pages > 0) ? If yes, not OOM
2. Has it been more than 5 seconds since the last failure? If yes, not OOM
3. Have we failed within the last second? If no, not OOM
4. If there hasn't been 10 failures at least in the last 5 seconds, we're not OOM
5. Has a process been killed within the last 5 seconds? If yes, not OOM

위의 조건을 통과한 프로세스는 oom_kill()를 통해서 프로세스 종료가 된다. OOM조건에 발동이 되려면, 먼저 CPU 사용율이 높은 프로세스를 아래처럼 계산하여 찾는다.

```
badness_for_task = total_vm_for_task / (sqrt(cpu_time_in_seconds) *
sqrt(sqrt(cpu_time_in_minutes)))
```

# OOM CONFIG

OOM를 동작하면, OS 및 루트 프로세스의 최소 메모리 크기를 보장하지만, 시스템에서 동작하는 프로그램, 예를 들어서 데이터베이스나 미들웨어 같은 프로그램이 강제로 종료가 되거나 혹은 인스턴스가 종료가 될 수 있다.

이러한 이유로 중요한 서버, 예를 들어서 데이터베이스 및 미들웨어에서는 이 기능을 꺼두는 경우도 있다.

```
# vi /etc/sysctl.d/disable-oom.conf
vm.overcommit_memory=2 혹은 1
```

일시적으로 끄기 위해서는 다음과 같이 명령어를 실행한다.

```
echo 2 > /proc/sys/vm/overcommit_memory
```

# OOM CONFIG

혹은 다음처럼 oom_score를 변경이 가능하다. 특정 프로세스에서 oom를 사용하기 싫으면 -17, +15 최우선 적용이다. oom_adj에 위의 값을 적용하면 된다.

```
echo -1000 > /proc/<PID>/oom_score_adj
```

systemd기반에서는 다음과 같이 서비스 파일에 override가 가능하다.

```
# mkdir -p /etc/systemd/system/httpd.service.d/
# vi /etc/systemd/system/httpd.conf.d/httpd.conf
[Service]
OOMScoreAdjust=-1000
# systemctl daemon-reload
# systemctl restart httpd
```

버전에 따라서 다르지만, 명령어로 다음과 같이 수정이 가능하다.

```
# systemctl edit --drop-in=httpd
```

# OOM SCREEN

OOM동작 메시지 및 화면을 확인하기 위해서 아래와 같이 C언어로 코드를 작성한다.

```
#include <stdio.h>
#include <stdlib.h>

#define CR 13

int main(){
   char *fptr;
   long i, k;

   i = 50000000000L;

   do{
      if(( fptr = (char *)malloc(i)) == NULL){
         i = i - 1000;
      }
   }
   while (( fptr == NULL) && (i > 0));

   sleep(15);
   for(k = 0; k < i; k++){
      fptr[k] = (char) (k & 255);
   }
   sleep(60);
   free(fptr);
   return(0);
}
```

# DAY 4

RPM/DNF3

# RPM/DNF

RPM은 레드햇 계열 배포판에서 사용하는 로컬 패키징 파일이다. 이를 rpm명령어로 관리하며, RPM에서 발생하는 장애도 역시 rpm명령어로 수정 및 해결한다.

RPM에서 발생하는 문제는 보통 아래와 같다.

- RPM 데이터베이스 손상
- GPG키 손상
- 패키지 파일 및 디렉터리 손상

장애나 혹은 파일 손상이 의심이 되는 경우, 어떠한 방식으로 복구가 가능한지 확인한다. RPM패키지 관리자는 원격 패키지에 대한 관리 기능을 제공하지 않기 때문에, DNF혹은 YUM를 통해서 원격에서 RPM다운로드 및 설치가 가능하다.

# RPM

RPM디비가 손상이 되었을 때 어떠한 방식으로 복구 하는지 확인한다. 가급적이면, RPM디비도 그때그때 백업하는 게 제일 안전하다.

```
# mkdir /backup
# tar -zcf /backups/rpmdb-$(date +"%d%m%Y").tar.gz  /var/lib/rpm
```

락킹 파일이 있는 경우, 락킹 파일을 제거한다.

```
# rm -f /var/lib/rpm/__db*
```

데이터베이스를 수동으로 백업 및 복구를 하기 위해서는 아래와 같이 작업이 가능하다.

```
# dnf install libdb-utils
# db_dump Packages > Packages-backup
# db_dump Packages-backup | db_load Packages
# rpm --rebuilddb
```

# RPM

레드햇 계열 9버전 이후로는 RPM데이터베이스가 Berkely DB에서 Sqlite으로 변경이 되었다. 기본적인 데이터베이스 파일 및 명령어는 비슷하지만, 파일 구성이 많이 달라졌다.

```
# pwd
/var/lib/rpm
# ls
rpmdb.sqlite   rpmdb.sqlite-shm   rpmdb.sqlite-wal
```

이전에 파일로 구성이 된 부분들은 전부 sqlite테이블로 통합이 되었다.

```
# sqlite3 rpmdb.sqlite
sqlite> .tables
sqlite> .tables
Basenames              Name                   Sigmd5
Conflictname           Obsoletename           Suggestname
Dirnames               Packages               Supplementname
```

# DNF

DNF는 이전 YUM를 대체해서 사용하는 네트워크 패키지 관리자 이다. 간단하게 설명하면, DNF는 기존 YUM에서 단점인 늦은 반응 및 많은 메모리 소비 문제를 해결하기 위해서 핵심 부분을 C/C++언어로 다시 재구성하여 만든 도구이다.

참고로, YUM은 거의 대다수 기능이 Python기반으로 동작한다. 또한, DNF는 추상적인 패키지 라이브러리 PackageKit를 지원 및 사용하기 때문에, 다른 배포판의 패키지도 DNF에서 사용이 가능하다.

DNF으로 통합이  되면서, 이전에 사용하였던 yum-utils와 같은 외부 패키지 명령어를 내부로 통합 하였다.

https://github.com/rpm-software-management/dnf

# 패키지 다운로드

폐쇄망 시스템에 RPM를 전달 혹은 RPM 저장소를 구성하기 위해서 다음과 같이 구성이 가능하다.

```
# mkdir /srv/rpms
# dnf reposync -p /srv/rpms
```

혹은 특정 패키지 의존성만 내려받기를 원하는 경우, 아래와 같이 명령어를 실행한다.

```
# dnf download httpd --resolve
# dnf groupinstall --downloadonly --downloaddir "Server with GUI"
```

# DAY 4

Linux Attribute Permission

# 리눅스 속성 문제

리눅스에는 다음과 같은 퍼미션 시스템이 있다.

- DAC
- MAC
- POSIX PERMISSION

DAC는 chmod, chown를 통해서 구성하며, MAC는 SELinux 그리고, POSIX는 ACL를 통해서 구현 및 구성한다. 하지만, 좀 더 섬세한 접근 제한 및 속성을 관리하기 위해서 Filesystem Attribute기능을 제공한다.

다만, 위의 기능은 모든 파일 시스템에서 제공하지 않으며, 지원하는 파일 시스템이 있다. Attribute는 Common 그리고 Extended형태로 두 개를 지원한다. 일반적으로 사용자가 수정이 가능한 부분은 Extended형태이다.

1. Ext4
2. BTRFS
3. XFS

# 파일 속성 설정

특정 파일에 대해서 속성 및 네임스페이스를 설정한다. 네임스페이스 flavor에 값 vanilla를 설정한다.

```
# attr -s flavor -V vanilla example.com
Attribute "flavor" set to a 7 byte value for example.com:
vanilla
```

위의 내용은 아래와 같이 짧게 표현이 가능하다.

```
# setfattr --name user.flavor --value chocolate example.com
```

추가된 내용을 확인하기 위해서 아래와 같이 명령어로 조회가 가능하다.

```
# getfattr --name user.flavor example.com
# file: example.com
user.flavor="chocolate"
# attr -l example.com
Attribute "selinux" has a 38 byte value for example.com
Attribute "flavor" has a 9 byte value for example.com
```

# 파일 속성 설정

위의 내용 계속…

```
# getfattr example.com
# file: example.com
user.flavor
```

내용 변경은 아래와 같이 가능하다.

```
# setfattr --name user.flavor --value strawberry example.com
# getfattr -d example.com
# file: example.com
user.flavor="strawberry"
```

# 파일 제거 방지 방법

파일 제거를 방지하기 위해서 몇 가지 방법이 있다.

```
# chattr +i example.com
# lsattr -l example.com
# rm -f example.com
# echo <<EOF> example.com
> hello
> This is my World
EOF
# chattr -i -a example.com
# chattr +a example.com
# echo <<EOF>> example.com
hshshshshs
skjdlksj
EOF
# rm -f example.com
```