[KISTI 기술문서] ISBN 978-89-294-1798-7

리눅스 컨테이너 기술 분석

2024. 12. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- 리눅스 시스템이 점점 더 복잡해지고 다양한 환경에서 사용됨에 따라, 기존의 시스템 관리 방식에 한계에 부딪히게 됨으로, 보다 효율적이고 유연한 시스템 관리 도구의 필요성이 대두됨. 가상화 기술과 컨테이너 기술[1]의 발전으로 하나의 물리적 시스템을 여러 개의 격리된 환경을 운영해야 할 필요성이 생기게 됨. 이는 리소스(자원) 관리와 프로세스 격리에 대한 새로운 접근 방식을 요구하게 되었다. 클라우드 컴퓨팅과 마이크로서비스 아키텍쳐와 같은 현대적인 IT환경을 지원하기 위한 목적으로 리눅스 시스템이 발전되었다.

- 이 문서에서는 다음과 같은 요소기술에 대해서 분석한다:
* systemd 는 기존의 init 시스템 보다 빠른 부팅 속도를 제공하며, 서비스들간의 복잡한 의존성을 효과적으로 관리함.
* cgroup은 프로세스 그룹에 대한 리소스 사용량을 제한할 수 있다. 컨테이너 기술의 핵심 구성요소로 컨테이너 간의 리소스 격리를 가능하게 함.
* namespace는 동일한 시스템에서 다른 프로세스 그룹을 분리함으로써, 보안을 강화한다. 컨테이너 기술의 또다른 핵심 구성요소임.
* linux capability는 전통적인 슈퍼유저 모델을 세분화하여 특정 권한을 개별적으로 부여할 수 있게 해주는 메커니즘임

- 컨테이너의 역사를 거슬러 올라가면 chroot 가 등장함. chroot는 프로세스의 루트 디렉터리를 변경하는 시스템 호출 명령으로 1979년 Unix 버전 7에 적용되어 원격유저(FTP 등)를 특정 디렉터리에 가두기 위한 용도로 사용됨. 2003년 Google의 Borg[2] 컨테이너 클러스터 관리 시스템 도입, 2004년 프로세스 컨테이너가 발전된 cgroup가 등장하였고, 2008년 cgroup은 리눅스 커널에 통합되었다. 리눅스 네임스페이스(namespace)가 도입되었고, cgroup와 namespace를 활용한 LXC(linux container)가 등장하였다. 이후 보다 현대적인 컨테이터 기술인 Docker(2013년)와 Kubernetes(2015년)가 발표됨.

- 하이퍼바이저(hypervisor)와 컨테이너 기술은 모두 가상화 기술이지만, 다음과 같은 차이가 존재함: 하이퍼바이저는 하드웨어 수준의 가상화를 지원하며, VM(virtual machine)마다 독립된 OS가 필요하지만, 컨테이너는 운영체제 수준의 가상화 기술이며, 호스트 OS의 커널을 공유한다는 차이가 있다.

- 본 문서에서 사용된 예제코드는 다음 깃허브 주소에서 찾을 수 있다:

```
git clone https://github.com/swkim85/linux-drill
cd linux-drill/container
```

1) https://en.wikipedia.org/wiki/Containerization_(computing)
2) https://en.wikipedia.org/wiki/Borg_(cluster_manager)

2. systemd

- systemd[3] (system and service manager)는 리눅스 시스템에서 사용되는 초기화 시스템 및 서비스 관리자임. 기존 SystemV UNIX의 init 시스템을 대체하기 위해서 개발됨. systemd은 시스템 구성 요소를 유닛(unit)이라는 단위로 추상화하여 관리하여, 서비스 간의 의존성을 고려하여 시작 순서를 제어함. 서비스를 병렬로 시작할 수 있어서 부팅 속도가 향상되는 장점이 있다.
- systemd는 여러 컴포넌트로 구성됨: ① systemd 데몬(`/sbin/init --> /lib/systemd/systemd`) ② systemd-journald : 로그관리 ③ systemd-logind: 사용자 로그인 및 세션 관리 ④ systemd-udevd : 장치관리 ⑤ systemd-networkd: 네트워크 관리 ⑥ systemd-resolvd: DNS 해석
- 서비스 뿐만 아니라, 시스템에서 다양한 자원을 제어한다. 이 자원들을 유닛(Unit)이라고 부름. 유닛의 종류: 서비스(Service), 소켓(Socket), 마운트(Mount), 타켓(Target), 스왑(Swap) 이 존재. 유닛을 나타내는 파일의 파일명은 `name.unit_type` 규칙으로 만든다.
  예시) `cron.service` : 이름이 cron 이고, 유닛의 종류는 service임
- 유닛 파일은 다음 경로에 저장됨
  * `/lib/systemd/system` 디렉터리: 패키지 관리자에 의해 설치된 기본 유닛 파일들이 저장됨. 시스템에 설치된 소프트웨어나 서비스가 자동으로 생성한 유닛 파일들이 이 위치에 저장됨
  * `/etc/systemd/system` 디렉터리: 시스템 관리자가 직접 생성하거나 수정한 유닛 파일들을 위한 디렉터리: 이 파일들은 /lib/systemd/system의 파일들보다 우선순위가 높다
  * `/run/systemd/system/` : 런타임에 생성된 systemd 유닛 파일들이 저장되는 디렉터리
  ※ /lib/systemd 는 /systemd 와 동일한 경로임 hard link로 연결되어 있다

- 유닛파일은 Section으로 나뉘어져 있음. 각 섹션은 key-value 형식으로 정의함. 유닛 파일을 수정한 이후에는 `systemctl daemon-reload` 를 수행해야 함.

```
   유닛 파일 형식:
[SectionName]
Directive1=value
Directive2=value
```

2.1 Systemd Unit

- nginx 웹서버의 systemd 서비스 유닛파일

```
  nginx 서비스 설치하기
$ sudo apt install nginx
$ dpkg-query -S /usr/lib/systemd/system/nginx.service
nginx-common: /usr/lib/systemd/system/nginx.service
$ more /usr/lib/systemd/system/nginx.service
[...]
[Unit]
Description=A high performance web server and a reverse proxy server
Documentation=man:nginx(8)
   After= 지시어는 해당 유닛이 지정된 다른 유닛들이 시작된 후에 시작되도록 순서를 지정한다.
  After=에 나열된 유닛들이 완전히 활성화 상태가 될 때까지 기다림. 의존성이 아닌 단순히 시작 순서만 지정할 뿐이므로, After=에 지정된 유닛이 실패해도 해당 유닛은 여전히 시작될수 있음.
After=network-online.target remote-fs.target nss-lookup.target
```

3) https://systemd.io/

```
  Wants= 지시어는 약한 의존성 관계를 설정함. Requires= 보다 덜 엄격한 의존성을 나타냄. 해당 유닛이 활성화 될 때 Wants=에 나열된 유닛들도 함께 시작하려고 시도함.
Wants=network-online.target

[Service]
  Type= 지시어는 서비스 유닛의 시작 유형을 정의함. 다음중 하나가 가능.
  simple(기본값): 메인프로세스가 즉시 시작됨. ExecStart=로 지정된 프로세스가 메인프로세스가 됨
  forking: 부모 프로세스가 fork()를 호출하고 종료되고, 자식 프로세스가 실제 메인 프로세스가 됨
  oneshot: 프로세스가 완료될 때까지 기다린 후 다음 유닛을 시작함. 짧은 작업을 수행하는 스크립트에 적합.
  notify: forking과 유사하지만 데몬이 준비되었음을 systemd에 알림. sd_notify() 함수를 사용하여 준비 상태를 알려야 함.
  dbus: 서비스가 D-Bus에 이름을 등록할 때까지 기다린다. BusName=옵션과 함께 사용
  idle: 다른 작업들이 완료된 후에 서비스를 시작한다
Type=forking
PIDFile=/run/nginx.pid        서비스 시작후 이 파일에서 메인 프로세스의 PID를 읽음
  ExecStartPre= 지시어는 메인프로세스가 시작되기 전에 실행될 명령을 지정함. 여러개의 ExecStartPre= 지시어가 실패하면 서비스 시작이 중단된다. 명령 앞에 '-'를 붙이면 실패를 무시하고 계속 진행. 주로 디렉터리 생성, 권한 설정, 환경 준비 등의 작업에 사용됨
ExecStartPre=/usr/sbin/nginx -t -q -g 'daemon on; master_process on;'
  ExecStart= 지시어는 서비스를 시작할 때 실행할 명령을 지정함
ExecStart=/usr/sbin/nginx -g 'daemon on; master_process on;'
  ExecReload= 지시어는 systemctl reload <unit-name>명령을 실행할 때 이 지시어에 정의된 명령이 실행됨.
ExecReload=/usr/sbin/nginx -g 'daemon on; master_process on;' -s reload
 ExecStop= 는 systemctl stop <unit-name> 명령을 실행할 때 이 지시어에 정의된 명령이 실행됨. 이것은 TimeoutStopSec 지시어로 지정된 시간 내에 완료되어야 함.
ExecStop=-/sbin/start-stop-daemon --quiet --stop --retry QUIT/5 --pidfile /run/nginx.pid
TimeoutStopSec=5
  KillMode= 지시어는 서비스 유닛 파일의 서비스 종료 방식을 지정함. 네 가지 옵션중 하나를 선택가능. control-group(기본값) 제어 그룹에 있는 모든 남은 프로세스를 종료함. mixed: 메인 프로세스에는 SIGTERM을 제어 그룹의 나머지 프로세스에는 SIGKILL을 보냄. process: 메인 프로세스만 종료함. non: 프로세스를 종료하지 않음
KillMode=mixed

[Install]
   [Install] 섹션의 WantedBy= 지시어는 유닛이 어떻게 활성화 되어야 하는지를 지정하는 가장 일반적인 방법. 유닛이 활성화 되면 지정된 타겟의 .wants/ 디렉터리에 심볼릭 링크를 생성함  (/etc/systemd/system/multi-user.target.wants/nginx.service -> /usr/lib/systemd/system/nginx.service )
WantedBy=multi-user.target
```

※ multi-user.target 은 systemd에서 관리하는 특별한 유닛임[4]. 특수 시스템 유닛은 다음과 같은 것들이 있다:

분류	유닛명	설명
부팅관련	basic.target	기본적인 부팅 과정을 다루는 타겟 유닛. 대부분의 서비스가 이 유닛 이후에 시작됨
	default.target	시스템 부팅 시 기본적으로 시작되는 타겟. 보통 multi-user.target 이나 graphical.target 에 연결됨
	initrd-fs.target	초기 RAM 디스크 부팅 과정을 관리
시스템 상태	multi-user.target	다중 사용자 모드를 나타내는 타겟

4) https://www.commandlinux.com/man-page/man7/systemd.special.7.html

관련	graphical.target	그래픽 사용자 인터페이스를 포함한 완전한 시스템 부팅을 나타냄
	rescue.target	단일 사용자 모드로 시스템을 부팅
	emergency.target	최소한의 환경으로 시스템을 부팅함
전원 관리	suspend.target	시스템 일시 중지를 관리
	hibernate.target	시스템 최대 절전 모드를 관리
	poweroff.target	시스템 종료를 관리
	reboot.target	시스템 재부팅을 관리
시스템 리소스	-.slice	모든 슬라이스의 루트
	system.slice	시스템 서비스를 위한 슬라이스
	user.slice	사용자 세션을 위한 슬라이스
기타 중요한 유닛	dbus.service	D-Bus 시스템 메시지 버스를 관리
	syslog.socket	시스템 로깅 서비스를 위한 소켓
	-.mount	루트 파일시스템의 마운트 포인트를 나타냄

2.2 systemd 주요 명령어

- Systemd 주요 명령어

```
$ systemctl --version
     # systemd 버전 확인 +는 활성화된 기능. -는 포함되어 있지 않거나 비활성화된 기능을 의미함.
systemd 255 (255.4-1ubuntu8.4)
+PAM +AUDIT +SELINUX +APPARMOR +IMA +SMACK +SECCOMP +GCRYPT -GNUTLS +OPENSSL +ACL +BLKID +CURL
+ELFUTILS +FIDO2 +IDN2 -IDN +IPTC +KMOD +LIBCRYPTSETUP +LIBFDISK +PCRE2 -PWQUALITY +P11KIT +QRE
NCODE +TPM2 +BZIP2 +LZ4 +XZ +ZLIB +ZSTD -BPF_FRAMEWORK -XKBCOMMON +UTMP +SYSVINIT default-hiera
rchy=unified
  서비스 시작/중지/재시작
sudo systemctl  [start ¦ stop ¦ restart ¦ reload]  nginx.service
  유닛을 사용 활성화/비활성화 하기
sudo systemctl  [enable ¦ disable] nginx.service
  유닛 조회하기
$ systemctl --no-pager  list-units          # active 인 것만
  UNIT                                LOAD   ACTIVE SUB       DESCRIPTION
  proc-sys-fs-binfmt_misc.automount   loaded active running   Arbitrary Executable File Formats Fil…
  sys-devices-platform-serial8250-seri… loaded active plugged   /sys/devices/platform/serial8250/seri…
  sys-devices-platform-serial8250-seri… loaded active plugged   /sys/devices/platform/serial8250/seri…
  sys-devices-platform-serial8250-seri… loaded active plugged   /sys/devices/platform/serial8250/seri…
  sys-devices-pnp0-00:06-00:06:0-00:06… loaded active plugged   /sys/devices/pnp0/00:06/00:06:0/00:06…
[....]
  nginx.service                       loaded active running   A high performance web server and a r…
[....]
$ systemctl  --no-pager  list-units  --all   # 전체
[...]
● home.mount                        not-found inactive dead      home.mount
  proc-sys-fs-binfmt_misc.mount     loaded    active   mounted   Arbitrary Executable File Formats …
$ systemctl  list-units  --type=service   # 타입이 service 인 것만
$ systemctl  list-units  --type=target    # 타입이 target 인 것만
  basic.target            loaded active active Basic System
  graphical.target        loaded active active Graphical Interface
  multi-user.target       loaded active active Multi-User System
[....]
$ systemctl  list-unit-files     # 설치된 모든 유닛을 리스트
UNIT FILE                                  STATE          PRESET
proc-sys-fs-binfmt_misc.automount          static         -
```

```
-.mount                                    generated       -
[...]
nginx.service                              enabled         enabled
[...]
```

시스템의 현재 기본 target을 표시함. /usr/lib/systemd/system/default.target 심볼릭 링크와 관련이 있다.

```
$ systemctl get-default
graphical.target
$ readlink  /usr/lib/systemd/system/default.target
graphical.target
```

유닛을 인스펙션하기

```
$ systemctl  cat  nginx.service       # 유닛파일을 출력하기. (.service는 생략가능)
$ systemctl list-dependencies  nginx.service       # 유닛의 의존성 트리를 보기
nginx.service
● ├─system.slice
● ├─network-online.target
● │ └─systemd-networkd-wait-online.service
● └─sysinit.target
●   ├─apparmor.service
●   ├─blk-availability.service
●   ├─dev-hugepages.mount
[...]
$ systemctl list-dependencies --all nginx.service  # 의존성 트리를 모두 보기(recursive)
$ systemctl show  nginx.service   # 저수준의 상세 정보를 보기
Type=forking
ExitType=main
Restart=no
[...]
$ systemctl list-dependencies timers.target   # timers.target과 연결된 타이머들을 확인
timers.target
○ ├─apport-autoreport.timer
● ├─apt-daily-upgrade.timer
● ├─apt-daily.timer
● ├─dpkg-db-backup.timer
● ├─e2scrub_all.timer
● ├─fstrim.timer
[...]
● └─update-notifier-motd.timer
```

유닛 파일을 수정하기

```
$ systemctl edit  nginx.service   # 편집하기
$ systemctl edit --full  nginx.service
$ systemctl  daemon-reload   # 수정한 이후에 변경 내용을 적용하기
```

부팅시간 및 문제 있는 서비스 확인 [5]. 시스템 부팅 과정을 시각적으로 표현하는 SVG 그래픽을 생성

```
$ systemd-analyze  plot  > plot.svg
```

5) https://www.freedesktop.org/software/systemd/man/systemd-analyze.html

시스템 부팅 과정에서 시간이 가장 오래 걸리는 유닛들의 연쇄를 트리 형태로 보여주는 도구

```
$ systemd-analyze  critical-chain
The time when unit became active or started is printed after the "@" character.
The time the unit took to start is printed after the "+" character.

graphical.target @55.413s
└─multi-user.target @55.412s
  └─snapd.seeded.service @42.573s +12.836s
    └─basic.target @42.246s
      └─sockets.target @42.245s
        └─snapd.socket @42.219s +24ms
          └─sysinit.target @42.150s
            └─cloud-init.service @23.801s +18.334s
              └─systemd-networkd-wait-online.service @22.222s +1.572s
                └─systemd-networkd.service @21.960s +253ms
                  └─network-pre.target @21.952s
                    └─cloud-init-local.service @12.352s +9.598s
                      └─systemd-remount-fs.service @6.645s +782ms
                        └─systemd-fsck-root.service @6.420s +219ms
                          └─systemd-journald.socket @6.299s
                            └─-.mount @6.208s
                              └─-.slice @6.208s
```

시스템 유닛의 의존성 그래프를 생성하는 도구

```
$ systemd-analyze dot  > plot.dot
$ systemd-analyze dot --to-pattern='*.target' --from-pattern='*.target' > plot.dot
$ cat plot.dot | dot -Tsvg > plot.svg
$ systemd-analyze dot 'cloud-init.*' | dot -Tsvg > cloud-init.svg
```

화살표의 색깔 green=After, red=Conflicts, grey=Wants, black=Requires

※ systemd 에서 target은 시스템의 특정 상태나 목표를 나타내는 유닛으로, 여러 서비스와 다른 유닛들을 논리적으로 그룹화 한 것이다. multi-user.target은 다중 사용자 텍스트 모드로 SysV의 runlevel 3과 유사하고, graphical.target은 GUI 모드로 SysV의 runlevel 5와 유사함.

※ systemd 소스 코드 받기[6] & 빌드

```
$ git clone https://github.com/systemd/systemd.git
$ cd systemd
$ meson  setup  builddir
$ cd builddir ;  ninja -v
```

- systemd를 테스트 모드로 실행하기. --test 는 실제 서비스를 시작하거나 중지하지 않으며, 구성 파일만 로드하고 검증한다. 로드된 유닛들의 목록을 표시함

```
$ /usr/lib/systemd/systemd --help
$ /usr/lib/systemd/systemd --user --test
Queued start job for default target default.target.
Loaded units and determined initial transaction in 50ms.
-> By units:
        -> Unit dev-disk-by\x2ddiskseq-3.device:
                Description: /dev/disk/by-diskseq/3
                Instance: n/a
                Unit Load State: loaded
[...]
-> By jobs:
        -> Job 3:
                Action: sockets.target -> start
                State: waiting
                Irreversible: no
                May GC: no
[...]
```

2.3 Systemd 서비스 등록

- 파이썬으로 간단한 웹서버를 만들고, systemd 서비스로 등록하기

```
$ more  /opt/my_server/my_server.py
from http.server import BaseHTTPRequestHandler, HTTPServer
hostName = "localhost"
```

6) https://github.com/systemd/systemd

```
default_serverPort = 8080      # 기본 서비스 포트  8080 포트

class MyServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes("Hello\n", "utf-8")) # 메시지를 출력

if __name__ == "__main__":
  port = default_serverPort
  if len(sys.argv) > 1:
    port = int(sys.argv[1])  # 서비스 포트를 입력받음
    print("port : ", port)

  webServer = HTTPServer((hostName, port), MyServer)    # 웹서버를 실행
  print("Server started http://%s:%s" % (hostName, port))
  try:
    webServer.serve_forever()
  except KeyboardInterrupt:
    pass
  webServer.server_close()
```

```
$ python3  /opt/my_server/my_server.py  8080
port :  8080
Server started http://localhost:8080
127.0.0.1 - - [10/Oct/2024 18:15:53] "GET /hello HTTP/1.1" 200 -
^C
```

```
 # 다른 터미널에서 http 요청
*$ curl localhost:8080/hello
Hello
```

```
$ sudo  vim  /etc/systemd/system/my-server.service   # my-server의 유닛 파일
[Unit]
Description=My Server
After=network.target
StartLimitIntervalSec=0
[Service]
Type=simple
Environment=ENV=production
Environment="PORT=7070"    # 포트번호를 7070 으로 설정
ExecStart=/usr/bin/python3  /opt/my_server/my_server.py ${PORT}   # 실행할 프로그램
User=ubuntu     # 사용자
Restart=always
RestartSec=1
[Install]
WantedBy=multi-user.target
```

```
$ sudo  systemctl  daemon-reload
$ sudo  systemctl  start   my-server
$ sudo  systemctl  enable  my-server
Created symlink /etc/systemd/system/multi-user.target.wants/my-server.service → /etc/systemd/sy
stem/my-server.service.

$ sudo  systemctl  status  my-server
● my-server.service - My Server
     Loaded: loaded (/etc/systemd/system/my-server.service; disabled; preset: enabled)
     Active: active (running) since Sat 2024-10-19 02:12:06 UTC; 13s ago
   Main PID: 40067 (python3)
      Tasks: 1 (limit: 9507)
```

```
     Memory: 8.8M (peak: 9.2M) <-------- 메모리 및 CPU사용량
        CPU: 56ms
     CGroup: /system.slice/my-server.service
             └─40067 /usr/bin/python3 /opt/my_server/my_server.py 7070

Oct 19 02:12:06 ip-172-31-3-131 systemd[1]: Started my-server.service - My Server.

$ netstat -ant ¦ egrep "7070¦8080"  # .service 파일에 설정한 7070 포트에서 서비스되는지 확인
tcp        0      0 127.0.0.1:7070          0.0.0.0:*               LISTEN
$ curl localhost:7070/hello    # 서비스를 요청
Hello

$ journalctl --no-pager -u my-server.service  ¦ tail
[...]
Oct 19 02:14:30 ip-172-31-3-131 python3[40067]: 127.0.0.1 - - [19/Oct/2024 02:14:30] "GET /hell
o HTTP/1.1" 200 -
  드롭인(drop-in) 방식으로 포트 설정을 변경하기. 드롭인 설정이 원본 설정 파일보다 우선함
$ sudo mkdir  /etc/systemd/system/my-server.service.d
$ sudo vi  /etc/systemd/system/my-server.service.d/override.conf
[Service]
Environment="PORT=9090"   # 서비스 포트 설정을 9090으로 변경함
$ sudo systemctl  stop    my-server   # 서비스 중지 후 설정을 다시 로드하기
$ sudo systemctl  daemon-reload
$ sudo systemctl  start   my-server
$ curl localhost:9090/hello          # 드랍인 설정이 적용되었는지 확인
Hello
```

※ journalctl 은 systemd의 로그를 조회하고 관리하는 명령어임. 로그 파일은 /var/log/journal/ 경로에 저장됨. /run/log/journal/ 경로는 휘발성 저장소로 링버퍼 구조이며, 시스템 재부팅시 로그가 유지되지 않는다. 로깅 데이터는 저널의 journald 서비스에서 수집, 저장, 처리함.

```
$ journalctl -u my-server.service   #   특정 서비스의 로그 보기
$ journalctl -f     # 실시간 로그 모니터링
$ sudo journalctl --vacuum-time=2d   # 오래된 로그를 삭제
$ sudo journalctl --vacuum-size=1G   # 특정 크기로 로그 제한
```

- 타이머 서비스를 등록하기

```
$ sudo vim /opt/mytimer.sh
#!/usr/bin/env bash
echo "i am running at $(date)"     # 현재 시간을 출력하는 쉘 스크립트
exit 0
$ sudo vim /etc/systemd/system/mytimer.service
[Unit]
Description=Run mytimer service
[Service]
ExecStart=/opt/mytimer.sh        # 위 쉘스크립트를 실행하는 서비스
$ sudo vim /etc/systemd/system/mytimer.timer
[Unit]
Description=mytimer service timer
[Timer]       # 서비스를 실행하는 타이머
OnBootSec=0min
OnCalendar=*:*:0/30
Unit=mytimer.service
```

```
[Install]
WantedBy=multi-user.target
  systemd-analyze calendar [CALENDAR_EXPRESSION]  CALENDAR_EXPRESSION을 parsing 한다.
$ systemd-analyze calendar  "*:*:0/30"
  Original form: *:*:0/30
Normalized form: *-*-* *:*:00/30
    Next elapse: Mon 2024-10-21 03:34:30 UTC
       From now: 5s left   <----- 다음 트리거까지 남은 시간
$ journalctl -fu mytimer.service    # 실시간으로 로그를 확인
Oct 21 03:36:17 ip-172-31-3-131 systemd[1]: Started mytimer.service - Run mytimer service.
Oct 21 03:36:17 ip-172-31-3-131 mytimer.sh[2016]: i am running at Mon Oct 21 03:36:17 UTC 2024
Oct 21 03:36:17 ip-172-31-3-131 systemd[1]: mytimer.service: Deactivated successfully.
Oct 21 03:37:17 ip-172-31-3-131 systemd[1]: Started mytimer.service - Run mytimer service.
Oct 21 03:37:17 ip-172-31-3-131 mytimer.sh[2025]: i am running at Mon Oct 21 03:37:17 UTC 2024
Oct 21 03:37:17 ip-172-31-3-131 systemd[1]: mytimer.service: Deactivated successfully.
[...]
$ systemctl list-timers    # systemd 타이머 목록을 조회하기
NEXT                            LEFT LAST                              PASSED UNIT        >
Mon 2024-10-21 04:50:00 UTC     6min Mon 2024-10-21 04:40:27 UTC  3min 8s ago sysstat-coll>
Mon 2024-10-21 05:11:45 UTC    28min Mon 2024-10-21 04:18:17 UTC    25min ago fwupd-refres>
Mon 2024-10-21 06:09:26 UTC 1h 25min Sun 2024-10-20 06:17:01 UTC            - apt-daily-up>
[...]
-                                  - Mon 2024-10-21 04:43:36 UTC     19ms ago mytimer.time>
```

3. 제어그룹(cgroup)

3.1 시스템 자원 제한 방법

- cgroup 이전의 리눅스 시스템에도 CPU 사용을 제한하는 방법이 존재함. ① nice/renice 명령: 프로세서의 우선순위를 조정하여 간접적으로 CPU 사용을 제어. -20(가장 높은)부터 19(가장 낮은)까지의 우선순위 값을 사용. ② ulimit : 쉘 및 그 자식 프로세스에 대한 리소스 제한을 설정 ③ cpulimit 도구[7]: 특정 프로세서의 CPU 사용률을 직접 제한할 수 있는 도구 ④ taskset 명령: 프로세스를 특정 CPU 코어에 할당하여 간접적으로 CPU 사용을 제어
- nice 유틸리티는 프로세스 스케줄링 메커니즘을 이용하여 간접적으로 CPU 사용을 제어함. 리눅스 커널은 완전 공정 스케줄러(CFS, Completely Fair Scheduler)를 사용. 각 프로세스별로 가상 실행시간(vruntime, 실제 실행 시간을 정규화 한 값)을 설정하여 스케쥴링에 이용, 시스템 부하가 낮을 때는 nice의 효과가 덜 두드러짐. 부하가 높을 수록 nice값의 영향이 명확해 진다. 메모리나 I/O등 다른 리소스에는 영향을 주지 않음

- cpu 부하를 발생시키는 프로그램

```
//  busy_single.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <time.h>
#include <sys/times.h>
#include <sys/resource.h>
#define COUNTER_SIZE  0xFFFFFFF   // 카운터가 이 값에 도달하면 초기화함

double calculate_cpu_usage() {  // 현재 프로세스의 cpu 점유율을 계산
  static clock_t last_cpu, last_sys_cpu, last_user_cpu;
  static int initialized = 0;
  struct tms time_sample;
  clock_t now;
  now = times(&time_sample);  // 현재 프로세스의 cpu 시간 정보를 얻는다.  times [8]
  if (!initialized) {
    last_cpu = now;
    last_sys_cpu = time_sample.tms_stime;  // 프로세스가 커널모드에서 실행한 cpu시간
    last_user_cpu = time_sample.tms_utime; // 프로세스가 사용자모드에서 실행한 cpu시간
    initialized = 1;
    return 0.0;
  }
  double percent = ((time_sample.tms_stime - last_sys_cpu) +
                    (time_sample.tms_utime - last_user_cpu)) /
                   (double)(now - last_cpu);  // cpu 점유율계산 = cpu clcoks  / wallclock
  percent *= 100;
  last_cpu = now;
  last_sys_cpu = time_sample.tms_stime;
  last_user_cpu = time_sample.tms_utime;
  return percent;
}
void *loop(  ) {
  int count = 0;
```

7) https://manpages.ubuntu.com/manpages/xenial/man1/cpulimit.1.html

```
  double cpu_usage = calculate_cpu_usage(); // initialize
  while (1) {
    count++; // 카운터를 증가
    if (count == COUNTER_SIZE) {
      cpu_usage = calculate_cpu_usage();
      int nice_val = getpriority(PRIO_PROCESS, 0);  // 현재 프로세스의 nice 값
      printf("nice:%d CPU Usage: %.2f%%\n", nice_val, cpu_usage);
    }
    if (count == COUNTER_SIZE) count = 0; // 최대값에서 0으로 초기화하여 반복
  }
}
int main(int argc, char **argv) {
  pid_t pid = getpid(); // 현재 프로세스의 pid값
  printf("pid: %d\n", pid);
  loop();
  return 0;
}
$ gcc -o busy_single  busy_single.c  -Wall -g
$ ./busy_single
pid: 2484
nice:0 CPU Usage: 100.00%
nice:0 CPU Usage: 100.00%
nice:0 CPU Usage: 100.00%
nice:0 CPU Usage: 100.00%
^C
$ nice -n 0  ./busy_single  &   # nice 값을 다르게 하여 3개의 프로세스를 백그라운드로 실행
$ nice -n 5  ./busy_single  &
$ nice -n 10  ./busy_single  &
*$ top -b -n 100 ¦ egrep "PID¦busy"    # 다른 터미널에서 cpu점유율을 모니터링
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4087 ubuntu    20   0    2680   1536   1536 R  68.2   0.2   0:15.07 busy_si+
   4088 ubuntu    25   5    2680   1536   1536 R  22.2   0.2   0:03.64 busy_si+
   4089 ubuntu    30  10    2680   1536   1536 R   8.0   0.2   0:00.89 busy_si+
   nice 값이 작을 수록 cpu 점유율이 높음을 주목.
  (이 결과는 single core 에서 실행한 결과로 core 가 여러개인 시스템에서는 %cpu 합이 100%를 넘어
갈 수 있음)
  nice 10인 프로세스의 nice 값을 -5 로 변경. nice 값을 감소(우선순위를 증기)하기 위해서는 root
권한이 필요함
*$ sudo renice -n -5 -p 4089
4089 (process ID) old priority 10, new priority -5
*$ top -b -n 100 ¦ egrep "PID¦busy"
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4089 ubuntu    15  -5    2680   1536   1536 R  69.7   0.2   0:15.90 busy_si+
   4087 ubuntu    20   0    2680   1536   1536 R  23.4   0.2   1:02.88 busy_si+
   4088 ubuntu    25   5    2680   1536   1536 R   6.9   0.2   0:19.28 busy_si+
     nice 값이 작을 수록 cpu 점유율이 높음을 주목
```

- 멀티코어 시스템에서 프로세스의 코어를 제한하려면 taskset 명령을 이용한다. 위에서 busy_single 프로그램을 실행하는 3개의 프로세스를 동일한 코어 1개로 제한하기:

```
형식: taskset  -pc <core번호> <PID>                          # core 0번으로 제한하기
$ taskset  -pc 0 4087 ; taskset  -pc 0 4088 ; taskset  -pc 0 4089
  프로세스 3개가 1개 코어에서 실행하면서 cpu 점유율 합이 100%가 됨
```

8) https://man7.org/linux/man-pages/man2/times.2.html

- cpulimit[9]은 프로세스의 CPU 사용량을 제한하는 도구임. 특정 프로그램이 너무 많은 CPU 사이클을 소모하지 않도록 하려는 경우 일괄 작업을 제어하는 데 유용하다.
- cpulimit의 동작원리: 지정된 프로세스를 지속적으로 모니터링 하여, 실제 CPU 사용량을 주기적으로 측정. 설정된 제한 값을 초과하면 해당 프로세스를 중지(pause)시키고, 일정 시간 후에 다시 재개(resume)한다.
- cpulimit 테스트

```
$ sudo apt-get install cpulimit
$ git clone https://git.launchpad.net/ubuntu/+source/cpulimit
$ cd cpulimit
$ make
$ sudo  cp  cpulimit /usr/bin/cpulimit
```

```
$ cpulimit
Error: You must specify a target process
CPUlimit version 3.1
Usage: cpulimit TARGET [OPTIONS...] [-- PROGRAM]
   TARGET must be exactly one of these:
      -p, --pid=N        pid of the process
      -e, --exe=FILE     name of the executable program file
                         The -e option only works when
                         cpulimit is run with admin rights.
      -P, --path=PATH    absolute path name of the
                         executable program file
[...]
```

```
$ cd test ; make
   busy 프로그램의 cpu 사용률을 90(-l)으로 제한. -f foreground 로 실행
$ cpulimit -l 90 -f -v -- ./busy_simple
Process 44650 detected
[...]
*$ ps -ao pid,ppid,pcpu,pmem,rss,vsz,cmd  ¦ egrep "busy¦PID"
    PID    PPID %CPU %MEM   RSS    VSZ CMD
  98144   93899  0.0  0.0  1664   2696 cpulimit -l 90 -f -- ./busy_simple
  98145   98144 89.7  0.0  1024   2548 ./busy_simple
   2개의 스레드에서 실행 사용률을 150%로 제한
$ cpulimit -l 150 -f  -- ./busy_thread 2
Process 98191 detected
pid: 98191
threads: 2
threadId = 0
threadId = 1
17:28:30 nice:0 CPU Usage: 153.26%
17:28:31 nice:0 CPU Usage: 156.99%
17:28:32 nice:0 CPU Usage: 155.91%
[...]
```

3.2 cgroup 소개

- cgroup(control group)는 리눅스 커널에서 제공하는 리소스 관리 및 제어 메커니즘임. CPU, 메모리, 디스크 I/O, 네트워크 대역폭 등의 시스템 리소스를 프로세스 그룹 단위로 할당하고 제한할 수 있다.

9) https://github.com/opsengine/cpulimit

cgroup는 계층적으로 구성되어 부모의 특성을 자식이 상속 받을 수 있음. Docker 등의 컨테이너 기술은 cgroup를 기반으로 하여 컨테이너의 리소스 사용을 제한하고 있음

- 구글의 엔지니어들이 2006년에 개발을 시작하였으며, 초기에는 "프로세스 컨테이너"라는 이름으로 명명했으나, 컨테이너라는 용어로 인한 혼란을 피하기 위해 컨트롤 그룹(control group)으로 변경하였다. cgroup의 첫 번째 버전(v1)[10]은 2008년 1월에 출시된 Linux 커널 2.6.24 버전에 공식적으로 포함됨. cgroup v2[11]는 2016년에 처음 등장하여 점진적으로 발전되고 있음. 커널 4.5부터 cgroup v2가 공식 릴리즈되었고, 커널 5.8부터 cgroup v2 가 완전히 지원되기 시작함. 현재 커널의 cgroup이 v1인지 v2인지 확인 하는 방법은 cgroup의 파일 시스템 타입을 확인:

```
$ stat -fc %T /sys/fs/cgroup/    # cgroup2fs 이면 v2, tmpfs 이면 v1 임을 확인
$ mount -l ¦ grep cgroup       # 시스템 마운트 정보를 확인
```

- /sys/fs/cgroup 디렉터리를 통해서 cgroup을 관리할 수 있음. 하위 디렉터리 구조는 v1과 v2가 다름.

```
  cgroup v1의 디렉터리 구조
/sys/fs/cgroup/           각 컨트롤러(subsystem) 별로 별도의 디렉터리가 존재
├── cpu                  각 컨트롤러 디렉터리내에 해당 리소스에 대한 제어 파일들이 있음
│   ├── cpu.shares
│   ├── cpu.cfs_period_us
│   └── my_cgroup/   <------ 사용자가 생성한 그룹
├── memory
│   ├── memory.limit_in_bytes
│   └── my_cgroup/    <------ 사용자가 생성한 그룹
└── blkio
    ├── blkio.weight
    └── my_cgroup/
  cgroup v2의 디렉터리 구조
/sys/fs/cgroup/             모든 컨트롤러가 단일 계층 구조로 통합됨
├── cgroup.controllers    루트 디렉터리에 모든 cgroup 관련 파일과 디렉터리가 위치함
├── cgroup.subtree_control
├── cpu.weight
├── memory.max
├── io.weight
└── my_cgroup/   <------ 사용자가 생성한 그룹
    ├── cgroup.controllers
    ├── cgroup.subtree_control
    └── ...
```

- cgroup 컨트롤러는 특정 시스템 리소스의 사용을 제어하고 모니터링 하는 역할을 한다. 컨트롤러의 종류는 cpu, memory, blkio, cpuacct, cpuset, devices, freezer, net_cls, ns 등이 있음

컨트롤러 서브시스템	설명
blkio	물리 드라이브와 같은 블록 장치에 대한 입력/출력 액세스에 제한을 설정
cpu	CPU에 cgroup 작업 액세스를 제공하기 위해 스케줄러를 사용
cpuacct	CPU 자원에 대한 보고서를 자동으로 생성
cpuset	개별 CPU (멀티코어 시스템에서) 및 메모리 노드를 cgroup의 작업에 할당
devices	cgroup의 작업 단위로 장치에 대한 액세스를 허용하거나 거부함
freezer	cgroup의 작업을 일시 중지하거나 다시 시작함
memory	cgroup의 작업에서 사용되는 메모리에 대한 제한을 설정하고 사용되는 메모리

10) https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html
11) https://docs.kernel.org/admin-guide/cgroup-v2.html

	자원에 대한 보고서를 생성
net_cls	Linux 트래픽 컨트롤러 (tc)가 특정 cgroup 작업에서 발생하는 패킷을 식별하게 하는 클래식 식별자 (classid)를 사용하여 네트워크 패킷에 태그를 지정
ns	namespace 서브시스템

- 프로세스가 속한 cgroup을 확인하는 방법

```
  /proc/<PID>/cgroup을 확인
$ cat /proc/$$/cgroup            $$는 현재 쉘의 pid 값임  또는  $ cat /proc/self/cgroup
0::/user.slice/user-1000.slice/session-1217.scope
  0:: 는 cgroup v2 계층을 나타냄
  /user.slice 는 사용자 세션 및 사용자 서비스를 위한 슬라이스 임
  /user-1000.slice 는 특정 사용자를 위한 슬라이스를 의미. 1000은 사용자의 UID 값임
  /session-1217.scope 는 특정사용자의 세션을 의미함
```

- cgroup 계층 구조를 조회하기. 새로운 cgroup을 생성하기

```
 형식 systemd-cgls  [options] <CGROUP>
$ systemd-cgls  /user.slice/user-1000.slice
CGroup /user.slice/user-1000.slice:
├─user@1000.service …
│ ├─session.slice
[...]
└─session-5.scope
  ├─1854 sshd: ubuntu [priv]
  └─1909 sshd: ubuntu@pts/0
  최상위 cgroup의 컨트롤러 목록
$ cat  /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
$ cat  /sys/fs/cgroup/cgroup.subtree_control
cpu memory pids

$ sudo mkdir /sys/fs/cgroup/mycgroup      # 새로운 cgroup을 생성
 # 사용가능한 컨트롤러 목록. 해당 cgroup에서 사용 가능한 모든 서브시스템의 목록을 보여줌
 cgroup.controllers 는 읽기 전용이며, 여기에 나열된 컨트롤러만 subtree_control 파일을 통해 활성
화 가능
$ cat  /sys/fs/cgroup/mycgroup/cgroup.controllers
cpu memory pids
$ cat  /sys/fs/cgroup/mycgroup/cgroup.subtree_control  # 비어 있음. cpu를 추가함
$ echo "+cpu" ¦ sudo tee /sys/fs/cgroup/mycgroup/cgroup.subtree_control
+cpu
$ sudo mkdir /sys/fs/cgroup/mycgroup/mysubcgroup      # 계층적 하위 cgroup을 생성
$ cat /sys/fs/cgroup/mycgroup/mysubcgroup/cgroup.controllers
cpu      # 상위 계층의 subtree_control 에 추가한 것만 존재함
   cgroup을 삭제하기 : -r recursive
$ sudo cgdelete -r -g cpu:/mycgroup
$ sudo apt install cgroup-tools  # control group 을 모니터하고 제어하는 도구 12)
$ sudo yum update ; sudo yum install libcgroup libcgroup-tools  # redhat 계열 리눅스
```

3.3 cgroup 자원 제한 방법

- cgroup v2를 사용하여 CPU를 제한하는 방법

12) https://github.com/libcgroup/libcgroup

```
$ mount ¦ grep cgroup2         # cgroup v2 파일 시스템이 마운트되어 있는지 확인

$ sudo mkdir /sys/fs/cgroup/mycgroup   # 새로운 cgroup을 생성
$ cat /sys/fs/cgroup/mycgroup/cgroup.controllers
cpu memory pids      # cpu 서브시스템 활성화 여부를 확인

  # CPU 사용량을 제한함. (50%로 제한)
$ echo "50000 100000" ¦ sudo tee /sys/fs/cgroup/mycgroup/cpu.max
50000 100000
  # cgroup 내에서 명령을 실행하기
$ sudo cgexec -g cpu:mycgroup  ./busy_single
pid: 21618
nice:0 CPU Usage: 50.34%
nice:0 CPU Usage: 50.00%
[...]
*$ systemd-cgls /mycgroup     # 다른터미널에서 확인/mycgroup 에서 해당 프로세스를 확인
CGroup /mycgroup:
└─21618 ./busy_single

  systemd-cgtop 명령은 자원 사용량에 따라서 cgroup을 모니터링 한다. -c는 cpu load순서
*$ systemd-cgtop -c
CGroup                      Tasks   %CPU   Memory  Input/s Output/s
/                             172   49.9   496.8M        -        -
mycgroup                        2   49.6   216.0K        -        -
user.slice                     14    0.8   199.7M        -        -
[...]

$ cgget  -g cpu:mycgroup    # cgget : 특정 cgroup의 CPU 컨트롤러 설정을 확인
mycgroup:
cpu.weight: 100
cpu.stat: usage_usec 145774346
        user_usec 145764353    <<<<< 사용자 모드에서 사용한 cpu time
        system_usec 9993       <<<<< 커널 모드에서 사용한 cpu time
        core_sched.force_idle_usec 0
        nr_periods 2325
        nr_throttled 2318
        throttled_usec 177859548
        nr_bursts 0
        burst_usec 0
cpu.weight.nice: 0
   cpu.pressure는 cgroup 내의 프로세스들이 CPU 리소스를 얻기 위해 경쟁하는 정도를 나타냄. 높은
pressure 값은 cgroup 내의 프로세스들이 CPU시간을 충분히 얻지 못하고 있음을 의미
cpu.pressure: some avg10=42.14 avg60=13.52 avg300=3.15 total=117664765
        full avg10=42.14 avg60=13.52 avg300=3.15 total=117664765
cpu.idle: 0
cpu.stat.local: throttled_usec 177859548
cpu.max.burst: 0
cpu.max: 50000 100000   <<<<< 설정된 max
cpu.uclamp.min: 0.00
cpu.uclamp.max: max

  # CPU 사용량을 제한을 변경 (=> 70%)   두번째 숫자의 단위는 마이크로초(us)임
*$ sudo cgset -r cpu.max="70000 100000" mycgroup
 또는
*$ echo "70000 100000" ¦ sudo tee /sys/fs/cgroup/mycgroup/cpu.max
```

```
70000 100000
[...]
nice:0 CPU Usage: 50.33%
nice:0 CPU Usage: 50.00%
nice:0 CPU Usage: 66.67%  <------ cpu 사용률이 증가함
nice:0 CPU Usage: 69.72%
[...]

**$ sudo cgexec -g cpu:mycgroup  ./busy_single   # 같은 cgroup 내에서 하나 더 실행하기
pid: 21641
nice:0 CPU Usage: 34.26%
nice:0 CPU Usage: 35.85%     cpu.max 70%이 동일한 두 프로세스의 경쟁에 의해 절반으로 줄어듦
[...]
*$ systemd-cgls /mycgroup     # mycgroup 에서 해당 프로세스를 확인
CGroup /mycgroup:
├─21618 ./busy_single  <---- 처음 실행한 것
└─21641 ./busy_single  <---- 두번째 실행한 것

$ echo "max 1000000" ¦ sudo tee /sys/fs/cgroup/mycgroup/cpu.max   # 제한이 없음
max 1000000
[...]
nice:0 CPU Usage: 35.07%
nice:0 CPU Usage: 39.27%    mycgroup에 속한 2개의 프로세스가 cpu.max 제한이 없어지면서
nice:0 CPU Usage: 100.00%   <--- 각각 100%를 사용. 멀티코어이므로 각각 서로 다른 코어를 사용
nice:0 CPU Usage: 100.00%
[...]
```

※ 시스템 부팅 이후에도 앞에서 만든 cgroup을 유지하려면 systemd 서비스로 등록해 준다.

```
$ sudo vi /opt/make_mycgroup.sh        # cgroup을 생성하는 쉘스크립트
#!/bin/sh
mkdir -p /sys/fs/cgroup/mycgroup
sleep 1
echo "50000 100000" >  /sys/fs/cgroup/mycgroup/cpu.max
$ sudo vi /etc/systemd/system/mycgroup.service
[Unit]
Description=Create custom cgroup
After=sys-fs-cgroup.mount

[Service]
Type=oneshot      # 위 쉘스크립트를 실행하는 서비스 Type은 oneshot 으로 설정
ExecStart=/opt/make_mycgroup.sh
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
$ sudo systemctl daemon-reload
$ sudo systemctl enable  mycgroup.service
Created symlink /etc/systemd/system/multi-user.target.wants/mycgroup.service → /etc/systemd/sys
tem/mycgroup.service.
$ reboot        시스템 재부팅
[...]
$ sudo systemctl status  mycgroup.service
● mycgroup.service - Create custom cgroup     재부팅 이후에 cgroup을 확인
     Loaded: loaded (/etc/systemd/system/mycgroup.service; enabled; preset: enabled)
```

```
     Active: active (exited) since Tue 2024-10-22 06:15:18 UTC; 4min 24s ago
    Process: 543 ExecStart=/opt/make_mycgroup.sh (code=exited, status=0/SUCCESS)
   Main PID: 543 (code=exited, status=0/SUCCESS)
        CPU: 30ms

Oct 22 06:15:16 ip-172-31-3-131 systemd[1]: Starting mycgroup.service - Create custom cgroup...
Oct 22 06:15:17 ip-172-31-3-131 sudo[544]:     root : PWD=/ ; USER=root ; COMMAND=/usr/bin/mkdi
r -p /sys/fs/cgroup/mycgroup
```

- cgroup v2를 사용하여 메모리 사용량을 제한하기

```
$ cat /sys/fs/cgroup/mycgroup/cgroup.controllers
cpu memory pids      <--- memory 서브시스템 활성화 여부를 확인

  현재 쉘의 프로세스 pid($$)를 cgroup.procs 에 추가하면 현재 쉘이 mycgroup에 속하게 됨
$ echo $$ ¦ sudo tee /sys/fs/cgroup/mycgroup/cgroup.procs
$ cat /proc/$$/cgroup
0::/mycgroup

  메모리 사용량을 1024M = 1GB 로 제한
$ echo "1024M" ¦ sudo tee /sys/fs/cgroup/mycgroup/memory.max
1024M
$ cat /sys/fs/cgroup/mycgroup/memory.max
1073741824
$ ./malloc1      <-- 메모리 할당을 테스트하는 프로그램
사용법: ./malloc1 <메모리_크기(MB)>
$ ./malloc1 1000     <-- 1000은 1024를 넘지 않으므로 할당 성공
1048576000 바이트의 메모리가 성공적으로 할당되었습니다.
$ ./malloc1 1100     <-- 1100은 1024를 초과
Killed
  메모리 사용량을 제한을 2GB 로 변경
$ echo "2048M" ¦ sudo tee /sys/fs/cgroup/mycgroup/memory.max
$ echo max ¦ sudo tee /sys/fs/cgroup/mycgroup/memory.max   # 메모리 제한을 해제(max로 설정)
   현재 쉘이 속해 있는 cgroup을 삭제하면, 최상위(/)로 변경됨
$ sudo cgdelete -r -g memory:mycgroup
$ cat /proc/self/cgroup
0::/
```

- systemd-run 으로 CPU 사용량을 제한하기

```
$ man  systemd-run
    systemd-run - Run programs in transient scope units, service units, or path-, socket-, or t
imer-triggered service units
SYNOPSIS
       systemd-run [OPTIONS...] COMMAND [ARGS...]
       systemd-run [OPTIONS...] [PATH OPTIONS...] {COMMAND} [ARGS...]
       systemd-run [OPTIONS...] [SOCKET OPTIONS...] {COMMAND} [ARGS...]
       systemd-run [OPTIONS...] [TIMER OPTIONS...] {COMMAND} [ARGS...]
[...]

   CPUQuota=50% 으로 하여 CPU 사용량을 제한한다.
$ sudo systemd-run --slice=cpu-limited.slice --property=CPUQuota=50% ./busy_single
Running as unit: run-r12b916a31ed447ad83dc21bc8e298336.service; invocation ID: b6d53a6a1ff344d8
b7435f561fc4da54
   run-*****.service 라는 이름으로 서비스가 실행됨. systemctl status 로 상태를 조회함
```

```
$ systemctl status run-r12b916a31ed447ad83dc21bc8e298336.service --no-pager -l
● run-r12b916a31ed447ad83dc21bc8e298336.service - /home/ubuntu/linux-drill/cgroup/./busy_single
     Loaded: loaded (/run/systemd/transient/run-r12b916a31ed447ad83dc21bc8e298336.service; tran
sient)
  Transient: yes
     Active: active (running) since Mon 2024-10-21 05:37:24 UTC; 2min 7s ago
   Main PID: 3458 (busy_single)
      Tasks: 1 (limit: 9507)
     Memory: 164.0K (peak: 328.0K)
        CPU: 1min 3.838s
     CGroup: /_cpu.slice/cpu-limited.slice/run-r12b916a31ed447ad83dc21bc8e298336.service
             └─3458 /home/ubuntu/linux-drill/cgroup/./busy_single

Oct 21 05:37:24 ip-172-31-3-131 systemd[1]: Started run-r12b916a31ed447ad83dc21bc8e298336.servi
ce - /home/ubuntu/linux-drill/cgrou…sy_single.
  서비스 중지하기
$ sudo systemctl stop  run-r12b916a31ed447ad83dc21bc8e298336.service
```

- systemd-run 으로 메모리 사용량을 제한하기

```
// malloc2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAXBLOCKS  2048  // 2048 * 1MB = 2GB (max)
int main( int argc, char *argv[] ) {
  if (argc != 2) {
    printf("사용법: %s <메모리_크기(MB)>\n", argv[0]);
    return 1;
  }
  size_t size = atoi(argv[1]);   // 할당할 메모리의 크기를 입력받음 MB 단위
  int *ptr[MAXBLOCKS];

  size_t size1m = 1024 * 1024; // 1 MB
  int i;
  for (i = 0; i < size; i++) {  // 1MB 단위로 반복하여 메모리를 할당함
    printf("malloc 1MB memory ... %d\n", i);
    ptr[i] = (int *)malloc(size1m);
    if (ptr[i] == NULL) {
      printf("메모리 할당에 실패했습니다.\n");
      return 1;
    }
  }
  printf("1MB * %zu 메모리가 성공적으로 할당되었습니다.\n", size);
  // 할당된 메모리에 쓰기
  for (i = 0; i < size; i++) {
    for (size_t j = 0; j < size1m / sizeof(int); j++) {
      ptr[i][j] = j;
    }
  }
  for (i = 0; i < size; i++) {// 할당된 메모리를 모두 해제
    free(ptr[i]);
  }
  printf("메모리가 해제되었습니다.\n");
  return 0;
```

```
}
  메모리 제한을 100M로 설정, MemoryHigh를 80MB로 하여 실행
$ systemd-run --user --scope -p MemoryLimit=100M -p MemoryHigh=80M  ./malloc2  70
Running as unit: run-u21.scope; invocation ID: a5ef0cb343e74079b9bb74dbd46d3f52
malloc 1MB memory ... 0
[...]
malloc 1MB memory ... 69
1MB * 70 메모리가 성공적으로 할당되었습니다.
메모리가 해제되었습니다.
$ systemd-run --user --scope -p MemoryLimit=100M -p MemoryHigh=80M  ./malloc2  90
Running as unit: run-u23.scope; invocation ID: d56be57960d3479c8a4af0f82d92cd4f
malloc 1MB memory ... 0
[...]
malloc 1MB memory ... 89
1MB * 90 메모리가 성공적으로 할당되었습니다.
^C
    MemoryHigh 로 설정된 80M를 초과하여 systemd는 메모리를 회수하려고 시도함. ^C로 강제 종료
```

※ cgsnapshot 명령은 현재 cgroup 설정의 스탭샷을 생성하는 명령임. /etc/cgconfig.conf 파일 형식으로 출력한다.

```
$ sudo cgsnapshot -f ./my.conf  cpu
  /etc/cgsnapshot_blacklist.conf 파일이 없다고 나올 경우 만들어 준다( sudo touch /etc/cgsnapsho
t_blacklist.conf )
[...]
group mycgroup {
        cpuset {
        }
        cpu {
                cpu.weight="100";
                cpu.weight.nice="0";
                cpu.pressure="some avg10=29.44 avg60=29.35 avg300=29.11 total=379329675
full avg10=29.44 avg60=29.35 avg300=29.11 total=379329675";
                cpu.idle="0";
                cpu.max.burst="0";
                cpu.max="70000 100000";
                cpu.uclamp.min="0.00";
                cpu.uclamp.max="max";
```

4. 리눅스 네임스페이스

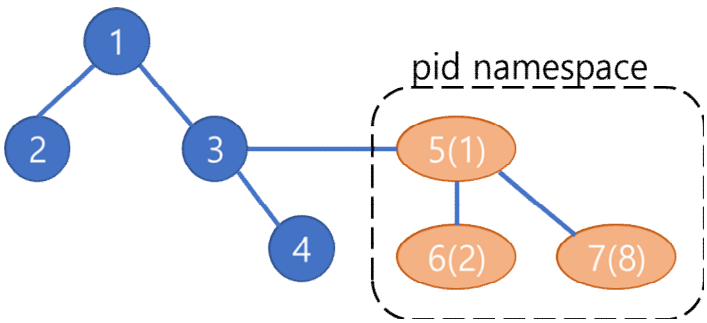
4.1. 네임스페이스 개요

- 리눅스 네임스페이스는 시스템 리소스를 격리하여 프로세스 그룹에 독립적인 환경을 제공하는 커널 기능임. 지원되는 네임스페이스의 유형은 다음과 같다: PID 네임스페이스, 네트워크 네임스페이스(ipc), mount 네임스페이스, UTS 네임스페이스(호스트 이름 및 도메인 이름 격리), 사용자 네임스페이스(user, 사용자 및 그룹 ID를 격리), IPC 네임스페이스(프로세스간 통신), Cgroup(제어 그룹) 네임스페이스
- 네임스페이스는 컨테이너 기술의 기반이 되며, 프로세스 그룹에 독립적인 환경을 제공한다.
- /proc/<PID>/ns 디렉터리에서 현재 프로세스에서 사용하고 있는 네임스페이스의 고유 ID를 확인

```
  1번 init 프로세스의 네임스페이스 정보
$ sudo ls -la /proc/1/ns/
[...]
lrwxrwxrwx 1 root root 0 Oct 22 09:56 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 mnt -> 'mnt:[4026531841]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 Oct 22 07:08 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Oct 22 09:56 uts -> 'uts:[4026531838]'
```

4.2. PID 네임스페이스

- PID는 init 프로세스 1을 시작하여 그 외에 모든 프로세스는 항상 1보다 큰 PID를 부여 받는다. PID 네임스페이스를 분리하면 PID가 다시 1부터 시작하며, 디폴트 네임스페이스와 분리된 PID 네임스페이스에 됨. 분리된 새로운 네임스페이스에서는 PID가 1부터 시작하지만, 디폴트 네임스페이스 관점에서는 1보다 큰 어떤 값을 PID로 가지게 된다.



- unshare 명령은 리눅스에서 프로그램을 새로운 네임스페이스에서 실행한다.

```
형식) unshare [options] [program]
  -m, --mount: 새로운 마운트 네임스페이스 생성
  -u, --uts: 새로운 UTS 네임스페이스 생성
  -i, --ipc: 새로운 IPC 네임스페이스 생성
  -n, --net: 새로운 네트워크 네임스페이스 생성
  -p, --pid: 새로운 PID 네임스페이스 생성
  -U, --user: 새로운 사용자 네임스페이스 생성
```

```
$ ps aux               init 프로세스는 PID 1로 실행됨
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1  22684 13788 ?        Ss   Oct22   0:03 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Oct22   0:00 [kthreadd]
[...]
$ echo $$      현재 쉘의 pid 값을 확인
6138

   PID 네임스페이스 생성하기
    --fork 옵션은 지정된 프로그램을 unshare의 자식 프로세스로 fork하여 실행함.
    --mount-proc 옵션은 실행하기 전에 /proc 파일 시스템을 마운트 지점에 마운트함
$ sudo unshare --pid --fork --mount-proc /bin/bash
    fork 된 네임스페이스에서 쉘의 PID는 1이 된다
# echo $$
1
# ps aux        ps 명령으로 프로세스를 확인
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   8004  4224 pts/1    S    00:05   0:00 /bin/bash
root           9  0.0  0.0  11320  4352 pts/1    R+   00:05   0:00 ps aux
# pstree -p
bash(1)──pstree(30)
# exit       쉘을 종료하므로 네임스페이스가 삭제됨
  위에서 네임스페이스가 생성되어 있는 상태에서 다른 터미널에서 네임스페이스를 조회하기
$ sudo lsns -t pid       타입이 pid인 네임스페이스목록을 조회(lsns)
        NS TYPE NPROCS   PID USER COMMAND
4026531836 pid     123     1 root /sbin/init
4026532295 pid       1  8296 root /bin/bash
```

- unshare 시스템콜[13]을 이용하여 pid 네임스페이스를 생성하는 코드

```
// unshare1.c
#define _GNU_SOURCE
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  int result = unshare(CLONE_NEWPID);  // 프로세스 실행 컨텍스트의 일부를 분리함
    CLONE_NEWPID 는 PID네임스페이스 공유를 해제한다. (sched.h 에 정의)
 // 호출 프로세스에 의해 생성된 첫 번째 자식은 pid 1을 갖게 된다.
  if (result == -1) { perror("unshare failed"); return 1; }

  pid_t pid = fork();
  if (pid == 0) { // child
    printf("child pid = %d\n", getpid());
    printf("child sleep \n");
    sleep(10);

  } else if (pid > 0) { // parent
    printf("pid = %d, Parent getpid() : %d\n", pid, getpid());
    wait(NULL);
    printf("exit program\n");
  }
```

13) https://man7.org/linux/man-pages/man2/unshare.2.html

```
  return 0;
}
$ gcc -o unshare1  unshare1.c
$ sudo ./unshare1          root 권한으로 실행해야 함
pid = 1546, Parent getpid() : 1545
child pid = 1      첫번째 자식 프로세스는 pid 1을 갖는다.
child sleep
exit program
```

4.3. 네트워크 네임스페이스

- 네트워크 네임스페이스 분리는 네트워크 스택을 분리함으로써 독립된 네트워크 환경을 갖게 됨. 네트워크 인터페이스, IP주소, 라우팅테이블, ARP 테이블, 방화벽 규칙과 같은 요소들이 격리에 포함된다. 가상화된 네트워크 환경은 독립성, 보안, 유연성 측면에서 다양한 용도로 활용될 수 있다.

```
    네트워크 인터페이스 목록을 조회 (-br: brief 간략하게 보기)
$ ip -br a
lo               UNKNOWN        127.0.0.1/8 ::1/128
enX0             UP             172.31.3.131/20 metric 100 fe80::47:7aff:fec2:e0cd/64
$ sudo unshare --net /bin/bash     네트워크 네임스페이스 실행
# ip -br a
lo               DOWN
```

- 가상 이더넷 인터페이스(veth)[14]를 이용하여 네트워크 네임스페이스 간에 연결시키기

```
  veth 쌍을 생성[15]
$ sudo ip link add veth0 type veth peer name veth1
  생성된 디바이스를 확인
$ ip link
[...]
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether e2:4c:bd:0a:bd:18 brd ff:ff:ff:ff:ff:ff
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether f6:ce:9e:0e:70:72 brd ff:ff:ff:ff:ff:ff

  네트워크 네임스페이스를 생성. namespace 이름은 jail 이라고 함.
$ sudo ip netns add  jail
$ ip netns list          또는    ls -l /var/run/netns
jail

  veth0은 호스트에, veth1은 네임스페이스에 할당한다.
$ sudo ip link set veth1 netns jail

  각 veth 인터페이스에 ip 주소를 할당
$ sudo ip addr add 10.0.123.1/24 dev veth0
$ sudo ip netns exec jail  ip addr add 10.0.123.2/24 dev veth1
  호스트에서 veth0의 IP 주소를 확인
$ ip addr show dev veth0
5: veth0@if4: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether f6:ce:9e:0e:70:72 brd ff:ff:ff:ff:ff:ff link-netns jail
```

14) https://man7.org/linux/man-pages/man4/veth.4.html

```
    inet 10.0.123.1/24 scope global veth0
       valid_lft forever preferred_lft forever
  인터페이스를 활성화 하기
$ sudo ip link set veth0 up
$ ip addr show dev veth0
5: veth0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default q
len 1000
    link/ether f6:ce:9e:0e:70:72 brd ff:ff:ff:ff:ff:ff link-netns jail
    inet 10.0.123.1/24 scope global veth0
[...]

 네임스페이스 내부에서 쉘 실행
$ sudo ip netns exec jail  /bin/bash

 라우팅 설정
# ip route add default via 10.0.123.1
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.123.0      0.0.0.0         255.255.255.0   U     0      0        0 veth1
 네임스페이스 내부에서   인터페이스를 활성화 하기
# ip link set veth1 up
# ip addr show dev veth1
4: veth1@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default q
len 1000
    link/ether e2:4c:bd:0a:bd:18 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.123.2/24 scope global veth1
       valid_lft forever preferred_lft forever
[...]
# ping -c 1 10.0.123.1         jail 내부에서 호스트로 ping 테스트
PING 10.0.123.1 (10.0.123.1) 56(84) bytes of data.
64 bytes from 10.0.123.1: icmp_seq=1 ttl=64 time=0.025 ms

--- 10.0.123.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.025/0.025/0.025/0.000 ms

  veth 인터페이스를 삭제함. veth0와 그 peer인 veth1가 모두 삭제된다.
# sudo ip link delete veth0
```

4.4. 마운트 네임스페이스

- 마운트 네임스페이스

```
$ sudo unshare -m  /bin/bash    새로운 마운트 네임스페이스 생성
# readlink /proc/$$/ns/mnt    네임스페이스를 확인
mnt:[4026532294]
# mkdir /tmp/mount_test        tmpfs를 마운트할 테스트 디렉터리를 생성
# mount -t tmpfs tmpfs /tmp/mount_test     tmpfs 를 마운트 하기
# df -h                   마운트 확인
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        77G  4.8G   72G   7% /
```

15) https://man7.org/linux/man-pages/man4/veth.4.html

```
[...]
/dev/xvda16     881M  133M  687M  17% /boot
/dev/xvda15     105M  6.1M   99M   6% /boot/efi
tmpfs           3.9G     0  3.9G   0% /tmp/mount_test <------ 방금 마운트한것
    다른 터미널에서 확인. 네임스페이스 안에서 생성된 tmpfs 는 보이지 않는다.
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        77G  4.8G   72G   7% /
[...]
/dev/xvda16     881M  133M  687M  17% /boot
/dev/xvda15     105M  6.1M   99M   6% /boot/efi
tmpfs           794M   16K  794M   1% /run/user/1000

  마운트를 해제하고 네임스페이스를 종료
# umount /tmp/mount_test
# rmdir /tmp/mount_test
# exit
```

4.5. UTS 네임스페이스

- UTS(UNIX Time-Sharing System) 네임스페이스는 호스트 이름, 도메인 이름, 시스템 식별자를 격리함. 실제 기능과는 달리 UTS 라는 이름은 역사적인 이유로 유지되고 있으며, 이런 불일치는 리눅스 커널 기술의 진화과정에서 발생한 것이다.

```
$ hostname      호스트 네임을 조회하기
ip-172-31-3-131
$ sudo unshare --uts /bin/bash      UTS 네임스페이스에서 쉘을 실행
# hostname new-hostname      호스트네임을 변경
# hostname        변경된 호스트네임을 확인
new-hostname
# exit          네임스페이스를 종료
   네임스페이스를 영속적(persistent)으로 만들기 (=file을 지정한다)
$ touch /tmp/uts_ns
   persistent namespace를 만들고, hostname 을 변경함
$ sudo unshare --uts=/tmp/uts_ns hostname test-host
  nsenter 는 기존 네임스페이스에 진입하여 명령을 실행함
$ sudo nsenter --uts=/tmp/uts_ns  /bin/sh
# hostname    네임스페이스 안에서 호스트명을 확인
test-host
```

※ uname 시스템 호출[16]은 utsname.h 에 정의된 nodename을 가져옴.

```
 /usr/include/x86_64-linux-gnu/sys/utsname.h (편집)
[...]
struct utsname  {
    char sysname[_UTSNAME_SYSNAME_LENGTH]; // operating system eg. "Linux"
    char nodename[_UTSNAME_NODENAME_LENGTH]; // name of this node eg. "ip-172-31-3-1"
    char release[_UTSNAME_RELEASE_LENGTH]; // os release  eg. "6.8.0-1017-aws"
    char version[_UTSNAME_VERSION_LENGTH]; // os version eg. "#18-Ubuntu SMP Wed Oct ..."
    char machine[_UTSNAME_MACHINE_LENGTH];  // hardware identifier eg. "x86_64"
# ifdef __USE_GNU
```

16) https://man7.org/linux/man-pages/man2/uname.2.html

```
    char domainname[_UTSNAME_DOMAIN_LENGTH]; // name of the domain of this node eg. "(none)"
# endif
  };
```

- clone() 시스템 호출[17]을 이용하여 새로운 UTS namespace를 생성하고, sethostname()을 이용하여 호스트네임을 변경하는 프로그램.

```
#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/utsname.h>
#include <sched.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); } while (0)
static int childFunc(void *arg) {
  struct utsname uts;
  printf("child pid = %d\n", getpid());
  // 자식 프로세스에서 호스트명을 변경
  if (sethostname(arg, strlen(arg)) == -1) errExit("sethostname");

  // Retrieve and display hostname
  if (uname(&uts) == -1) errExit("uname");
  printf("uts.nodename in child:  %s\n", uts.nodename);
  sleep(200);  // 자식프로세스를 유지함
  return 0;
}

#define STACK_SIZE (1024 * 1024)    // Stack size for cloned child
int main(int argc, char *argv[]) {
  char *stack;
  char *stackTop;
  pid_t pid;
  struct utsname uts;
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
    exit(EXIT_SUCCESS);
  }
  // Allocate stack for child
  stack = malloc(STACK_SIZE);
  if (stack == NULL) errExit("malloc");
  stackTop = stack + STACK_SIZE;  // Assume stack grows downward

  // clone 시스템호출 CLONE_NEWUTS 플래그는 새로운 uts namespace를 만든다. argv[1]은 호스트명
  pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv[1]);
  if (pid == -1) errExit("clone");
  printf("clone() returned %ld\n", (long) pid);
  printf("parent pid = %d\n", getpid());
  // Parent falls through to here
  sleep(1);   // clone() 호출직후 새로운 child가 실행되기 까지 약간의 시간이 필요함. 새로 생성
된 태스크가 스케쥴링되는 시간, task_struc 생성을 위한 리소스 할당 시간, 컨텍스트 스위칭에 시간
```

17) https://man7.org/linux/man-pages/man2/clone.2.html

```
이 필요함. 이런식으로 대기 시간을 지정하는 것은 바람직하지 않으며 세마포어나 뮤텍스같은 동기화
메커니즘을 사용하거나, 시그널, IPC를 이용하는 것이 좋다.

  if (uname(&uts) == -1) errExit("uname");
  printf("uts.nodename in parent: %s\n", uts.nodename);
  if (waitpid(pid, NULL, 0) == -1) errExit("waitpid"); // child 종료까지 대기
  printf("child has terminated\n");

  exit(EXIT_SUCCESS);
}
$ gcc -o clone1 clone1.c
$ sudo ./clone1 new-hostname      <-- root 권한으로 실행
clone() returned 1430
parent pid = 1429
child pid = 1430
uts.nodename in child:  new-hostname
uts.nodename in parent: ip-172-31-3-131
```

4.6. User 네임스페이스

- User 네임스페이스는 다른 네임스페이스와 달리 일반 사용자도 사용할 수 있음(Linux 3.8이후).

```
  user네임스페이스 지원을 확인
$ grep CONFIG_USER_NS /boot/config-$(uname -r)
CONFIG_USER_NS=y
   user 네임스페이스 활성화. user.max_user_namespaces 는 리눅스 커널 파라미터로 사용자 네임스페
이스안에서 최대 사용자의 개수를 제한함
$ sudo sysctl -w user.max_user_namespaces=10000
  새로운 user 네임스페이스 생성
$ unshare -U /bin/bash      sudo 가 아닌 일반 사용자권한으로 실행
$ id             네임스페이스 안에서 id 명령으로 정보를 확인
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

- /proc/[pid]/uid_map 파일은 user 네임스페이스와 관련된다. user 네임스페이스 내부의 UID를 외부 네임스페이스의 UID와 매핑하는 역할을 함. 형식은 "ID-inside-ns ID-outside-ns length" 임. ID는 네임스페이스 내부/외부의 UID이고, length는 매핑되는 UID의 개수임. /proc/[pid]/gid_map 도 그룹ID에 대하여 동일하다.

```
$ cat /proc/self/uid_map
         0          0 4294967295
```

- user_namespace 구조체는 include/linux/user_namespace.h [18]에 정의됨

```
#define UID_GID_MAP_MAX_BASE_EXTENTS 5        최대 개수는 5개
struct uid_gid_extent {
    u32 first;         네임스페이스 내부의 ID
    u32 lower_first;   네임스페이스 외부(parent namespace)의 ID
    u32 count;         ID의 개수
};
struct uid_gid_map { /* 64 bytes -- 1 cache line */
        u32 nr_extents;
        union {
```

18) https://github.com/torvalds/linux/blob/master/include/linux/user_namespace.h

```
                struct uid_gid_extent extent[UID_GID_MAP_MAX_BASE_EXTENTS];
                struct {
                        struct uid_gid_extent *forward;
                        struct uid_gid_extent *reverse;
                };
        };
};
struct user_namespace {
        struct uid_gid_map      uid_map;      사용자 ID 매핑 정보
        struct uid_gid_map      gid_map;      그룹 ID 매핑 정보
        struct uid_gid_map      projid_map;
        struct user_namespace   *parent;      부모 user namespace에 대한 포인터
        int                     level;        네임스페이스의 중첩 레벨(최대 32)
        kuid_t                  owner;        네임스페이스 소유자와 그룹
        kgid_t                  group;
        struct ns_common        ns;           공통 네임스페이스 구조체
        unsigned long           flags;        관련 플래그
        bool                    parent_could_setfcap;
[...]
};
```

- demo_userns.c [19] 를 이용하여 uid_map의 작용에 대해 알아본다.

```
#define _GNU_SOURCE
#include <sys/capability.h>
#include <sys/wait.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); } while (0)
static int  childFunc(void *arg) {
  cap_t caps;
  for (;;) {   // child 에서 geteuid(), getegid() 의 결과를 표시
    printf("eUID = %ld;  eGID = %ld;  ", (long) geteuid(), (long) getegid());
    caps = cap_get_proc();
    printf("capabilities: %s\n", cap_to_text(caps, NULL));
    if (arg == NULL) break;  // arg 가 null이 아니면 무한 반복
    sleep(5);
  }
  return 0;
}
#define STACK_SIZE (1024 * 1024)
static char child_stack[STACK_SIZE];    // Space for child's stack
int main(int argc, char *argv[]) {
  pid_t pid;
  // CLONE_NEWUSER 플래그를 이용하여 user namespace를 생성하고 child를 실행
  pid = clone(childFunc, child_stack + STACK_SIZE, CLONE_NEWUSER | SIGCHLD, argv[1]);
  if (pid == -1) errExit("clone");
  if (waitpid(pid, NULL, 0) == -1) errExit("waitpid");
  exit(EXIT_SUCCESS);
}
$ sudo apt-get install libcap-dev
```

19) https://lwn.net/Articles/539941/

```
$ gcc -o demo_userns  demo_userns.c  -lcap       // cap 라이브러리가 필요함
$ ./demo_userns
eUID = 65534;  eGID = 65534;  capabilities: =
$ ./demo_userns x      child 프로세서는 user namespace 안에서 실행됨. 65534 는 nobody에 해당
child pid = 2310
eUID = 65534;  eGID = 65534;  capabilities: =
eUID = 65534;  eGID = 65534;  capabilities: =
[...]
  다른 터미널에서 위의 demo_userns 가 실행중인 자식 프로세스에 대한 사용자 id 맵핑을 만들어준다
*$ echo '0 1000 1' > /proc/2310/uid_map    <--- 2310 은 위 child 의 pid

[...]
eUID = 65534;  eGID = 65534;  capabilities: =
eUID = 0;  eGID = 65534;  capabilities: =   <------- uid_map 이 변경됨
eUID = 0;  eGID = 65534;  capabilities: =
^C
  uid_map 변경은 네임스페이스당 단 한번만 쓰기가 가능하다.
  여러 줄의 매핑을 정의할 수 있지만 이는 단일 쓰기 작업으로 이루어 져야 함.
```

4.7. IPC 네임스페이스

- IPC 네임스페이스 테스트

```
  ipcmk 명령은 다양한 IPC 자원을 생성한다. 공유메모리, 메시지큐, 세마포어 등이 있다.
$ ipcmk -M 100        호스트에서 100 B의 공유 메모리를 생성
Shared memory id: 1
  ipcs 명령은 IPC 현황을 조회한다. (-m 옵션은 shmem만 )
$ ipcs  -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x4c9d210d 1          ubuntu     644        100        0

  ipc 네임스페이스에서 쉘을 실행
$ sudo unshare -i /bin/bash

# ipcmk -M 200     네임스페이스 안에서 200 B의 공유 메모리를 생성
Shared memory id: 0
# ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0xc732d687 0          root       644        200        0
# ipcrm -m 0   공유메모리를 삭제
```

4.8. 제어그룹 네임스페이스

- cgroup 네임스페이스 테스트

```
 테스트를 위한 cgroup을 생성
$ sudo mkdir /sys/fs/cgroup/test
$ echo 50000 ¦ sudo tee /sys/fs/cgroup/test/cpu.max
$ cat /sys/fs/cgroup/test/cpu.max
50000 100000

  새로운 cgroup 네임스페이스 생성
```

```
$ sudo unshare -C /bin/bash

  네임스페이스 안에서 cgroup 계층 구조 확인
# systemd-cgls
CGroup /../../..:
-.slice
# cat /proc/self/cgroup    # 현재 cgroup 경로를 확인
0::/
# echo $$           쉘의 pid 를 확인
1107

  호스트에서 위 프로세스를 cgroup에 할당
$ echo 1107¦ sudo tee /sys/fs/cgroup/test/cgroup.procs

  네임스페이스 안에서 cgroup 경로를 다시 확인하면 변경된 것을 확인
# cat /proc/self/cgroup
0::/../../../test

# ./busy_single      cpu 부하를 발생하는 프로그래램을 실행
pid: 1090
nice:0 CPU Usage: 51.03%
nice:0 CPU Usage: 50.67%
^C
   호스트에서 리소스 사용량을 확인
$ cat /sys/fs/cgroup/test/cpu.stat
usage_usec 8892739
user_usec 8882719
system_usec 10020
[...]
```

4.9. 타임 네임스페이스

- 타임 네임스페이스 테스트

```
  새로운 네임스페이스를 생성
$ sudo unshare -T /bin/bash
# date         시스템 시간을 확인
Wed Oct 23 02:27:04 UTC 2024
# sudo date -s '1984-01-01 00:00:00'    시스템 시간을 변경
Sun Jan  1 00:00:00 UTC 1984
# date
Sun Jan  1 00:00:10 UTC 1984
# echo $$      현재 쉘의 pid를 확인
1174
$ sudo nsenter -t 1174 -T /bin/bash   다른 터미널에서 위의 time 네임스페이스로 진입
# date                        시스템 시간은 네임스페이스에서 유지됨
Sun Jan  1 00:01:10 UTC 1984
```

- time_namespace.h[20] 내부 관련 구조체 (커널 5.6 이후)

```
struct timens_offsets {
        struct timespec64 monotonic;
```

20) https://github.com/torvalds/linux/blob/master/include/linux/time_namespace.h#L22

```
        struct timespec64 boottime;
};

struct time_namespace {
        struct user_namespace   *user_ns;
        struct ucounts          *ucounts;
        struct ns_common        ns;
        struct timens_offsets   offsets;
        struct page             *vvar_page;
        /* If set prevents changing offsets after any task joined namespace. */
        bool                    frozen_offsets;
} __randomize_layout;
```

- setns 시스템 호출[21] 예제

```
// setns1.c
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define errExit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)

int main(int argc, char *argv[]) {
  int fd;
  if (argc < 3) {
    fprintf(stderr, "Usage: %s /proc/PID/ns/FILE cmd args...\n", argv[0]);
    exit(EXIT_FAILURE);
  }
  // 네임스페이스 파일을 열기
  fd = open(argv[1], O_RDONLY | O_CLOEXEC);  //O_CLOEXEC 플래그는 자식프로세스에 파일 디스크립
터가 상속되지 않는다. close-on-exec 로 'exec'계열의 함수를 호출할때 자동으로 닫히게 한다.
  if (fd == -1) errExit("open");
  // 지정된 네임스페이스에 참여 (setns(fd))
  if (setns(fd, 0) == -1) errExit("setns");
  execvp(argv[2], &argv[2]);  // 프로그램을 실행
  errExit("execvp");
}
$ gcc -o setns1       setns1.c
 위에서 만들어진 time namespace (pid=1174) 에 참여하여 date 명령을 실행
$ sudo ./setns1  /proc/1174/ns/time  date
Sun Jan  1 00:03:20 UTC 1984
```

21) https://man7.org/linux/man-pages/man2/setns.2.html

5. 리눅스 세부권한(Linux Capability)

- 리눅스 capability(기능)는 루트 권한을 더 세분화된 권한으로 나누어 프로세스에 필요한 최소한의 권한만 부여할 수 있게 해주는 커널의 기능이다. 전통적인 root/non-root 이분법적인 권한 구조를 개선하고, 프로세스에 필요한 최소한의 권한만 부여하여 보안을 강화하는 것이 목적. 약 40개의 개별 기능이 존재.
- 최소권한의 원칙(PoLP: principle of least privilege, PoMP: minimal privilege)[22]은 사용자 계정이나, 프로세스에 의도한 기능을 수행하는데 필수적인 권한만 부여하는 것을 말함.
- 관련 도구
  * `getcap, setcap:` capability 조회 및 실행
  * `capsh:` capability 테스트 및 관리
  * `libcap-ng-utils:` pscap 등 추가 도구 제공

※ Linux Capability와 SELinux(Secutiry-Enhanced Linux)는 모두 리눅스 시스템의 보안을 강화하기 위한 메커니즘이지만 차이가 존재함: Linux Capability는 루트권한을 세분화된 권한들로 나누어 프로세스 수준의 권한 제어에 목적이 있으나, SELinux는 강제적 접근 제어(MAC, mandatory access control)를 구현하여 전체 시스템에 대한 보안 정책을 적용함. 파일, 프로세스, 포트 등 시스템에 걸치 보안 정책을 적용

- 세부 기능의 종류는 `capability.h` 파일에 정의되어 있다

```
$ grep "define CAP_" /usr/include/linux/capability.h
#define CAP_CHOWN            0
#define CAP_DAC_OVERRIDE     1
#define CAP_DAC_READ_SEARCH  2
#define CAP_FOWNER           3
[...]
```

- 세부 기능의 전체 목록[23][24]

세부 기능	설명
CAP_CHOWN	파일의 소유자와 그룹을 변경할 수 있는 권한
CAP_DAC_OVERRIDE	파일 접근 권한 검사를 우회할 수 있는 권한
CAP_DAC_READ_SEARCH	파일과 디렉토리의 읽기/실행 권한 검사를 우회할 수 있는 권한
CAP_FOWNER	파일 소유자 검사를 우회할 수 있는 권한
CAP_FSETID	파일의 setuid와 setgid 비트를 설정할 수 있는 권한
CAP_KILL	모든 프로세스에 시그널을 보낼 수 있는 권한
CAP_SETGID	프로세스의 그룹 ID를 변경할 수 있는 권한
CAP_SETUID	프로세스의 사용자 ID를 변경할 수 있는 권한
CAP_SETPCAP	프로세스의 capability를 변경할 수 있는 권한
CAP_LINUX_IMMUTABLE	파일의 불변 속성을 변경할 수 있는 권한
CAP_NET_BIND_SERVICE	1024 미만의 포트에 바인딩할 수 있는 권한
CAP_NET_BROADCAST	브로드캐스트와 멀티캐스트를 할 수 있는 권한
CAP_NET_ADMIN	네트워크 인터페이스 설정을 변경할 수 있는 권한
CAP_NET_RAW	RAW 소켓을 사용할 수 있는 권한
CAP_IPC_LOCK	공유 메모리 세그먼트를 잠글 수 있는 권한
CAP_IPC_OWNER	IPC 소유권 검사를 우회할 수 있는 권한
CAP_SYS_MODULE	커널 모듈을 로드하거나 언로드할 수 있는 권한
CAP_SYS_RAWIO	직접적인 I/O 포트 접근을 허용하는 권한
CAP_SYS_CHROOT	chroot() 시스템 콜을 사용할 수 있는 권한
CAP_SYS_PTRACE	ptrace() 시스템 콜과 관련 기능을 사용할 수 있는 권한

22) https://en.wikipedia.org/wiki/Capability-based_security
23) https://man7.org/linux/man-pages/man7/capabilities.7.html
24) https://linux-audit.com/kernel/capabilities/overview/

CAP_SYS_PACCT	프로세스 계정 관리를 수행할 수 있는 권한
CAP_SYS_ADMIN	다양한 시스템 관리 작업을 수행할 수 있는 권한
CAP_SYS_BOOT	reboot 시스템 콜을 사용할 수 있는 권한
CAP_SYS_NICE	프로세스의 우선순위를 변경할 수 있는 권한
CAP_SYS_RESOURCE	리소스 제한을 무시할 수 있는 권한
CAP_SYS_TIME	시스템 시간을 설정할 수 있는 권한
CAP_SYS_TTY_CONFIG	tty 장치를 구성할 수 있는 권한
CAP_MKNOD	특수 파일을 생성할 수 있는 권한
CAP_LEASE	파일에 대한 리스를 설정할 수 있는 권한
CAP_AUDIT_WRITE	커널 감사(audit) 로그에 기록을 쓸 수 있는 권한
CAP_AUDIT_CONTROL	Linux 커널의 감사(auditing) 시스템을 제어할 수 있는 권한
CAP_SETFCAP	프로세스가 파일에 임의의 capability를 설정할 수 있게 합
CAP_MAC_OVERRIDE	SELinux나 AppArmor와 같은 보안 모듈에 의해 강제되는 강제적 접근 제어(MAC) 규칙을 무시할 수 있게 함
CAP_MAC_ADMIN	MAC 정책을 관리할 수 있음
CAP_SYSLOG	특권이 필요한 syslog 작업을 수행할 수 있게 함
CAP_WAKE_ALARM	프로그래밍 가능한 타이머와 알람을 통해 시스템 웨이크업 이벤트를 트리거할 수 있게 함
CAP_BLOCK_SUSPEND	시스템 일시 중지를 차단할 수 있는 권한
CAP_AUDIT_READ	감사 로그를 읽을 수 있는 권한
CAP_PERFMON	성능 모니터링 관련 활동을 수행할 수 있는 권한
CAP_BPF	BPF(Berkeley Packet Filter) 프로그램을 로드하고 BPF 맵을 생성할 수 있는 권한
CAP_CHECKPOINT_RESTORE	프로세스의 체크포인트와 복원 작업을 수행할 수 있는 권한

- capsh 명령을 사용하면 활성 커널에서 사용가능한 세부기능의 전체 목록을 볼수 있다

```
$ sudo capsh --print
Current: =ep    "="기호는 해당 기능이 프로세스에 부여되어 있음을 의미. "e"는 effective의 약자로 현재 활성화 되어 있음을 의미. "p"는 permitted로 해당 기능이 허용되어 있음을 의미
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read,cap_perfmon,cap_bpf,cap_checkpoint_restore
Ambient set =
Current IAB:
Securebits: 00/0x0/1'b0 (no-new-privs=0)
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=0(root)
Guessed mode: HYBRID (4)
```

※ 프로세스 IAB(Inheritable, Ambient, Bounding)란 프로세스의 capability를 관리하는 고급 매커니즘. IAB tuple은 세가지 요소로 구성됨

* Inheritable(I) 세트: 자식 프로세스로 상속될 수 있는 capabilities
* Ambient(A) 세트: 실행 파일에 특별한 capabilities 설정이 없을 때 사용되는 capabilities
* Bounding(B) 세트: 프로세스가 획득할 수 있는 capabilities의 상한선

- 특정 프로세스의 세부기능을 보려면 /proc 디렉터리에서 status 파일을 이용한다.

```
[root]# cat /proc/self/status ¦ grep Cap
CapInh: 0000000000000000     상속된 기능
CapPrm: 000001fffffffffff     허용된 기능
CapEff: 000001fffffffffff    효과적인(effective) 기능
CapBnd: 000001fffffffffff    바운딩 세트
CapAmb: 0000000000000000    주변 기능 세트

  capsh 명령을 이용하여 위 값을 이름으로 디코딩 할 수 있음
# capsh --decode=000001fffffffffff
0x000001fffffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kil
l,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,
cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroo
t,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys
_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_
mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read,cap_perfm
on,cap_bpf,cap_checkpoint_restore
```

- ping 에 대한 CAP_NET_RAW 기능을 삭제하면 ping 유틸리티가 더 이상 작동하지 않는다

```
$ /bin/ping -c 1 localhost           로컬 시스템에 single ping 테스트
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.014 ms

--- localhost ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.014/0.014/0.014/0.000 ms

$ capsh --drop=cap_net_raw -- -c "/bin/ping -c 1 localhost"
unable to raise CAP_SETPCAP for BSET changes: Operation not permitted
```

- getcap 명령은 특정파일의 cap 정보를 조회한다.

```
   ping 유틸리티의 capability 정보를 조회하기
$ getcap /bin/ping
/bin/ping cap_net_raw=ep    이 속성이 들어가 있기 때문에 non-root 권한 사용자가 실행가능

  capability 정보는 파일 시스템의 확장 속성(extended attributes)으로, inode에 저장된다. 대부분
의 현대 리눅스 파일시스템(ext4, xfs 등)은 inode 외부에 별도의 확장 속성 영역을 가지고 있다
  getfattr 명령은 이 정보를 조회한다.
$ sudo apt install attr
$ getfattr -n security.capability  /bin/ping
getfattr: Removing leading '/' from absolute path names
# file: bin/ping
security.capability=0sAQAAAgAgAAAAAAAAAAAAAAAAAA=
```

- setcap 명령[25]은 특정 파일의 cap를 설정한다.

```
   ping 유틸리티의 capability 속성을 삭제하기
$ sudo setcap  -r /bin/ping
$ getcap /bin/ping
  속성이 삭제됨. 이제 root 권한이 아니면 ping을 사용할 수 없음
   속성을 추가하기
```

25) https://man7.org/linux/man-pages/man8/setcap.8.html

```
$ sudo setcap  'cap_net_raw=+ep'   /bin/ping
$ getcap /bin/ping
/bin/ping cap_net_raw=ep
```

- 앞에서 소개한 my_server.py 에서 서비스 포트를 1024보다 작은 포트에서 서비스 하려면 python 바이너리에 cap_net_bind_service 권한을 설정한다.

```
$ /usr/bin/python3  ./my_server.py  90
port :  90
Traceback (most recent call last):
  File "/home/ubuntu/linux-drill/systemd/my_server.py", line 24, in <module>
    webServer = HTTPServer((hostName, port), MyServer)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
PermissionError: [Errno 13] Permission denied
   1024 이하의 포트에서 non-root 권한으로 실행이 안됨

  파이썬 인터프리터에 setcap 으로 cap_net_bind_service 권한을 부여한다. 심볼릭 링크(/usr/bin/python3 -> python3.12)에는 속성부여가 되지 않으므로, 실제 파일에 속성을 설정함
$ sudo setcap  "cap_net_bind_service=+ep"   /usr/bin/python3.12
$ getcap /usr/bin/python3.12
/usr/bin/python3.12 cap_net_bind_service=ep

   non-root 사용자도 1024 이하의 포트를 사용가능
$ python3 my_server.py  90
port :  90
Server started http://localhost:90
```

- getpcaps 유틸리티[26]는 실행중인 프로세스의 capability를 조회한다

```
$ getpcaps 1       init 프로세스의 cap조회
1: =ep
  nginx 프로세스의 cap 조회
$ getpcaps $(pgrep nginx)
613: =ep
614: =
615: =
  위에서 실행중인 my_server.py 의 cap 조회
$ getpcaps $(pgrep python3)
2715: cap_net_bind_service=ep
```

26) https://man7.org/linux/man-pages/man8/getpcaps.8.html

6. LXC 컨테이너

- LXC(Linux Containers)[27][28]는 리눅스 시스템 운영체제 수준의 가상화 기술임. 단일 호스트에서 여러 개의 독립된 리눅스 시스템(컨테이너)를 실행하기 위한 가상화 방법. 2006년 Google이 Process Containers(이후 cgroup으로 이름 변경) 기능을 발표함. 2008년 RedHat에서 시스템 자원을 논리적으로 분할하는 Namespace기술을 발표. 같은 해 2008년 IBM에서 LXC를 발표. LXC는 cgroup과 namespace를 활용하여 구현된 최초의 리눅스 컨테이너 엔진이다. 이후 Docker(2013년), Kubernetes(2014년)과 같은 기술 발전으로 이어짐. Docker는 초기에 LXC를 컨테이너 드라이버로 사용했지만, 이후에 자체 드라이버(libContainer)로 전환하였다.
- LXD(Linux Container Daemon)은 LXC를 기반으로 한 컨테이너 관리 시스템임. REST API를 통해 원격에서 컨테이너를 관리할 수 있는 기능을 제공.

6.1. LXC 컨테이너 실행

- LXC 설치하고 실행하기

```
$ sudo apt-get install lxc
$ dpkg-query  -L lxc
$ lxc-create --version   버전확인
5.0.3

  컨테이너를 실행하기. templete는 ubuntu를 사용
$ sudo lxc-create -n mycontainer -t ubuntu

  컨테이너를 실행하기  (-d는 daemon으로 실행)
$ sudo lxc-start -n mycontainer -d

  상태확인
$ sudo lxc-ls --fancy
NAME        STATE   AUTOSTART GROUPS IPV4 IPV6 UNPRIVILEGED
mycontainer RUNNING 0         -      -    -    false
$ sudo lxc-info --name mycontainer
Name:           mycontainer
State:          RUNNING
PID:            29797
IP:             10.0.3.10
Link:           vethdjuMyf
 TX bytes:      14.25 KiB
 RX bytes:      14.69 KiB
 Total bytes:   28.94 KiB

  컨테이너에 접속하기
$ sudo lxc-attach -n mycontainer
root@mycontainer # sleep 1000
```

※ 이미지 저장위치는 `/var/lib/lxc/mycontainer` 이며 이미지 템플릿은 `/usr/share/lxc/templates` 이다

※ 컨테이너 중지하기

```
$ sudo lxc-stop -n mycontainer
$ sudo lxc-ls --fancy
```

27) https://linuxcontainers.org/
28) https://github.com/lxc/lxc

```
NAME        STATE   AUTOSTART GROUPS IPV4 IPV6 UNPRIVILEGED
mycontainer STOPPED 0         -      -    -    false
$ sudo lxc-destroy -n mycontainer     컨테이너 삭제
```

- 컨테이너가 실행중인 상태에서 호스트의 프로세스 구조

```
$ pstree -a       컨테이너의 프로세스 확인
[...]
  ├─lxc-start
  |   └─systemd      컨테이너 내부의 systemd 프로세스를 확인
  |       ├─agetty -o -p -- \\u --noclear --keep-baud - 115200,38400,9600 vt220
  |       ├─cron -f -P
  |       ├─dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
  |       ├─rsyslogd -n -iNONE
  |       |   └─3*[{rsyslogd}]
  |       ├─systemd-journal
  |       ├─systemd-logind
  |       ├─systemd-network
  |       └─systemd-resolve
[...]
  ├─sshd
  |   ├─sshd
  |   |   └─sshd
  |   |       └─bash
  |   |           └─sudo lxc-attach -n mycontainer
  |   |               └─sudo lxc-attach -n mycontainer
  |   |                   └─3 -n mycontainer
  |   |                       └─bash   <---- 컨테이너 내부의 쉘
  |   |                           └─sleep 1000
```

- cgroup 계층구조

```
$ systemd-cgls
CGroup /:
-.slice     <----- root 슬라이스
├─lxc.monitor.mycontainer
| └─25782 [lxc monitor] /var/lib/lxc mycontainer
[...]
├─lxc.payload.mycontainer
| ├─.lxc
| | ├─25993 /bin/bash    <------- 컨테이너 쉘
| | └─26199 sleep 1000
| ├─init.scope
| | └─25787 /sbin/init
| └─system.slice       <--- 컨테이너 내부의 system.slice
|   ├─systemd-networkd.service
|   | └─25878 /usr/lib/systemd/systemd-networkd
|   ├─cron.service
|   | └─25897 /usr/sbin/cron -f -P
|   ├─systemd-journald.service
|   | └─25842 /usr/lib/systemd/systemd-journald
|   ├─rsyslog.service
|   | └─25921 /usr/sbin/rsyslogd -n -iNONE
|   ├─console-getty.service
|   | └─25911 /sbin/agetty -o -p -- \u --noclear --keep-baud - 115200,38400,9600 vt220
|   ├─systemd-resolved.service
|   | └─25892 /usr/lib/systemd/systemd-resolved
|   ├─dbus.service
|   | └─25898 @dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog
-only
|   └─systemd-logind.service
|     └─25901 /usr/lib/systemd/systemd-logind
```

```
[...]
```

- PID 네임스페이스

```
  컨테이너 내부에서  init 프로세스는 PID 1을 확인할 수 있다. 모든 프로세스는 init 프로세스의 자식 프로세스임.
root@mycontainer# ps ax -o pid,ppid,command
    PID    PPID COMMAND
      1       0 /sbin/init
     20       1 /usr/lib/systemd/systemd-journald
     56       1 /usr/lib/systemd/systemd-networkd
     68       1 /usr/lib/systemd/systemd-resolved
[...]
  호스트에서 ps 결과는 컨테이너의 init 프로세스는 pid 1이 아님
$ ps ax ¦ grep init
      1 ?        Ss     0:03 /sbin/init  <------ 호스트의 init process
  25787 ?        Ss     0:00 /sbin/init  <------ 컨테이너의 init process
  lsns 명령으로 확인
$ sudo lsns -t pid
        NS TYPE NPROCS   PID USER COMMAND
4026531836 pid     131     1 root /sbin/init
4026532293 pid      10 25787 root /sbin/init
```

- 컨테이너의 CPU 자원을 제한하기

```
$ sudo lxc-cgroup -n mycontainer cpu.max            cpu 제한량 확인
max
$ sudo lxc-cgroup -n mycontainer cpu.max "300000 1000000"     cpu 제한량 설정
$ sudo lxc-cgroup -n mycontainer cpu.max
300000 1000000
$ cat  /sys/fs/cgroup/lxc.payload.mycontainer/cpu.max       cgroup 파일시스템에서 확인
300000 1000000
  파일을 컨테이너 안으로 복사하기
$ cp busy_single   /var/lib/lxc/mycontainer/rootfs/home/ubuntu/
root@mycontainer# ./busy_single      컨테이너에서 파일을 실행하면 cpu가 30%로 제한됨
pid: 118
nice:0 CPU Usage: 39.06%
nice:0 CPU Usage: 26.32%
nice:0 CPU Usage: 34.88%
^C
```

- 컨테이너의 메모리 자원을 제한하기

```
$ sudo vi  /var/lib/lxc/mycontainer/config          설정파일을 수정하기
[...]
lxc.cgroup2.memory.max = 1073741824             1GB로 제한하기

  컨테이너를 실행하고 cgroup 메모리 제한 설정을 확인
$ sudo lxc-start -n mycontainer -d
$ cat  /sys/fs/cgroup/lxc.payload.mycontainer/memory.max
1073741824

  컨테이너 내부에서 총 메모리는 호스트의 전체 메모리 용량으로 보여짐
root@mycontainer# free -h
               total        used        free      shared  buff/cache   available
```

```
Mem:           7.8Gi       659Mi       3.6Gi       2.3Mi       3.8Gi       7.1Gi
Swap:             0B          0B          0B
  설정된 용량을 초과하면 프로세스가 종료(Kill)됨
root@mycontainer# ./malloc2  1100          1.1GB 메모리 할당 시도
1MB * 1100 메모리가 성공적으로 할당되었습니다.
write Killed... 977
```

- 네트워크 조회

```
$ lxc network list
+--------+----------+---------+----------------+---------------------------+-------------+---------+---------+
|  NAME  |   TYPE   | MANAGED |      IPV4      |           IPV6            | DESCRIPTION | USED BY |  STATE  |
+--------+----------+---------+----------------+---------------------------+-------------+---------+---------+
| enX0   | physical | NO      |                |                           |             | 0       |         |
+--------+----------+---------+----------------+---------------------------+-------------+---------+---------+
| lxcbr0 | bridge   | NO      |                |                           |             | 0       |         |
+--------+----------+---------+----------------+---------------------------+-------------+---------+---------+
| lxdbr0 | bridge   | YES     | 10.198.117.1/24 | fd42:b49d:512a:285f::1/64 |             | 1       | CREATED |
+--------+----------+---------+----------------+---------------------------+-------------+---------+---------+
```

6.2. chroot와 컨테이너

- Containers from Scratch [29]의 내용을 정리한다.
- chroot [30]로 파일시스템 격리하기

```
  앞에서 만든 LXC 컨테이터의 rootfs 를 이용한다.
# cd /var/lib/lxc/mycontainer
# sudo chroot  rootfs  /bin/bash         chroot로 rootfs 폴더를 / 로 변경하기
# ls
bin               boot  etc   lib               lib64  mnt  proc  run   sbin.usr-is-merged  sys  usr
bin.usr-is-merged  dev   home  lib.usr-is-merged  media  opt  root  sbin  srv                 tmp  var
# which python3
/usr/bin/python3
# /usr/bin/python3 -c 'print("Hello, container world!")'
Hello, container world!
# exit                      chroot 를 빠져나오기
    앞서 systemd 에서 테스트한 파이썬 웹서버 프로그램을 rootfs로 복사
$ cp /opt/my_server/my_server.py  rootfs/
# sudo chroot rootfs python3  /my_server.py      chroot 로 파이썬을 실행
Server started http://localhost:8080
[...]
     다른터미널에서 curl localhost:8080 으로 http 요청하기
127.0.0.1 - - [24/Oct/2024 08:36:11] "GET / HTTP/1.1" 200 -
 ^C
```

- chroot는 호스트의 특정폴더안으로 프로세스를 격리시켜 주지만, 호스트이 프로세스 정보는 컨테이너 내부에 그대로 노출된다.

```
   chroot 외부에서 쉘을 실행하고, pid를 조회하기
$ sh
[new shell]$ echo $$
1406
  다른 터미널에서 chroot 로 쉘을 실행하여 프로세스 조회. proc 파일시스템 마운트가 필요함
```

29) https://ericchiang.github.io/post/containers-from-scratch/
30) https://man7.org/linux/man-pages/man1/chroot.1.html

```
# sudo chroot rootfs /bin/bash
# mount -t proc proc /proc
# ps aux ¦ grep “ sh"
ubuntu      1406  0.0  0.0   2800  1664 ?        S+   08:41   0:00 sh  <----- chroot 밖에서 실행중인 쉘
# kill -9 1406         <--- chroot 밖에서 실행중인 쉘을 강제종료
```

- pid 네임스페이스를 분리(unshare)함으로써, chroot 안에서 실행중인 프로세스는 밖의 프로세스에 접근할 수 없게 된다.

```
   --mount-proc[=mountpoint] 옵션은 해당 프로그램을 실행하기 직전에 proc 파일시스템을 mountpoint 에 연결한다. =mountpoint 가 생략될 경우 기본값은 /proc 임. chroot 내부에서는 호스트의 /proc을 볼수 없게 되므로, rootfs/proc 에 마운트해 줌.
# sudo unshare --pid  -f --mount-proc=$PWD/rootfs/proc  chroot rootfs /bin/bash
# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0  10604  4224 ?        S    08:49   0:00 /bin/bash
root           8  0.0  0.0  13868  4224 ?        R+   08:49   0:00 ps aux
# echo $$
1
# readlink /proc/self/ns/pid
pid:[4026532290]     <------ chroot 내부에서 실행된 쉘의 pid namespace
  chroot 내부에서 실행되는 쉘을 찾기
# ps aux ¦ grep /bin/bash
[...]
root        1648  0.0  0.0  10608  4224 pts/4    S+   09:08   0:00 /bin/bash
    위 쉘 프로세스의 namespace 를 조회 (pid = 1648)
# readlink /proc/1648/ns/pid
pid:[4026532290]          <-------- chroot 내부에서 조회한 것과 동일함

  위 namespace에 조인하여 새로운 쉘을 실행하려면 nsenter 를 이용한다.
$ sudo nsenter --pid=/proc/1648/ns/pid \
    unshare -f --mount-proc=$PWD/rootfs/proc  chroot /var/lib/lxc/mycontainer/rootfs /bin/bash
# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0  10608  4224 ?        S+   09:08   0:00 /bin/bash
root          13  0.0  0.0  10604  4224 ?        S    09:14   0:00 /bin/bash
root          26  0.0  0.0  13868  4224 ?        R+   09:15   0:00 ps aux
   네임스페이스에 진입한 두번째 쉘에서 첫번째 쉘의 pid 1을 확인할 수 있다
```

- 컨테이너 내부에서 사용할 수 있는 파일이나 설정을 주입하기 위해서 종종 읽기전용의 마운트 폴더를 사용한다.

```
   호스트에서 마운트할 폴더를 생성함
# mkdir readonlyfiles
# echo "hello" > readonlyfiles/hi.txt
   컨테이너 rootfs 내부에 마운트할 폴더를 생성
# sudo mkdir -p rootfs/var/readonlyfiles
   mount --bind 로 읽기전용으로 디렉터리를 바인드한다
# sudo mount --bind -o ro $PWD/readonlyfiles $PWD/rootfs/var/readonlyfiles
   chroot 로 쉐을 실행
# sudo chroot rootfs /bin/bash
   chroot 내부의 쉘에서 바인드된 폴더를 볼수 있지만, 변경은 불가하다.
# cat /var/readonlyfiles/hi.txt
```

```
hello
# echo "bye" > /var/readonlyfiles/hi.txt
bash: /var/readonlyfiles/hi.txt: Read-only file system

  호스트에서 해당 폴더를 unmount 할 수 있음
# sudo umount $PWD/rootfs/var/readonlyfiles
```

- 컨테이너 cgroup을 생성하고 메모리 제한

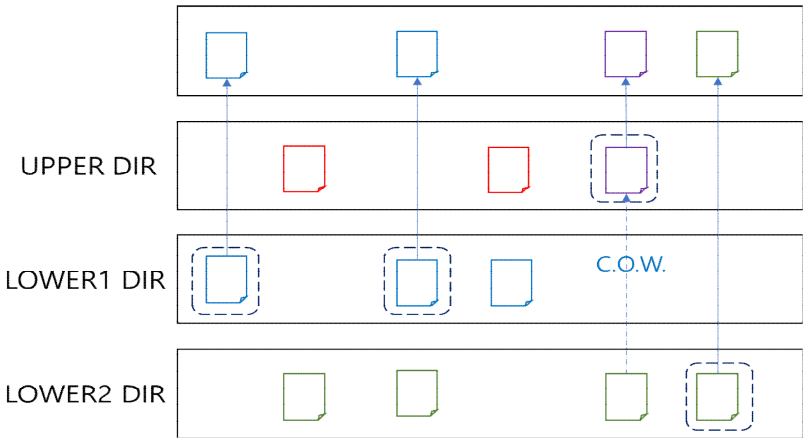
```
$ sudo su
   jail 이라는 cgroup을 생성하고 메모리를 500MB 로 제한. swap은 off 한다.
# mkdir /sys/fs/cgroup/jail
# echo "524288000" > /sys/fs/cgroup/jail/memory.max
# echo "0" > /sys/fs/cgroup/jail/memory.swap.max

   현재 쉘을 jail cgroup에 포함시킨다
# echo $$ > /sys/fs/cgroup/jail/cgroup.procs
# cat /proc/self/cgroup
0::/jail

   chroot 로 쉘을 실행하고, 메모리 할당 프로그램으로 테스트
# chroot /var/lib/lxc/mycontainer/rootfs  /bin/bash
# ./malloc2 400
1MB * 400 메모리가 성공적으로 할당되었습니다.
메모리가 해제되었습니다.
# ./malloc2 500
1MB * 500 메모리가 성공적으로 할당되었습니다.
write Killed... 465   <------ 제한된 500MB 메모리를 초과할 수 없음
```

7. 오버레이 파일시스템(OverlayFS)

- OverlayFS는 컨테이너 환경에서 효율적인 파일 시스템 관리를 위해서 사용된다. 동일한 파일을 여러 컨테이너가 공유할 수 있으므로 데이터 중복 문제가 해결됨. 아래(lower) 레이어의 파일은 읽기전용으로 컨테이너의 이미지에 해당하며, 상위(upper) 레이어는 컨테이너에서 쓰기가 발생할 경우 CoW(copy-on-write)방식으로 기록된다. 아래 레이어 파일을 삭제해도 실제로 삭제되지 않으며, 상위 레이어에 화이트아웃(whiteout) 파일이 생성되어 삭제된 것으로 인식된다. 유니온 파일 시스템(Union File System)의 일종임.



7.1 OverlayFS 테스트

```
  커널 모듈에서 OverlayFS 모듈 확인
$ lsmod ¦ grep overlay
overlay               212992  0
  커널 모듈이 로딩되지 않은 경우 modprobe 를 실행해 준다.
# modprobe overlay

  커널 설정에서 오버레이 파일시스템이 활성화 되어 있는지 확인
$ grep CONFIG_OVERLAY_FS /boot/config-$(uname -r)
CONFIG_OVERLAY_FS=m
[...]

$ cat /proc/filesystems ¦ grep overlay     커널에서 지원되는 파일시스템 확인
nodev   overlay
   nodev 의미는 물리적 디바이스가 없이 사용되는 파일시스템으로 가상파일시스템이거나 특수한 목적으로 사용되는 파일시스템임을 의미함
```

```
  OverlayFS 마운트 방법
# sudo mkdir -p /lower  /upper /work  /merged
# sudo mount -t overlay \
    -olowerdir=/lower,upperdir=/upper,workdir=/work  overlay  /merged
    타입(-t)을 overlay 로 지정, 옵션(-o)으로 lowerdir,upperdir,workdir 의 위치를 지정

$ df -h
Filesystem      Size  Used Avail Use% Mounted on
[...]
overlay          77G  9.0G   68G  12% /merged
```

```
    /etc/fstab 에 넣을경우
# vi /etc/fstab
[...]
overlay /merged  overlay  lowerdir=/lower,upperdir=/upper,workdir=/work 0 0
   lower 에 파일을 생성
# echo alice >> /lower/file1
# tree /lower /upper /work /merged
/lower
└── file1       lower/file1 이 merged/file1 에도 그대로 보여짐
/upper
/work
/merged
└── file1
  merged 폴더에 파일을 생성
# echo bob >> /merged/file2
# tree /lower /upper /work /merged
/lower
└── file1
/upper
└── file2
/work
/merged
├── file1
└── file2     merged/file2 는 실제로는 upper/file2 에 저장된다.

  merged 폴더의 파일을 수정
# echo cat >> /merged/file1
# tree /lower /upper /work /merged
/lower
└── file1    <-- 원본 파일은 그대로 유지됨
/upper
├── file1    <-- 수정된 버전
└── file2
/work
└── work
/merged
├── file1    <-- 실제로는 lower/file1 에 있는 파일 merged/file1을 수정하면
└── file2             CoW 원칙으로 upper/file1 에 복사된 버전이 수정된다
# cat /lower/file1
alice
# cat /upper/file1
alice
cat
   merged 폴더에서 파일을 삭제
# rm  /merged/file1
# tree /lower /upper /work /merged
/lower
└── file1   <-- 원본 파일은 그대로 유지
/upper
├── file1   <-- 삭제된 파일 파입이 'c'인 특수 파일로 변경됨
└── file2
/work           <-- workdir은 OverlayFS의 내부 작업 공간으로 활용됨
└── work
    └── #7
```

```
/merged          <-- merged/file1은 삭제되어 보이지 않음
└── file2
# ls -la /upper/file1
c--------- 2 root root 0, 0 Oct 30 14:29 /upper/file1
```

※ OverlayFS는 리눅스 커널에 통합된 유니온 파일시스템이며, MS Windows에서는 필터드라이브(Filter Driver)[31]라는 기술을 활용하여 이를 구현한다. 필터 드라이버 기술은 파일 시스템 소프트웨어 스택에 연결되어 파일 I/O 작업을 모니터링, 필터링, 수정하는 기능을 제공함. I/O 작업에 대한 가상 레이어를 생성하여 읽기 작업이 들어오면 가상레이어에서 해당 파일을 찾아 반환하고, 가상레이어에 없으면 실제 파일시스템에서 읽어서 반환하는 방식. 실제 파일 시스템은 변경되지 않고, 모든 수정사항은 가상 레이어에 저장됨.

- Overlay2 파일시스템은 커널 4.0 이상에서 사용가능하며(overlay는 커널 3.18이상), 최대 128개의 lower 레이어를 지원함. 성능면에서 overlay2가 overlay보다 우수함. 둘다 ext4 또는 xfs 파일시스템에서 동작한다.

```
  디렉터리 생성
# mkdir -p /tmp/overlay2/{lower1,lower2,upper,work,merged}
  각 레이어에 파일을 생성
# echo "This is lower1" > /tmp/overlay2/lower1/file1.txt
# echo "This is lower2" > /tmp/overlay2/lower2/file1.txt
# echo "This is lower2" > /tmp/overlay2/lower2/file2.txt
# echo "This is upper" > /tmp/overlay2/upper/file3.txt

  lower1, lower2 레이어가 2개인 OverlayFS를 마운트. 먼저나오는 lower1 가 상위에 위치한다.
# sudo mount -t overlay overlay -o lowerdir=/tmp/overlay2/lower1:/tmp/overlay2/lower2,upperdir=
/tmp/overlay2/upper,workdir=/tmp/overlay2/work /tmp/overlay2/merged

# tree /tmp/overlay2/
/tmp/overlay2/
├── lower1
│   └── file1.txt
├── lower2
│   ├── file1.txt
│   └── file2.txt
├── merged
│   ├── file1.txt
│   ├── file2.txt
│   └── file3.txt
├── upper
│   └── file3.txt
└── work
    └── work
   file1은 lower1, lower2 에 동시에 존재함. lower1이 상위에 위치한다
# cat /tmp/overlay2/merged/file1.txt
This is lower1

  신규파일을 생성하거나, 기존 파일을 변경하기
# echo "New file in merged" >  /tmp/overlay2/merged/new_file.txt
# echo "Modified in merged" >> /tmp/overlay2/merged/file1.txt
# tree /tmp/overlay2/
```

31) https://learn.microsoft.com/ko-kr/windows-hardware/drivers/ifs/about-file-system-filter-drivers

```
/tmp/overlay2/
├── lower1
│   └── file1.txt
├── lower2
│   ├── file1.txt
│   └── file2.txt
├── merged
│   ├── file1.txt     <----- 수정된 파일
│   ├── file2.txt
│   ├── file3.txt
│   └── new_file.txt  <----- 새로 생성된 파일
├── upper
│   ├── file1.txt   <----- 수정된 파일
│   ├── file3.txt
│   └── new_file.txt  <----- 새로 생성된 파일
└── work
    └── work
  마운트 해제
# sudo umount /tmp/overlay2/merged
```

7.2 도커와 Overlay2 파일시스템

- 도커(docker) [32]에서 오버레이 파일시스템이 어떻게 활용되는지 확인한다.

```
$ sudo apt  install docker.io         도커를 설치함
# sudo docker version
Client:
 Version:           24.0.7
 API version:       1.43
[...]
Server:
 Engine:
  Version:          24.0.7
  API version:      1.43 (minimum version 1.12)
[...]

      이미지 검색 nginx 웹서버를 검색한다
$ sudo docker search nginx
NAME                                      DESCRIPTION                                     STARS     OFFICIAL
 AUTOMATED
nginx                                     Official build of Nginx.                        20313     [OK]
[...]
    nginx 도커 이미지를 다운로드
$ sudo docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
[...]
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

# docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
```

32) https://hub.docker.com/

```
nginx        latest    3b25b682ea82   4 weeks ago   192MB
```

- 이미지 파일의 메타 데이터는 /var/lib/docker/image/overlay2/imagedb/content/sha256/<image-hash> 경로에 저장된다.

```
# cat /var/lib/docker/image/overlay2/imagedb/content/sha256/3b25b682ea82*  ¦ jq '.'
{
  "architecture": "amd64",
  "config": {
    "ExposedPorts": {
      "80/tcp": {}
    },
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "NGINX_VERSION=1.27.2",
      "NJS_VERSION=0.8.6",
      "NJS_RELEASE=1~bookworm",
      "PKG_RELEASE=1~bookworm",
      "DYNPKG_RELEASE=1~bookworm"
    ],
    "Entrypoint": [
      "/docker-entrypoint.sh"
    ],
    "Cmd": [
      "nginx",
      "-g",
      "daemon off;"
    ],
[...]
  "os": "linux",
  "rootfs": {
    "type": "layers",
    "diff_ids": [    diff_ids 는 도커 이미지의 각 레이어에 대한 SHA256 해시값임. 레이어의 내용이 변경되지 않았음을 확인하기 위해 사용됨. docker inspect <image-id> 명령으로도 확인할 수 있다. 첫번재 요소가 맨 아래 레이어이며, 마지막 요소가 가장 위 레이어이다.
      "sha256:98b5f35ea9d3eca6ed1881b5fe5d1e02024e1450822879e4c13bb48c9386d0ad",
      "sha256:b33db0c3c3a85f397c49b1bf862e0472bb39388bd7102c743660c9a22a124597",
      "sha256:63d7ce983cd5c7593c2e2467d6d998bb78ddbc2f98ec5fed7f62d730a7b05a0c",
      "sha256:296af1bd28443744e7092db4d896e9fda5fc63685ce2fcd4e9377d349dd99cc2",
      "sha256:8ce189049cb55c3084f8d48f513a7a6879d668d9e5bd2d4446e3e7ef39ffee60",
      "sha256:6ac729401225c94b52d2714419ebdb0b802d25a838d87498b47f6c5d1ce05963",
      "sha256:e4e9e9ad93c28c67ad8b77938a1d7b49278edb000c5c26c87da1e8a4495862ad"
    ]
  }
}
 도커 이미지의 ChainID는 이미지 레이어의 구조와 내용을 나타내는 식별자임.
 ChainID는 특정 레이어와 그 아래의 모든 레이어를 포함한는 체인을 고유하게 식별하는 값이다.
- 단일 레이어의 경우: ChainID(A) = DiffID(A)
- 두 개 이상의 레이어일 경우 ChainID(A¦B) = SHA256hex(ChainID(A) + " " + DiffID(B))
  ChainID(A¦B¦C) = SHA256hex(ChainID(A¦B) + " " + DiffID(C))

위의 예시에서 맨 아래와 2번째 레이어의 해시값은 다음과 같다.
"sha256:98b5f35ea9d3eca6ed1881b5fe5d1e02024e1450822879e4c13bb48c9386d0ad" << 맨 아래 layer
"sha256:b33db0c3c3a85f397c49b1bf862e0472bb39388bd7102c743660c9a22a124597" << 그 위의 layer
```

```
두번째 layer의 chainID는 다음과 같이 계산된다.
echo -n "<첫번째hash>" + " " + "<두번째hash>" ¦ sha256sum

# echo -n "sha256:98b5f35ea9d3eca6ed1881b5fe5d1e02024e1450822879e4c13bb48c9386d0ad sha256:b33db
         0c3c3a85f397c49b1bf862e0472bb39388bd7102c743660c9a22a124597" ¦ sha256sum
fbc611fa4a4aff4cf0bfd963c49e2c416ff8047c9f84c2dc9328d3b833f1118d  -

 마찬가지 방법으로 3번째 레이어의 chain ID는
echo -n "<두번째레이어의ChainID>" + " " + "<세번째레이어의hash>" ¦ sha256sum
# echo -n "sha256:fbc611fa4a4aff4cf0bfd963c49e2c416ff8047c9f84c2dc9328d3b833f1118d sha256:63d7c
        e983cd5c7593c2e2467d6d998bb78ddbc2f98ec5fed7f62d730a7b05a0c" ¦ sha256sum
6e433330e8b1553bee0637fac3b1e66c994bb2c0cab7b2372d2584171d1c93d8  -

 7개 레이어에 대하여 ChainID를 계산할 수 있으며, 계산된 정보는 /var/lib/docker/image/overlay2/l
ayerdb/sha256 폴더에 저장된다.
# ls -a /var/lib/docker/image/overlay2/layerdb/sha256
3e8a4396bcdb62aeb916ec1e4cf6450003808083 9f049c498c256742dd842334 <- layer 1+2+3+4+5+6+7
46834c975bf2d807053675d76098806736ee94604c650aac5fe8b5172ab008c8 <- layer 1+2+3+4
6e433330e8b1553bee0637fac3b1e66c994bb2c0cab7b2372d2584171d1c93d8 <- layer 1+2+3
8dd6a711fbdd252eba01f96630aa132c4b4e96961f09010fbbdb11869865f994 <- layer 1+2+3+4+5+6
9368c52198f80c9fb87fc3eaf7770afb7abb3bfd4120a8defd8a8f1a68ff375d <- layer 1+2+3+4+5
98b5f35ea9d3eca6ed1881b5fe5d1e02024e1450822879e4c13bb48c9386d0ad <- layer 1
fbc611fa4a4aff4cf0bfd963c49e2c416ff8047c9f84c2dc9328d3b833f1118d <- layer 1+2

  각 폴더의 cache-id 파일에 실제 이미지 파일의 캐시ID가 존재한다.
# layer1_cache_id=`cat /var/lib/docker/image/overlay2/layerdb/sha256/98b5f35ea9d3eca6ed1881b5fe
    5d1e02024e1450822879e4c13bb48c9386d0ad/cache-id`         가장 아래 레이어의 cache-id

  각 레이어의 실제 파일은 /var/lib/docker/overlay2/ 에 위치한다.
# ls  /var/lib/docker/overlay2/$layer1_cache_id/diff
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  u
sr  var
```

- nginx 이미지를 이용하여 새로운 컨테이너 실행하기

```
  -d는 detach 로 백그라운드로 실행하고 container ID 를 출력한다.
# docker run --name my-nginx -d  -p 8080:80  nginx
ad6ee3f5d371b930a98941d29e8a527c6f5d6847dc733b3265892db7f04ab7a1

   컨테이너 목록 조회
# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS     NAMES
ad6ee3f5d371   nginx     "/docker-entrypoint.…"   5 seconds ago   Up 4 seconds   0.0.0.0:8080->80/tcp,
:::8080->80/tcp   my-nginx

   nginx 가 실행중이므로 http 요청을 테스트함
# curl localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[...]
</body>
</html>
     컨테이너 프로세스를 조회함
```

```
# pstree -tp ¦ grep nginx
[...]
           ├─containerd-shim(5293)─┬─nginx(5314)─┬─nginx(5359)
           │                       │             └─nginx(5360)
[...]
```

프로세스의 cgroup 정보를 조회

```
# cat /proc/5314/cgroup
0::/system.slice/docker-ad6ee3f5d371b930a98941d29e8a527c6f5d6847dc733b3265892db7f04ab7a1.scope
```

네임스페이스 정보를 조회(parent)	네임스페이스 정보를 조회(child)
`# readlink /proc/5293/ns/*`	`# readlink /proc/5314/ns/*`
`cgroup:[4026531835]`	`cgroup:[4026532365]` <-- 분리됨
`ipc:[4026531839]`	`ipc:[4026532294]` <-- 분리됨
`mnt:[4026531841]`	`mnt:[4026532292]` <-- 분리됨
`net:[4026531840]`	`net:[4026532296]` <-- 분리됨
`pid:[4026531836]`	`pid:[4026532295]` <-- 분리됨
`pid:[4026531836]`	`pid:[4026532295]` <-- 분리됨
`time:[4026531834]`	`time:[4026531834]` time과 user를 제외하고
`time:[4026531834]`	`time:[4026531834]` 별도의 namespace로 분리됨
`user:[4026531837]`	`user:[4026531837]`
`uts:[4026531838]`	`uts:[4026532293]` <-- 분리됨

도커 컨테이너 상세 정보 조회 (docker inspect)

```
# docker inspect my-nginx
[
    {
        "Id": "ad6ee3f5d371b930a98941d29e8a527c6f5d6847dc733b3265892db7f04ab7a1",
        "Created": "2024-10-31T12:24:39.642379155Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
[...]
        "GraphDriver": {    도커의 GraphDriver는 overlay2를 이용하여 레이어 관리를 하는 컴포넌트임
            "Data": {    lower 레이어는 여러개의 레이어로 구성되며, 먼저 나열된 것이 상위 레이어에 위치한다
        "LowerDir": "/var/lib/docker/overlay2/3e584cd524a950786d6c6b797b5d9acc7aec346b63d2711e420232ff15deb7a3-init/diff:/var/lib/docker/overlay2/d9675ae00d402d4eddf18c913fc2507cc4d4c08c14716bc0bc5324ef1639beea/diff:/var/lib/docker/overlay2/a792c76a2e0b7fd0154560931f243fe167fb78499045d84c6ba75269713ec074/diff:/var/lib/docker/overlay2/0bc07f43f47721b159bbb764f1bb0138dc00a2d0e092edface9bae9caaa05eee/diff:/var/lib/docker/overlay2/5771dcf82931ebc44b1e4c93233729b6bb534c8eab3c37db6b4298fb8a5a455c/diff:/var/lib/docker/overlay2/f3c66aa76ca90c717f712563c4257c12292396d1e276968c9cd8e07d04df3074/diff:/var/lib/docker/overlay2/2ba1d1a37e191cedd638d8d1523be934f9a9264c3956ed8b88c083f1237b272b/diff:/var/lib/docker/overlay2/bce44f9c74affcd19539b85011547362aaed6dcbb779dd6c3f2883de36b36758/diff",
            "MergedDir": "/var/lib/docker/overlay2/3e584cd524a950786d6c6b797b5d9acc7aec346b63d2711e420232ff15deb7a3/merged",
            "UpperDir": "/var/lib/docker/overlay2/3e584cd524a950786d6c6b797b5d9acc7aec346b63d2711e420232ff15deb7a3/diff",
            "WorkDir": "/var/lib/docker/overlay2/3e584cd524a950786d6c6b797b5d9acc7aec346b63d2711e420232ff15deb7a3/work"
            },
            "Name": "overlay2"
        },
[...]
```

컨테이너의 overlayFS의 merged 폴더

```
# merged_dir=`docker inspect my-nginx ¦ jq '.[].GraphDriver.Data.MergedDir' ¦ tr -d '"'`
# cd $merged_dir

  컨테이너 내부의 index.html 을 변경하고 http 요청을 하여 테스트
# echo "Hello My Container" > usr/share/nginx/html/index.html
# curl localhost:8080
Hello My Container

  실행중인 컨테이너의 PID 알아내기
# pid=`docker inspect my-nginx ¦ jq '.[].State.Pid'`

  nsenter 를 이용하여 해당 PID의 namespaces 에서 쉘을 실행
# nsenter -t $pid   -m -u -i -n -p   /bin/bash

  nginx 의 index.html 내용
# cd /usr/share/nginx/html
# cat index.html
Hello My Container
```

[참고자료]

- Creating a Linux service with systemd https://www.youtube.com/watch?v=xvDXSIcl3vc
- Linux Nice and Priority values https://www.youtube.com/watch?v=II2M3rqgCQA
- Introduction to systemd timers https://www.youtube.com/watch?v=DixhIrgMy3M
- Understanding systemd, Red Hat, https://people.redhat.com/pladd/systemd_NYRHUG_2016-03.pdf
- Controlling Process Resources with Linux Control Groups https://labs.iximiuz.com/tutorials/controlling-process-resources-with-cgroups
- Using cgroups to limit I/O https://andrestc.com/post/cgroups-io/
- The Linux Userspace API Group https://uapi-group.org/
- man systemd.special https://www.freedesktop.org/software/systemd/man/latest/systemd.special.html
- 리눅스 네임스페이스란? https://www.44bits.io/ko/keyword/linux-namespace
- 리눅스 UTS namespace https://braindisk.tistory.com/141
- 사용자(User) 네임스페이스 https://milhouse93.tistory.com/91
- Container 격리 기술 이해하기 https://mokpolar.tistory.com/60
- Linux Capability 101 https://linux-audit.com/kernel/capabilities/linux-capabilities-101/
- Introduction to LXC and LXC vs Virtualization https://www.youtube.com/watch?v=fbRX3wHmBos&list=PLd78WKkBK5Dj7uKDcaQMl9m1wQQigNjJy
- LXD 5.0 LTS의 새로운 기능 https://www.youtube.com/watch?v=WW6dQGIqGsA&t=1793s
- LXC Getting Started https://linuxcontainers.org/lxc/getting-started/
- Containers from scrach https://ericchiang.github.io/post/containers-from-scratch/
- Linux Primitives, Nati & Avishai, https://docs.google.com/presentation/d/10vFQfEUvpf7qYyksNqiy-bAxcy-bvF0OnUElCOtTTRc/htmlpresent
- Linux Container Primitives: cgroups, namespaces, and more! https://www.youtube.com/watch?v=x1npPrzyKfs
- Overlayfs testing https://github.com/amir73il/overlayfs/wiki/Overlayfs-testing
- 컨테이너 인터널, 컨테이너 파일시스템, kakao enterprise Techo& https://tech.kakaoenterprise.com/171

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

ISBN 978-89-294-1798-7 (PDF)