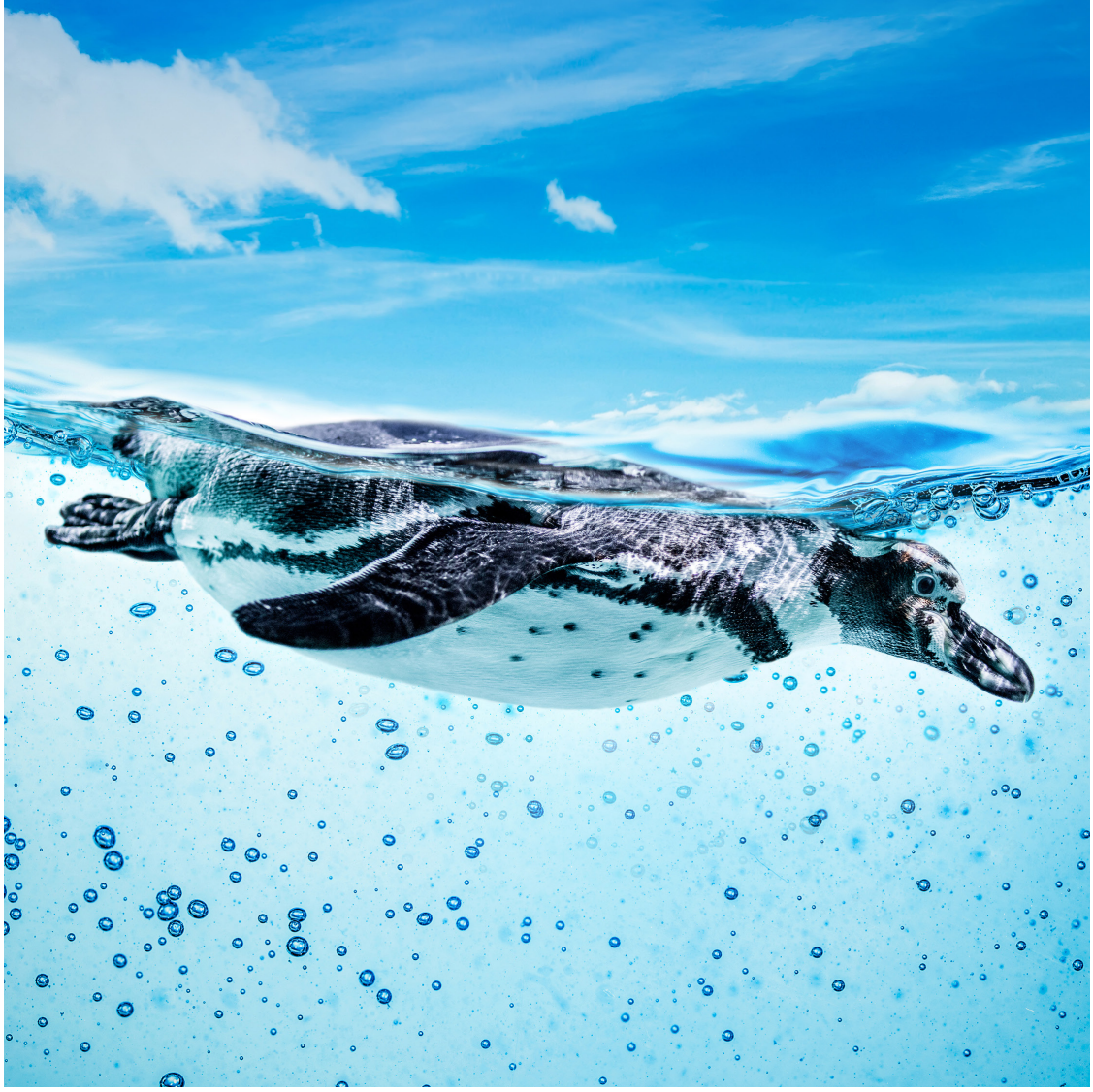
IDG
STRATEGIC
MARKETING SERVICES

IT WORLD CIO

ITWORLD HOW TO

기초부터 다시 시작하는 리눅스 기본 활용편

> 명령어 정보를 찾는 명령어 3총사
> 리눅스용 최고의 셸 프로그램 '배시' 기초 가이드
> "윈도우도 괜찮아" 브라우저에서 리눅스 터미널 사용하기
> 리눅스 true, false 명령어의 숨겨진 진실
> "네가 쓴 명령어를 알고 있다" 리눅스의 hash 사용법 총정리
> 리눅스에서 수치 연산 함수를 사용하는 방법
> "함께 하면 더 강력하다" 파이프로 여러 명령어 동시에 실행하기



기초부터 다시 시작하는 리눅스 기본 활용편

Sandra Henry-Stocker | Network World

전 세계에서 운영되는 웹사이트 10개 중 4개 정도가 리눅스를 사용한다. 500가지 이상의 배포판이 나왔고, 광범위한 안드로이드 기기와 각종 슈퍼 컴퓨터에서 쓰이는 것은 물론 심지어 화성에서 운행 중인 탐사차 '퍼서비어런스'에도 리눅스가 들어갔다.

이처럼 리눅스는 등장한 지 30년이 넘었지만 여전히 가장 중요한 운영체제로 꼽히고, 기업의 IT 환경에서는 특히 더 그렇다. 인프라 운영부터 앱 개발까지 필수 기술로 자리 잡은 리눅스를 기초부터 차근차근 쉽게 접근할 수 있도록 안내한다. 지금부터 리눅스 기본 활용 방법을 중심으로 살펴보자.

01 명령어 정보를 찾는 명령어 3총사 : whereis, whatis, which

리눅스 파일 시스템을 다루면서 특정 명령에 대한 정보를 알아보고자 할 때 whereis, whatis, which 명령이 도움이 될 수 있다. 이들은 명령에 대해 서로 다른 관점의 정보를 제공한다. 여기서는 3가지 명령이 어떤 정보를 제공하는지 알아본다.

which

which 명령은 3가지 중 가장 단순한 명령이다. which를 사용해 리눅스 명령에 대해 질의하면 검색 경로에서 지정된 이름으로 된 실행 파일을 찾는다. 시스템에서 사용할 수 있는 명령일 수도 있고 스크립트일 수도 있다. 파일에 대한 쓰기 권한만 있으면 바로 사용할 수 있다. 예를 들면 다음과 같다.

```
$ which date
/usr/bin/date
$ which init
/usr/sbin/init
$ which loop
~/bin/loop
```

which 명령은 파일의 위치를 보여주는 기능만 한다. 또한 일치하는 파일을 찾으면 바로 검색을 중단한다. 지정된 이름의 실행 파일이 포함된 검색 경로상의 첫 위치가 결과로 표시된다.

일반적으로 which 명령은 명령의 이름을 입력할 때 어느 실행 파일이 실행되는지 알 수 있도록 명령의 위치를 표시하는 데 사용한다. 실행하고자 하는 명령이 아닌 다른 명령이 실행되지 않도록 하는 것이 중요할 때가 종종 있다. 잘 작성된 검색 경로는 정확히 원하는 명령을 실행하는 데 도움이 된다. 이는 일반적으로 /usr/bin, /usr/sbin, /usr/local/bin과 같은 시스템 디렉터리가 개인 디렉터리 및 개인 bin 디렉터리보다 선행한다는 것을 의미한다.

whereis

whereis 명령은 파일을 찾는 접근 방식 측면에서 더 자유롭다. 사용자가 찾는 파일 및 관련 명령을 찾아준다. 또한 이 명령은 바이너리, 소스, 매뉴얼 페이지 파일에서만 명령을 찾는다. 파일은 실행 파일이 아니어도 된다. 사용자의 검색 경로를 따르지 않고 특정 위치에서만 찾는다. 다음 예에서 볼 수 있듯이 bin 디렉터리도 그중 하나다.

```
$ whereis date
date: /usr/bin/date /home/shs/bin/date /usr/share/man/man1/date.1.gz
/usr/share/man/man1p/date.1p.gz
```

이 예에서 whereis 출력에 표시된 마지막 2개 파일은 gzip으로 압축된 매뉴얼 페이지로, man 명령을 사용해 읽을 때 압축이 해제된다.

whatis

whatis 명령은 파일을 찾지 않고, 리눅스 명령에 대한 간략한 설명을 제공한다. 관련 man 페이지에서 정보를 가져온다. 예를 들면 다음과 같다.

```
$ whatis date
date (1)          - print or set the system date and time
date (1p)         - write the date and time
```

여기서 whatis 명령은 date 명령에 대한 2가지 매우 간단한 설명을 제공한다. 첫 번째 설명은 주 man 페이지에서, 두 번째 설명은 1p 폴더에 저장된 man 페이지에서 가져온다(/usr/share/man/man1p/date.1p.gz). man 페이지를 보려면 다음과 같은 명령을 사용하면 된다.

```
$ man date
$ man 1p date
```

관련 man 페이지의 NAME 섹션에서 간략한 설명이 포함된 것을 볼 수 있다.

```
$ man date | head -4
DATE(1)                   User Commands                   DATE(1)

NAME
```

```
date - print or set the system date and time
$ man 1p date | head -10
DATE(1P)                POSIX Programmer's Manual               DATE(1P)

PROLOG
       This  manual  page is part of the POSIX Programmer's Manual.  The Linux
       implementation of this interface may differ (consult the  corresponding
       Linux  manual page for details of Linux behavior), or the interface may
       not be implemented on Linux.

NAME
       date — write the date and time
```

whereis, whatis, which 명령은 정확히 의도한 명령을 실행하고 명령 및 관련 파일을 찾는 데 도움이 되며 명령의 기능에 대한 매우 간단한 설명을 제공한다.

02 리눅스용 최고의 셸 프로그램, '배시' 기초 가이드

리눅스에서 스크립팅(파일에 여러 명령을 넣어 하나의 그룹으로 실행하는 방법)을 사용하면 프로세스를 반복할 필요가 없으므로 명령줄에서 실행하는 것보다 훨씬 더 쉽게 작업할 수 있다. 별칭(Aliases)을 사용해 손쉽게 명령을 반복할 수도 있지만, 보통 별칭은 기억하기 어렵거나 복잡한 개별 명령에만 사용된다.

다음 예에서 볼 수 있는 것처럼 배시(bash) 셸은 스크립트의 테스트, 루프, 함수 만들기, 주석 달기를 위한 풍부한 명령을 제공한다. 이런 스크립팅을 배우기 위한 최선의 방법은 문제를 먼저 구상하고 스크립트로 이 문제를 해결하는 것이다. 몇 가지 기본적인 스크립팅 기법부터 시작하자.

스크립트에 유의미한 이름 부여하기

스크립트의 이름은 스크립트의 기능을 설명하는 역할을 해야 한다. 예를 들어 다음 스크립트는 명령줄에서 입력되는 숫자의 제곱을 제공하므로 'square'라는 이름이 적합하다.

```
$ cat square
#!/bin/bash

number=$1
echo $number^2 | bc
```

이를 실행하면 다음과 같다.

```
$ square 12
144
```

스크립트를 실행할 수 있는 사람에 따라 권한 부여하기

스크립트를 실행하려면 실행 권한이 필요하다. 750 권한을 할당하면 자신과 그룹 내의 모든 사람이 실행할 수 있지만, 편집은 자신만 가능하다.

```
$ chmod 750 square
```

스크립트에 실행 권한이 없더라도 '소싱(sourcing)'하면 실행할 수 있다. 즉, 다음과 같은 명령을 사용해 파일 내용을 한 줄 씩 읽고 실행할 수 있다.

```
$ . square 12
144
```

변수 설정 및 확인

앞서 살펴본 square 스크립트는 이를 실행할 때 제공되는 숫자 변수를 사용해 값을 캡처하는데, 값이 제공되지 않으면 다음과 같이 오류가 발생한다.

```
$ square
(standard_in) 1: syntax error
```

이런 오류는 인수가 스크립트에 제공되는지 확인함으로써 해결할 수 있다. 참고로 이 확인은 제공된 인수가 숫자인지 여부는 테스트하지 않는다.

```
#!/bin/bash

if [ $# == 1 ]; then
  number=$1
  echo $number^2 | bc
fi
```

여기서는 값이 입력되지 않으면 스크립트가 오류로 종료되는 것이 아니라 그냥 아무것도 하지 않는다.

인수 요구 메시지

앞서 살펴 본 square 스크립트 문제를 해결하는 다른 방법도 있다. 필요한 값을 요구하는 메시지를 표시하거나 값이 제공되지 않은 경우에만 메시지를 표시하는 것이다.

```
#!/bin/bash

if [ $# == 1 ]; then
  number=$1
else
  echo -n "number to square: "
  read number
fi

echo $number^2 | bc
```

if 명령을 사용해 테스트 실행

앞선 예에서 볼 수 있듯이, if 명령을 사용해 테스트를 실행해 필요한 인수가 제공되었는지 확인할 수 있다. if 테스트는 많은 요소의 테스트에 사용할 수 있다. 다음 스크립트에서는 금요일에 실행되는지 여부를 확인한다.

```
#!/bin/bash

if [ `date +%A` = "Friday" ]; then
  echo Send weekly report
else
  echo Add daily updates to weekly report
fi
```

for, while과 같은 루프 명령 사용하기

스크립트에서 루프는 필요한 만큼, 또는 스크립트를 종료할 때까지 명령을 반복할 수 있으므로 상황에 따라 매우 유용하다. for 명령을 사용하는 몇 가지 간단한 예를 살펴보자. 첫 번째 예는 알파벳을 순서대로 진행하면서 각 문자 사이에서 10초 동안 멈춘다. 두 번째는 시스템의 홈 디렉터리를 순환하며 실행하면서 각 디렉터리가 사용 중인 디스크 공간을 보고한다. 개별 사용자는 다른 홈 디렉터리에는 액세스할 수 없으므로 sudo로 실행돼야 한다.

첫 번째 스크립트는 다음과 같다.

```
#!/bin/bash

for x in {a..z}
do
   echo $x
   sleep 10
done
```

두 번째 스크립트는 다음과 같다.

```
#!/bin/bash

for user in `ls /home`
do
   du -skh /home/$user
done
```

while 명령은 조건이 참인 동안 계속 실행된다. "while true"를 사용하면 오류가 발생할 때까지 루프가 계속 실행된다.

```
#!/bin/bash

while true
do
  who
  sleep 5
done
```

앞선 스크립트는 ^c를 사용해 멈출 때까지 실행된다. 반면 다음 스크립트는 앞에 라인 번호가 붙은 파일의 각 라인을 표시하는 작업을 마치면 멈춘다.

```
#!/bin/bash

echo -n "File> "
read file
n=0

while read line; do
 ((n++))
 echo "$n: $line"
done < $file
```

스크립트에 주석 추가하기

주석은 스크립트의 목적을 이해하거나 기억하는 데 많은 도움이 된다. 스크립트의 기능이나 복잡한 명령을 명확히 설명하는 주석을 추가하는 것이 좋다. 주석은 # 기호로 시작하며 라인의 맨 앞이 아니어도 된다.

```
#!/bin/bash
# calculate space used by home directories

for user in `ls /home`
do
   du -sk /home/$user              # show size of each home directory
done
```

리턴 코드를 통해 스크립트의 명령이 성공적으로 실행되었는지 손쉽게 확인할 수 있다. 앞선 예제 스크립트에서는 다음과 같은 방법을 사용하면 된다.

```
#!/bin/bash
# create or update a file

if [ $# == 1 ]; then    # one argument expected
  touch $1 2> /dev/null
else
  exit 1
fi

# report on whether the command was successful
if [ $? -eq 0 ]        # $? is return code for the touch command
then
  echo "Successfully created file"
else
  echo "Could not create file"
fi
```

이 스크립트는 touch 명령으로 파일을 만들거나 업데이트한다. 권한 문제로 인해 touch 명령이 성공적이지 않은 경우 리턴 코드($?)는 0이 아니고, 이는 실패를 나타낸다.

```
$ touchFile /etc/whatever
Could not create /etc/whatever
```

앞의 사례에서 touch 명령에서 오류 출력은 오류가 화면에 표시되는 것을 방지하기 위해 /dev/null로 전달된다. 1 이상의 오류 코드는 파일이 생성되지 않았음을 설명하는 메시지로 이어지며, 필요한 경우 "이해하기 쉬운" 오류 메시지를 만드는 데 사용할 수 있다.

필요할 때 호출되는 함수 만들기

함수를 이용하면 스크립트에 있는 코드의 일정 부분을 필요한 만큼 많이, 손쉽게 반복할 수 있다. 예를 들어 주기적으로 디렉터리를 생성한 다음 cd로 옮겨야 하는 스크립트를 작성한다면, 다음과 같은 함수를 추가한 다음 "newdir report" 또는 "newdir backups"와 같은 명령을 사용해 호출하면 된다.

```
newdir () {
   mkdir -p $1
   cd $1
}
```

이렇게 하면 명령줄에서 함수를 설정하고 사용할 수 있지만, 로그아웃 시 유지되지 않는다.

```
$ test_function () { echo This is a function; }
$ test_function
This is a function
```

스크립트를 작성하는 데 익숙해지면 일상적인 작업을 쉽게 해주는 스크립트를 만들어 재미있고 유익하게 사용할 수 있다. 잘 작성된 스크립트는 많은 시간을 절약해주고, 자신이 개발한 문제 해결 방법을 보존하는 방법이 되기도 한다.

03 "윈도우도 괜찮아" 브라우저에서 리눅스 터미널 사용하기

리눅스 터미널을 사용해보고 싶은데 당장 리눅스 시스템이 없더라도 실망할 필요는 없다. 브라우저 내에서 리눅스 터미널을 실행할 수 있는 여러 서비스가 있기 때문이다. 이런 서비스의 종류를 알아보고 무엇을 할 수 있고 성능은 어느 정도인지 살펴보자. 여기서 설명하는 리눅스 터미널 세션은 모두 윈도우 시스템에서 크롬 브라우저를 사용해 실행한다. 리눅스 시스템의 브라우저에서도 리눅스 터미널을 실행할 수 있지만 굳이 그렇게 해야 할 이유는 없을 것이다.

JSLinux

JSLinux는 기본적으로 자바스크립트로 구현되는 컴퓨터다. 브라우저를 열고 URL만 제대로 입력하면 바로 시작할 수 있다. 다음 사이트에서 JSLinux로 연결되는 링크를 찾을 수 있다.

https://bellard.org/jslinux/

나열된 8개 시스템 중에서 6개가 리눅스인데 콘솔 창을 지원하는 중 눈여겨 볼 것은 다음 3가지다.

- 알파인 리눅스(Alpine Linux) 3.12.0 : https://bit.ly/3upej6i
- 빌드루트(Buildroot) : https://bit.ly/3leyoBT
- 페도라(Fedora) 33 : https://bit.ly/3Rnsu5V

필자는 이 중에서 페도라 33을 선호한다. 다른 둘과 달리 man 페이지가 포함돼 있기 때문이다. 루트로 로그인하지만 자신의 존재를 확인하는 who 명령은 사용할 수 없다. 그래도 whoami 및 pwd 명령으로 ID를 확인할 수는 있다.

```
localhost:~# who; whoami; pwd
sh: who: not found
root
```

```
/root
```

원하는 경우 hello.c 프로그램을 컴파일해서 실행할 수 있는데, 다음과 같은 화면이 표시된다.

```
localhost:~# cc -o hello hello.c
localhost:~# ls
bench.py hello hello.js hello.c readme.txt
localhost:~# hello
sh: hello: not found
localhost:~# ./hello
hello world
```

즐겨 사용하는 리눅스 명령을 실행하고 스크립트 한두 개를 만들어 명령줄을 살펴보는 것도 좋다. 필자는 검색 경로의 각 디렉터리에 있는 파일의 수를 세는 간단한 배시 스크립트 만들어 실행해 봤다.

```
$ cat count_commands
#!/bin/bash

for dir in `echo $PATH | sed "s,:, ,g"`
do
  echo $dir
  ls $dir | wc -l
  echo "==========="
done
[root@localhost ~]# ./count_commands
/usr/local/sbin
0
===========
/bin
2349
===========
/sbin
609
===========
/usr/bin
2349
===========
/usr/sbin
609
===========
/usr/local/bin
9
```

확실히 이 시스템에는 많은 리눅스 명령이 있다. 스크립트 실행 시 문제가 생기면 다음과 같이 소싱하면 된다.

```
[root@localhost ~]# ./count_commands
sh: ./count_commands: not found
[root@localhost ~]# . count_commands
```

필자는 시스템 중 하나의 검색 경로에 현재 파일 시스템 위치에 있었음에도 소싱을 해야 제대로 작동했다. 검색 경로를 확인하려면 다음과 같은 명령을 사용한다.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

페도라의 man 페이지 디렉터리는 다음과 같다.

```
[root@localhost ~]# ls /usr/local/share/man
man1  man2  man3  man4  man5  man6  man7  man8  man9  mann
man1x  man2x  man3x  man4x  man5x  man6x  man7x  man8x  man9x
[root@localhost ~]# ls /usr/share/man
ca  es  it    man1  man2x  man4  man6  man8  mann  pt_BR  sv    zh_TW
cs  fr  ja    man1p  man3  man4x  man6x  man8x  nl   ru    tr
da  hu  ko    man1x  man3p  man5  man7  man9  pl   sk    uk
de  id  man0p  man2  man3x  man5x  man7x  man9x  pt   sr    zh_CN
```

man 페이지를 사용할 수 있을 때 man 페이지 명령을 실행하면 예상대로 동작한다.

```
[root#localhost !]# man date
[root@localhost ~]# DATE(1)      User Commands      DATE(1)

NAME
      date - print or set the system date and time

SYNOPSIS
      date [OPTION]... [+FORMAT]
      date [-u|—utc|—universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
      Display the current time in the given FORMAT, or set the system date.

      Mandatory  arguments  to  long  options are mandatory for short options
      too.
...
```

사용 중인 시스템의 IP 주소에 대한 정보를 요청하자 루프백(127.0.0.1) 인터페이스와 내부 10.x.x.x 주소가 표시됐다.

```
localhost:~$ ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNO
WN qlen 1000
    link/ether 02:46:81:31:ca:a3 brd ff:ff:ff:ff:ff:ff
    inet 10.5.218.60/16 brd 10.5.255.255 scope global dynamic eth0
       valid_lft 817sec preferred_lft 667sec
```

참고로 JSLinux 콘솔 중 하나를 열면 항상 같은 곳, 즉 새 리눅스 터미널에서 시작된다. 스크립트 또는 변경 사항은 어떤 방식으로도 보존되지 않는다.

copy.sh

copy.sh 역시 브라우저 내에서 리눅스(또는 다른 여러 OS)를 실행할 수 있는 가상화 툴이다. 사용 가능한 모든 옵션을 보려면 http://copy.sh/v86/을 참고하면 된다. 리눅스 외에도 윈도우, 프리BSD, 오베론(Oberon)을 비롯한 20여 가지 버전을 선택할 수 있다. 이 중 필자가 확인한 옵션은 다음과 같다.

- linux26 : http://copy.sh/v86/?profile=buildroot
- archlinux : https://copy.sh/v86/?profile=archlinux

댐 스몰 리눅스(Damn Small Linux) 옵션은 그래픽 인터페이스를 제공하는데, 필자의 경우 아직 다 살펴보지는 못했다.

- 댐 스몰 리눅스: http://copy.sh/v86/?profile=dsl

스크립트를 실행하는 방법은 사용 중인 배포판에 따라 다르다. 빌드루트 터미널에서는 스크립트를 소싱해야 했지만, 아치리눅스(archlinux) 터미널에서는 그럴 필요가 없었다.

```
~% cat showme
#!/bin/bash

echo "Hi, there"
echo -n "What are you looking for?: "
read ans
echo "Sorry, I have never heard of coffee"
~% .showme
./showme: not found
~% . ./showme
Hi, there
```

```
What are you looking for?: coffee
Sorry, I have never heard of coffee
```

count_commands 스크립트도 실행해 봤다.

```
~% ./count_commands
/sbin
55
==========
/usr/sbin
32
==========
/bin
75
==========
/usr/bin
131
==========
```

copy.sh에서 매우 마음에 들었던 점 중 하나는 "상태 저장" 및 "상태 로드" 옵션이 제공된다는 것이다. 즉, 추가한 스크립트를 시스템에 저장된 v86state.bin 파일에 보존해서 다음에 연결할 때 복구할 수 있다. 반면 copy.sh 터미널에서 겪은 한 가지 이상한 문제는 터미널에서 트랙볼을 빼내기 위해 Control+Alt+Delete를 누른 다음 "취소(Cancel)"를 눌러야 했다는 점이다.

브라우저 내 터미널 옵션의 경우 속도가 다소 들쭉날쭉하지만 브라우저 내에서 리눅스를 사용하고 기능을 살펴볼 수 있다는 점은 유용하다. 많은 리눅스 명령을 사용할 수 있고, 몇 가지 불편한 점과 성능 문제에도 불구하고 가상화된 리눅스 시스템은 경우에 따라 상당히 유용하다.

04 리눅스 true, false 명령어의 숨겨진 진실

True와 false는 모든 형태의 컴퓨팅에서 보편적인 개념이다. 부울 로직(Boolean logic)의 중요한 요소이기도 하다. 그러나 리눅스에서는 true와 false가 놀랍게도 명령어다.

그 사용법을 최대한 간단히 설명하면, true 명령은 종료 코드 0을 생성하고 false 명령은 종료 코드 1을 만든다. 그러나 이 설명으로는 이 두 명령을 가장 잘 사용하는 방법을 구체적으로 알 수 없다. 여기서는 true와 false 명령이 어떻게 작동하는지, 명령줄 또는 스크립트에서 어떻게 사용하는지 알아본다.

종료 코드 보기

먼저, 중요한 점은 리눅스 시스템의 성공적인 종료 코드(다시 말해 '리턴 코드(return code)')는 0이라는 것이다. 오류가 '없음(0)'을 의미한다고 생각하면 된다. 어떤 형태로든 실패를 나타내는 종료 코드는 1 이상의 값을 갖는다.

리눅스에서는 명령을 실행할 때마다 종료 코드가 생성된다. 정상적인 출력이나 오류 메시지는 그냥 표시되지만 종료 코드는 요청을 해야만 표시된다. 요청하려면 echo $? 명령을 사용한다. $? 문자열은 종료 코드를 나타내며 echo 명령은 다음과 같이 코드를 표시한다.

```
$ echo hello
hello
$ echo $?
0
```

echo hello 명령은 성공적이었으므로 종료 코드는 0이다. 이제 명령이 성공적이지 않은 경우의 종료 코드를 표시하는 간단한 예를 보자. 키보드에서 아무 키나 몇 개 누르면 다음과 같은 화면이 표시된다.

```
$ asjdlkjdad
bash: asjdlkjdad: command not found...
```

여기서 바로 종료 코드를 요청하면 다음이 표시된다.

```
$ echo $?
127
```

종료 코드 127은 방금 입력한 명령이 시스템에 존재하지 않는다는 의미다. 존재하지 않는 파일을 표시하려고 하는 다른 예를 보자.

```
$ cat dhksdfhjksfjhskfhjd
cat: dhksdfhjksfjhskfhjd: No such file or directory
$ echo $?
1
```

종료 코드 1은 일반적인 종료 코드이며 다양한 오류에서 반환된다.

true와 false 사용하기

true 및 false 명령이 명령줄에서 어떻게 작동하는지 보는 가장 간단한 방법은 다음 명령을 실행하는 것이다.

```
$ true; echo $?
0
$ false; echo $?
1
```

여기서 볼 수 있듯이 단순하게 true는 0을 반환하고 false는 1을 반환한다. true 명령의 가장 일반적인 용도는 무한 루프 시작이다. while [$num -le 12345678]과 같은 명령으로 루프를 시작하는 대신 while true를 사용하면 ^c로 멈출 때까지 루프가 계속된다.

```
while true
do
   echo “still running”
   sleep 10
done
```

while false 루프는 즉시 실패한다. 무한 루프를 시작하는 또 다른 방법은 다음과 같은 구문을 사용하는 것이다.

```
until false; do
   echo still running
   sleep 10
done
```

^c를 눌러 무한 루프를 중단한 다음 오류 코드를 확인하면 다음과 같이 표시된다.

```
$ run-forever
still running
still running
^C$ echo $?
130
```

오류 코드 130은 루프가 ^c에 의해 종료되었음을 의미한다.

```
$ more run-forever
until false; do
   echo still running
   sleep 10
done
```

명령 자체가 실패하더라도 명령의 결과로 성공적인 종료 코드가 표시되기를 원한다면 다음과 같이 출력을 true로 파이프로 연결할 수 있다.

```
$ cat nosuchfile | true; echo $?
cat: nosuchfile: No such file or directory
0            <== exit code
```

오류 메시지는 여전히 표시되지만 종료 코드는 0(성공)이 된다. 다음과 같은 if 테스트를 사용하는 경우 if true는 항상 지정된 명령을 실행하며 if false는 임베딩된 명령을 실행하지 않는다.

```
if true
> then
>    echo This command always runs
> fi
This command always runs
if false
> then
>    echo This command never runs
> fi
$
```

여기서 볼 수 있듯이 if false 명령에는 출력이 없다. echo 명령이 실행되지 않기 때문이다. true 대신 콜론을 사용하는 경우 true와 같은 효과를 얻게 된다. 예를 들면 다음과 같다.

```
$ if :
> then
>    echo This command always runs
> else
>    echo This command never runs
> fi
This command always runs
```

나만의 종료 코드를 설정할 수도 있다. 예를 들어 다음 예와 같이 스크립트에 exit 111 명령이 있는 경우, 이 스크립트의 종료 코드는 111이 된다.

```
#!/bin/bash

if [ $# == 0 ]; then
   echo "$0 filename"
   exit 1
fi

echo $0
exit 111
```

스크립트를 실행하면 다음과 같은 화면이 표시된다.

```
$ myscript oops
/home/shs/bin/oops
```

파일의 전체 이름이 확인된다. 종료 코드를 확인하면 다음과 같은 화면만 표시된다.

```
$ echo $?
```

```
111
```

true와 false 명령의 기능은 제한적이지만 종료 코드를 제어해야 하거나 원하는 시점까지 명령을 실행하고자 할 때 유용하다.

05 "네가 쓴 명령어를 알고 있다" 리눅스의 hash 사용법 총정리

리눅스 시스템에서 'hash'를 입력하면 사용하는 셸에 따라 2가지 매우 다른 결과를 얻게 된다. 배시 또는 ksh 등 관련 셸을 사용하면 터미널 세션이 시작된 이후 사용한 명령 목록이 표시되고, 경우에 따라 각 명령이 쓰인 횟수도 함께 볼 수 있다. 최근 명령 활동을 보는 것이 목적이라면 history 명령을 사용하는 것보다 더 유용할 수 있다. 단, hash 명령은 실행 파일이 아니라 현재 사용 중인 셸을 기준으로 작동한다.

배시의 hash

배시의 hash 명령 실행 예를 보면 다음과 같다.

```
$ hash
hits    command
   1    /usr/bin/who
   1    /usr/bin/vi
   1    /usr/bin/man
   2    /usr/bin/ls
   1    /usr/bin/clear
   3    /usr/bin/cat
   3    /usr/bin/ps
```

배시의 경우 hash 명령이 내장돼 있다. 별도의 실행 파일이 아니라 셸에 포함돼 있다. pwd, echo 등 일부 명령은 hash 출력에서 잡히지 않는다. 다음 명령으로도 같은 결과를 얻을 수 있다.

```
$ builtin hash "$@"
hits    command
   1    /usr/bin/who
   1    /usr/bin/vi
   1    /usr/bin/man
   2    /usr/bin/ls
   1    /usr/bin/clear
   3    /usr/bin/cat
   3    /usr/bin/ps
```

필자는 페도라 시스템의 /usr/bin/hash 스크립트에서 이 명령을 찾았다. 다음과 같은 형태의 /usr/bin/sh 스크립

트다.

```
$ cat /usr/bin/hash
#!/usr/bin/sh
builtin hash "$@"
```

이 스크립트를 실행해도 출력은 나오지 않는다.

```
$ /usr/bin/hash
$          <== no output
```

왜일까? 스크립트가 새 셸을 시작하는데 이 셸에서는 실행된 명령이 없으므로 hash 명령이 보고할 내용도 없기 때문이다. 반면 "."를 사용해 스크립트를 소싱하면 매우 다른 결과를 얻게 된다.

```
$ . /usr/bin/hash
Hits    command
   1    /usr/bin/who
   1    /usr/bin/vi
   1    /usr/bin/man
   2    /usr/bin/ls
   1    /usr/bin/clear
   3    /usr/bin/cat
   3    /usr/bin/ps
```

ksh의 hash

ksh의 경우 "hash"를 입력하면 역시 현재 세션에서 실행한 명령 목록이 표시되지만 "Hits" 열이 없으므로 사용된 명령과 실행 파일을 이름=경로 형식으로 보여준다. 이는 셸이 애초에 이 정보를 수집하고 있는 이유를 명확하게 파악하는 데 도움이 된다. 사용하는 모든 명령에 대한 실행 파일을 찾기 위해 셸이 검색 경로를 살펴야 하는 경우 이런 실행 파일이 어디에 있는지 '기억한다면' 명령을 사용할 때마다 매번 경로를 검색할 필요가 없는 것이다.

```
$ hash
bash=/usr/bin/bash
cat=/usr/bin/cat
column=/usr/bin/column
date=/usr/bin/date
ls=/usr/bin/ls
man=/usr/bin/man
ps=/usr/bin/ps
```

ksh의 경우 hash는 별칭으로 표시된다.

```
$ alias | grep hash
```

```
hash='alias -t —'
```

별칭과 연결된 명령을 실행하면 같은 출력을 얻게 된다.

```
$ alias -t —
bash=/usr/bin/bash
cat=/usr/bin/cat
column=/usr/bin/column
date=/usr/bin/date
ls=/usr/bin/ls
man=/usr/bin/man
ps=/usr/bin/ps
```

zsh의 hash

zsh 등 다른 셸에서 hash 명령은 매우 상이한 결과를 제공한다. 현재 셸의 명령 사용 내역이 아닌 명령과 각 명령의 파일 시스템 위치의 목록이 길게 표시된다. 시스템이 기억하는 위치 목록이 얼만큼 긴지 보려면 다음과 같은 명령을 사용한다.

```
shs@dragonfly ~ % hash | wc -l
2753
```

목록을 보려면 명령 출력을 more 명령에 파이프로 연결하면 된다.

```
shs@dragonfly ~ % hash | more
.keepme=/home/linuxbrew/.linuxbrew/bin/.keepme
7z=/usr/bin/7z
7za=/usr/bin/7za
AtomicParsley=/usr/bin/AtomicParsley
BoD_meeting=/home/shs/bin/BoD_meeting
EZscript=/home/shs/bin/EZscript
Mail=/usr/bin/Mail
ModemManager=/usr/sbin/ModemManager
NetworkManager=/usr/sbin/NetworkManager
VBoxClient=/usr/bin/VBoxClient
VBoxClient-all=/usr/bin/VBoxClient-all
VBoxControl=/usr/bin/VBoxControl
```

이 경우 출력에는 사용자가 실행한 명령 외의 다른 명령도 포함된다. 셸은 시스템 명령만 추적하는 것이 아니라 홈 디렉터리에 추가한 스크립트의 위치도 기록한다. 새 스크립트를 실행하면 hash 출력에 스크립트가 추가되며 다음 번 셸을 시작할 때 볼 수 있다.

```
shs@dragonfly ~ % hash | grep shs | head -10
BoD_meeting=/home/shs/bin/BoD_meeting
EZscript=/home/shs/bin/EZscript
about=/home/shs/bin/about
append=/home/shs/bin/append
backups.log=/home/shs/bin/backups.log
bash.pg=/home/shs/bin/bash.pg
bigfile=/home/shs/bin/bigfile
bigfile2=/home/shs/bin/bigfile2
calcPower=/home/shs/bin/calcPower
newscript=/home/shs/bin/newscript
```

hash 명령의 결과는 사용 중인 셸, 내가 실행한 명령에 대한 참조를 유지하는 방법, 시스템의 실행 파일에 따라 달라진다. 명령 기록을 유지하는 주요 목적은 해당 명령을 찾는 데 필요한 시간을 줄임으로써 응답 시간을 더 빠르게 하는 데 있다.

06 리눅스에서 수치 연산 함수를 사용하는 방법

리눅스 시스템에서 많은 양의 계산을 계획한다면, 배시의 강력한 기능을 활용해서 빠른 함수를 만든 다음 이를 반복적으로 사용해 계산을 수행하면 된다. 여기서는 연산 함수를 어떻게 사용하는지, 계산의 정확함을 보장하기 위해 주의할 점은 무엇인지 알아본다.

먼저 다음 수학 함수를 예로 들어 살펴보자.

```
$ ? () { echo "$*" | bc ; }
```

이 명령은 bc 계산기 명령에 인수로 제공한 값과 수학 연산자를 전달하는 함수를 설정한다. 이 함수를 호출하려면 "?"를 입력하고 그 뒤에 인수를 붙이기만 하면 된다. 다음 첫 번째 예에서 인수는 1이고 뒤에 곱셈 문자 "*", 2, "+" 기호, 3이 있다. 결과는 5다.

```
$ ? 1*2+3
5
```

이처럼 빠른 함수를 설정하면 "?"와 인수만 사용해 긴 일련의 계산을 실행할 수 있다. 즉, 다음과 같이 각 계산을 수행할 필요가 없다.

```
$ echo 19*2+5 | bc
```

```
43
$ echo 2+5*11 | bc
57
```

대신 다음과 같이 계산에만 초점을 맞추면 된다.

```
$ ? 19*2+5
43
$ ? 2+5*11
57
```

참고로 이 방법으로 정의한 함수는 다음 .bashrc 파일의 맨 아래 라인과 같이 .bashrc 파일에 추가하지 않는 한 로그아웃하고 나면 더는 사용할 수 없다.

```
$ tail -1 .bashrc
? () { echo "$*" | bc ; }
```

중요한 점은 bc의 경우 계산의 곱셈 또는 나눗셈 부분이 덧셈과 뺄셈보다 우선한다는 것이다. 즉, 아래 첫 번째 예에서 19*2는 5를 더하기 전에 계산된다. 두 번째 예에서 5*19는 2를 더하기 전에 계산된다.

```
$ ? 19*2+5
43
$ ? 2+5*19
97
```

덧셈 또는 뺄셈보다 곱셈 또는 나눗셈이 우선하는 일반 우선순위를 바꾸려면 다음과 같이 덧셈 부분을 괄호로 묶은 명령을 사용한다.

```
$ ? '(2+5)*19'
133
```

그러면 "2+5"(즉, 7)가 먼저 계산되고 그 결과에 19를 곱하게 된다(작은 따옴표로 둘러싸야 한다는 것도 기억하자). 물론 bc 명령은 덧셈과 곱셈만으로 제한되지 않는다. 다음 명령 예에서는 11의 제곱을 계산한다.

```
$ ? 11^2
121
```

음수도 제곱할 수 있다. 다음과 같이 -2의 제곱은 4다.

```
$ ? -2^2
4
```

더 높은 거듭제곱을 사용한 계산도 가능하다. 예를 들어 다음 첫 번째 예에서는 -2의 세제곱을 계산하며(-8) 두

번째는 -2의 8제곱을 계산한다(256).

```
$ ? -2^3
-8
$ ? -2^8
256
```

다음 예에서는 먼저 121을 2로 나눈다. 60은 정확하지는 않지만 % 연산자를 사용해서 나머지를 볼 수 있다.

```
$ ? 121/2
60
$ ? 121%2
1
```

더 정확한 답을 필요하면 척도, 즉 bc에 결과에 표시할 소수 자릿수를 지정할 수 있다.

```
$ ? 'scale=2;121/2'
60.50
```

빠른 함수의 다른 용도

지금까지 bc 명령을 다루기 위한 빠른 함수를 설정하는 방법을 설명하고 bc 명령의 연산자를 살펴봤다. 그러나 빠른 함수 설정이 bc 사용에만 제한되는 것은 아니다. 현재 시간을 반복적으로 알려주는 기능을 원한다면 다음과 같은 함수를 사용하면 된다.

```
$ ? () { echo -n "It's already "; date; echo "Work faster!"; }
```

이제부터는 일을 빨리 하라는 잔소리를 듣고 싶을 때마다 간단히 "?"만 입력하면 된다. 이 빠른 함수는 아무런 인수도 요구하지 않으므로 그 외에는 아무것도 필요 없다.

```
$ ?
It's already Mon Apr 18 04:33:43 PM EDT 2022
Work faster!
$ ?
It's already Mon Apr 18 04:33:51 PM EDT 2022
Work faster!
```

힘들이지 않고 실행할 수 있는 더 유용한 명령을 떠올릴 수도 있을 것이다. 실제로 "?" 외에 "+"와 "@" 기호를 사용해 한번에 여러 빠른 함수를 설정할 수도 있다.

07 "함께 하면 더 강력하다" 파이프로 여러 명령어 동시에 실행하기

리눅스에서 필자가 가장 좋아하는 기능은 파이프를 사용해 여러 명령을 연결해 공들이지 않고도 많은 작업을 하는 것이다. 원하는 형식으로 결과를 출력할 수도 있는데, 파이프 자체뿐만 아니라 리눅스 명령의 유연함 덕분이다. 명령을 실행하고 출력에서 원하는 부분을 선택하고 결과를 정렬하거나 특정 문자열을 대조하기 위해 필요한 항목만 남도록 결과를 축약할 수 있다.

여기서는 파이프의 강력함을 보여주는 두 가지 명령과 여러 명령을 쉽게 조합해 사용하는 방법을 살펴본다.

chkrootkit 상태 확인

첫 번째 명령 예는 sudo를 사용해 chkrootkit 명령을 실행하는 것이다. 이 명령은 알려진 루트킷과 관련 특징을 탐지하는 세부적인 프로세스를 사용해 시스템에서 루트킷 징후를 확인한다. 그런데 이 명령에서 생성하는 코드는 100라인을 훌쩍 넘기는 경우가 많다. 따라서 발견한 정보를 유용한 수준으로 요약하려면 다음 명령을 실행하면 된다.

```
$ sudo chkrootkit | awk '{print $(NF-1) " " $NF}' | sort | uniq -c
      1 a while...
      2 enp0s25: PF_PACKET(/usr/sbin/NetworkManager)
      1 is `/'
     21 not found
      3 nothing deleted
      2 nothing detected
     56 nothing found
     41 not infected
      3 not tested
      1 PF_PACKET sockets
      1 pts/0 bash
      1 suspect files
      1 TTY CMD
      1 /usr/lib/.build-id /usr/lib/debug/.dwz
      1 /var/run/utmp !
```

이 명령은 chkrootkit을 루트로 실행하고 각 출력 라인의 마지막 두 문자열만 선택하고 결과를 정렬한 다음 각 2문자열 결과가 반환된 횟수를 계산한다. 전체 출력 확인을 대신할 수는 없지만 발견 가능한 루트킷과 관련해 시스템 상태에 대해 많은 것을 알 수 있다.

이 출력 결과를 보면 대부분이 우리가 보고자 하는 정보임을 손쉽게 알 수 있다. "nothing deleted"와 "nothing detected"도 좋지만 41 “not infected” 메시지는 확실히 반가운 내용이다. 하나의 “suspect files” 메시지는 사실

상 “no suspect files”에 해당하는데, 이 경우 원본 출력을 거듭 확인해야 한다.

전체 명령의 awk 식은 마지막 두 필드를 표시한다. NF는 awk가 필드의 수를 표시하는 방법이므로 $NF는 마지막 필드의 값이고 $(NF-1)은 그 앞 필드의 값이다. 따옴표 안의 빈 칸은 두 필드가 서로 붙지 않도록 한다. sort 명령은 모든 출력을 알파벳순으로 정렬하고 마지막 명령인 uniq -c는 각 출력 라인이 전체 출력의 각 순차 그룹에서 나타난 횟수를 계산한다.

이처럼 모든 통계가 어디서 오는지 알면 도움이 된다. chkrootkit 출력을 직접 살펴보면 다음과 같은 라인이 많이 보일 것이다.

```
ROOTDIR is `/'
Checking `amd'... not found
Checking `basename'... not infected
Checking `biff'... not found
Checking `chfn'... not infected
Checking `chsh'... not infected
Checking `cron'... not infected
Checking `crontab'... not infected
Checking `date'... not infected
Checking `du'... not infected
Checking `dirname'... not infected
Checking `echo'... not infected
Checking `egrep'... not infected
Checking `env'... not infected
Checking `find'... not infected
Checking `fingerd'... not found
Checking `gpm'... not found
Checking `grep'... not infected
```

결과를 보면 chkrootkit이 명령 감염 가능성을 많이 확인했지만 문제가 발견되지 않았음을 알 수 있다. 잠시 후에 시스템에서 악성코드 가능성을 나타내는 징후가 발견되지 않았음을 나타내는 다음과 같은 라인을 볼 수 있다.

```
Searching for sniffer's logs, it may take a while... nothing found
Searching for HiDrootkit's default dir... nothing found
Searching for t0rn's default files and dirs... nothing found
Searching for t0rn's v8 defaults... nothing found
```

파이프로 연결한 명령은 정보를 유용하게 요약해 제공하며, 손쉽게 스크립트로 변환할 수 있으므로 사용할 때마다 입력하거나 기억할 필요가 없다.

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

```
sudo chkrootkit | awk '{print $(NF-1) " " $NF}' | sort | uniq -c
```

필자의 시스템에서 명령 실행 시간은 약 30초에 불과했다. 주기적으로 이런 확인 작업을 매우 쉽게 실행할 수 있게 해주는 기능이다.

사용자 프로세스

특정 사용자가 실행 중인 프로세스와 연결된 프로세스 ID 목록을 생성하려면 다음과 같은 명령을 사용한다.

```
$ ps aux | grep nemo | grep -v grep | awk '{print $2}'
903665
903674
903680
903695
903703
```

파이프로 연결된 이 명령은 ps aux를 사용해 실행 중인 모든 프로세스를 나열한 후 이 중 사용자 nemo가 실행한 프로세스만으로 추리고 "grep nemo" 명령을 제외한 다음(nemo가 실행한 것이 아니므로) 프로세스 ID만 남도록 목록을 축약한다. 모든 프로세스 ID 대신 프로세스의 수만 보려면 파이프를 하나 더 추가하고 wc -l 명령을 사용하면 된다.

```
$ ps aux | grep nemo | grep -v grep | awk '{print $2}' | wc -l
5
```

시스템 콘솔에 로그인한 사용자를 관찰 중이라면 출력을 column 명령에 파이프로 연결해 한 화면에서 모든 프로세스를 볼 수 있다.

```
$ ps aux | grep shs | grep -v grep | awk '{print $2}' | column
4508   4620   4728   4764   4802   4856   4884   4893   5030   6003
4515   4621   4729   4770   4814   4868   4886   4895   5046   897442
4528   4623   4737   4774   4819   4869   4888   4896   5049   897447
4538   4650   4741   4775   4821   4878   4889   4897   5092   897455
4541   4703   4742   4781   4827   4879   4890   4992   5258   904175
4543   4710   4745   4788   4839   4880   4891   5016   5276   904178
4545   4723   4753   4790   4846   4881   4892   5021   5997   904179
```

파이프는 리눅스 명령의 출력을 관심 있는 부분만 표시하도록 변환하는 데 유용하다. 또한 명령이 얼마나 복잡하든 상관없이 별칭 또는 스크립트로 저장하면 나중에 필요할 때마다 다시 만들 필요가 없다는 점에서도 알아 둘 필요가 있다.