

셀+앤서블

기본과정

강사소개

최 국현 :)

tang@linux.com, bluehelix@gmail.com

셀 개념

CONCEPT

KSH

ksh은 유닉스에서 많이 사용하는 셸 중 하나. 보통 다음과 같은 유닉스 시스템에서 많이 사용한다.

- **BSD 및 Unix 계열(HP, AIX, Solaris)**

리눅스도 ksh를 지원하지만, 일반적으로 잘 사용하지는 않는다.

KSH/BASH 차이점

ksh와 bash의 큰 차이점은 실제로는 거의 없다.

하지만, [Conflicts between ISO/IEC 9945 \(POSIX\) and the Linux Standard Base](#)의 내용에 충돌이 되기 때문에 서로 호환이 되지 않는 부분이 있다.

예를 들어서 LSB에서는 /usr/xpg4/bin를 사용하지 않으며 또한 솔라리스도 이러한 POSIX규격을 따르지 않는다.

리눅스 및 솔라리스 경우 2004년도 기준으로 양쪽 진영은 표준 사양을 변경하였다. 이를 통해서 리눅스 및 솔라리스 같은 유닉스와 스크립트 격차를 많이 줄었다.

Shell POSIX

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

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

그러한 이유로, echo명령어 대신 호환성을 높이기 위해서 printf명령어 사용하는 것을 권장한다.

BASH 무엇인가?

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

가끔, ssh와 셸을 혼동하는 경우가 있지만, 셸은 SSH와 같은 도구가 아님. 컴퓨터와 대화하기 위한 도구.

BASH 무엇인가?

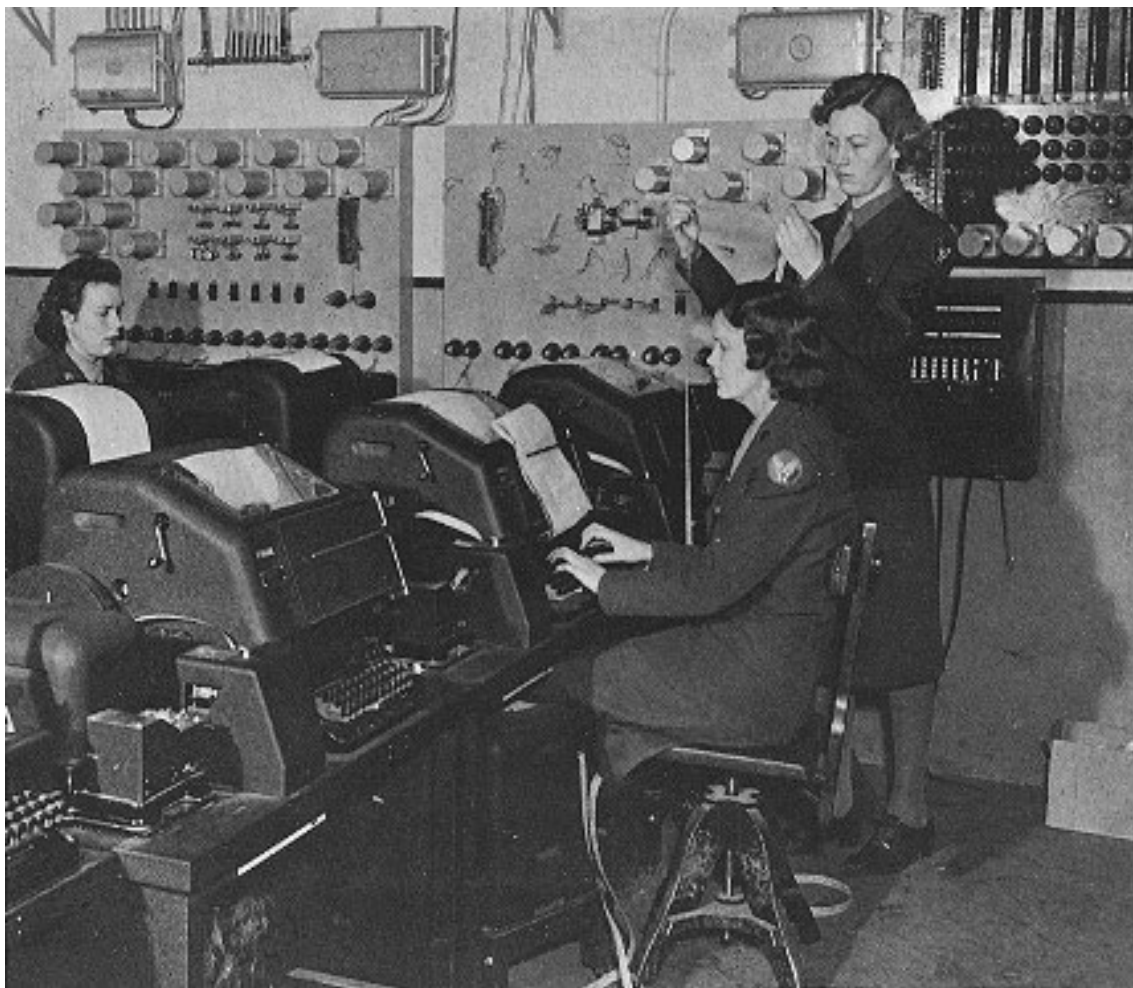
셸은 TTY가 아니다. 많은 사용자들이 오해하고 있는게 TTY == SHELL라는 오해를 가지고 있음.

PuTTY/iTerm/Xterm와 같은 터미널 도구가 아님. 터미널에 접근하기 위한 에뮬레이터 도구임.

Bash는 일반적으로 사용하는 CLI 도구와는 다름. BASH는 COMMAND LINE INTERFACE와 비슷하기는 하지만, 대화형 형태로 사용한다. 그래서, BASH셸에서 사용할 명령어의 이름을 넣어주면, BASH는 사용자가 입력한 이름의 바이너리 파일을 찾아서 실행한다.

BASH사이트 다음과 같이 BASH를 설명하고 있다.

<https://www.gnu.org/software/bash/manual/bash.txt>



THE TTY

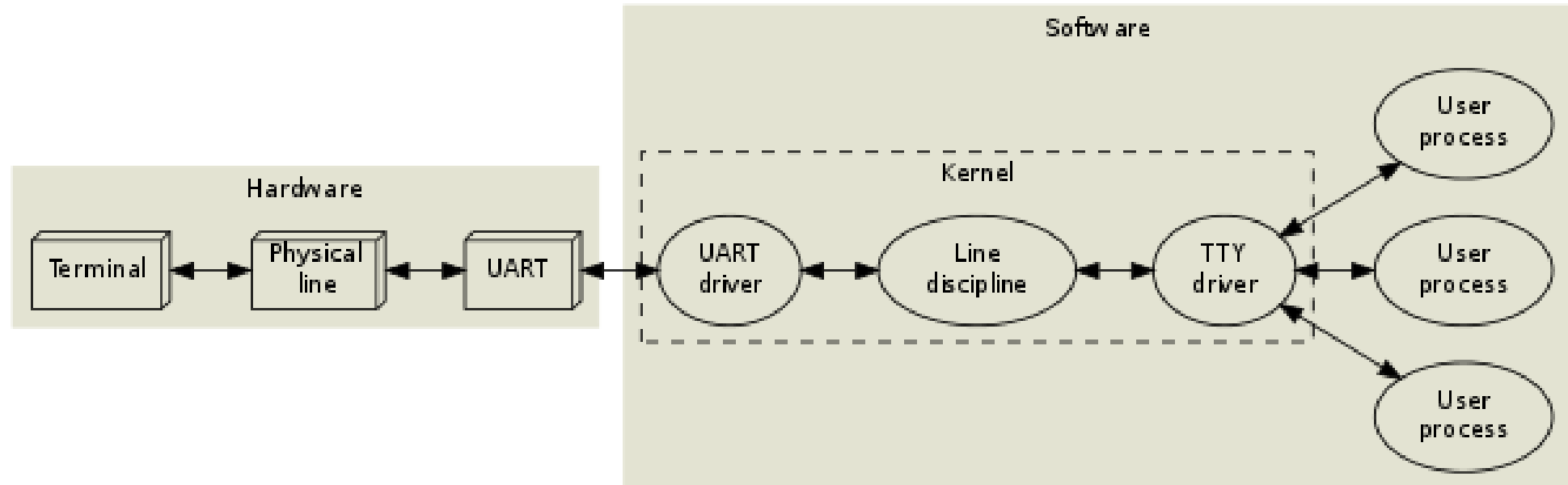
TTY는 "Teletypewriter"의 약자. 초기 유닉스 시스템은 로컬만 지원하였으며, BSD의 유닉스 프로젝트를 통해서 TCP/IP를 도입 후, 원격에서 시스템 접근이 필요.

"A Teleprinter just needed a single operator to easily convey a message. While it did not have a modern-layout keyboard, its system was later evolved by Donald Murray in 1901 to include a typewriter-like keyboard."

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

이를 통해서 원격에서 리모트에 있는 시스템에 타이핑이 가능해짐.

THE TTY



THE BASH

1.1 What is Bash?

=====

Bash is the shell, or command language interpreter, for the GNU operating system. The name is an acronym for the 'Bourne-Again SHell', a pun on Stephen Bourne, the author of the direct ancestor of the current Unix shell 'sh', which appeared in the Seventh Edition Bell Labs Research version of Unix.

<https://www.gnu.org/software/bash/manual/bash.txt>

THE BASH

Bash is largely compatible with 'sh' and incorporates useful features from the Korn shell 'ksh' and the C shell 'csh'. It is intended to be a conformant implementation of the IEEE POSIX Shell and Tools portion of the IEEE POSIX specification (IEEE Standard 1003.1). It offers functional improvements over 'sh' for both interactive and programming use.

<https://www.gnu.org/software/bash/manual/bash.txt>

THE BASH

1.2 What is a shell?

=====

At its base, a shell is simply a macro processor that executes commands.

The term macro processor means functionality where text and symbols are expanded to create larger expressions.

THE BASH

A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities.

The programming language features allow these utilities to be combined. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories such as `'/bin'`, allowing users or groups to establish custom environments to automate their common tasks.

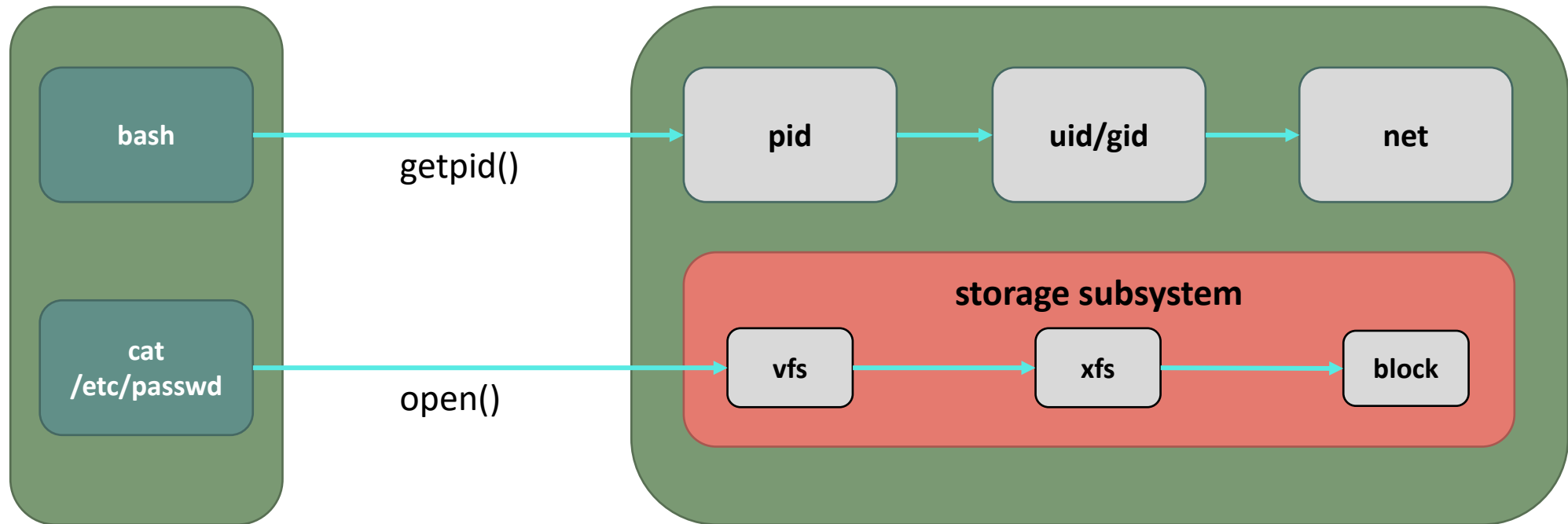
BASH 무엇인가?

거의 대다수의 리눅스는 bash가 설치되어 있음. 리눅스는 **GNU/Linux**는 **bash**가 표준이라고 생각해도 됨.

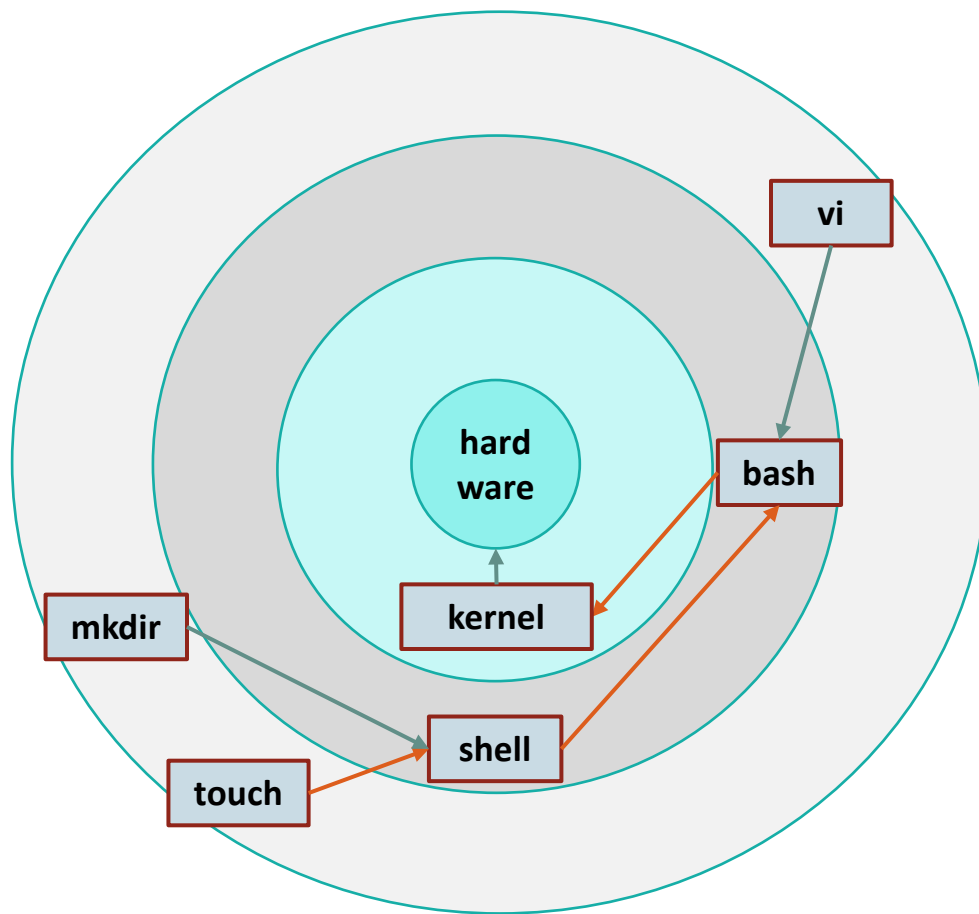
예를 들어서 **FreeBSD/NetBSD/OpenBSD** 그 외 다른 유닉스 들에서는 bash가 아닌 csh, tsh를 사용하기도 한다. 하지만 bash사용이 가능하며, 기존에 사용하였던 csh, tsh에서 사용하던 스크립트는 여전히 bash에서 거의 수정 없이 사용이 가능.

<https://www.gnu.org/software/bash/manual/bash.txt>

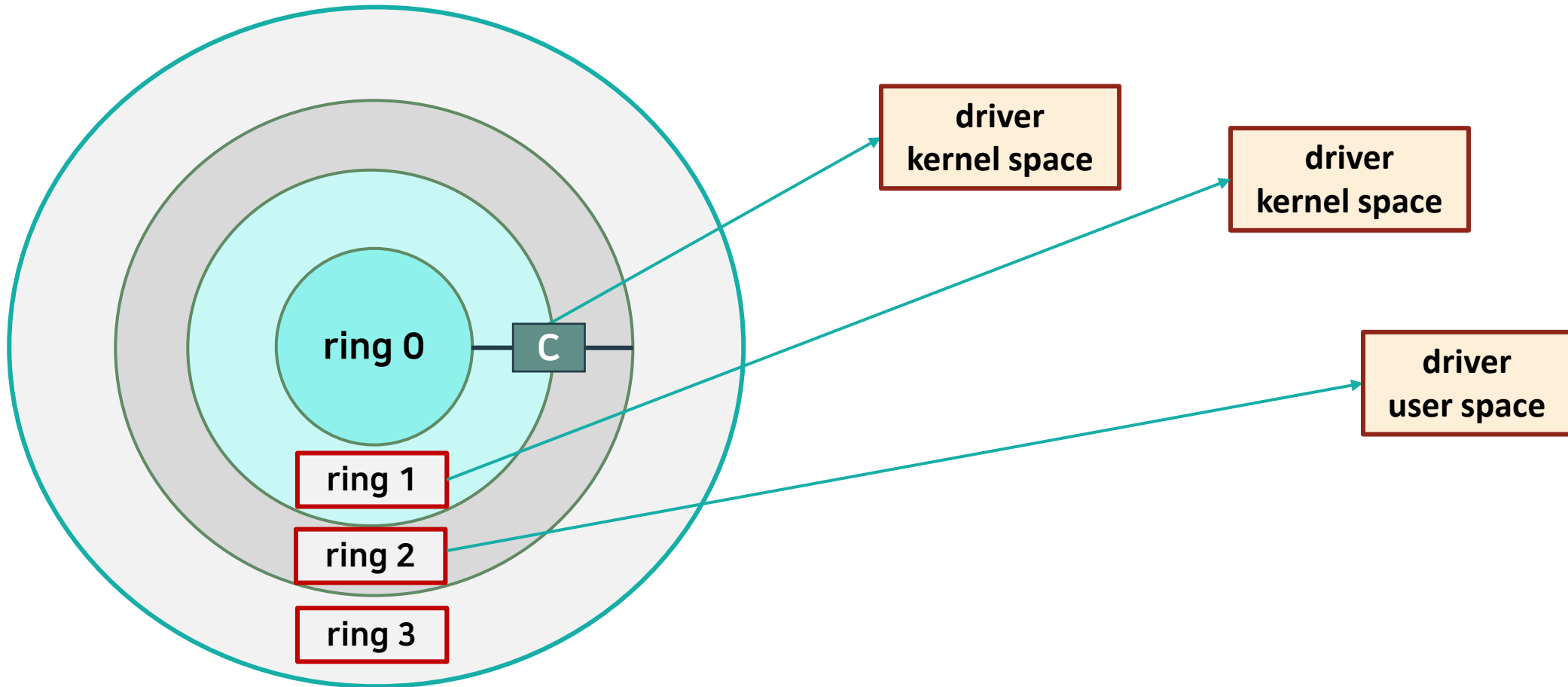
BASH 무엇인가?



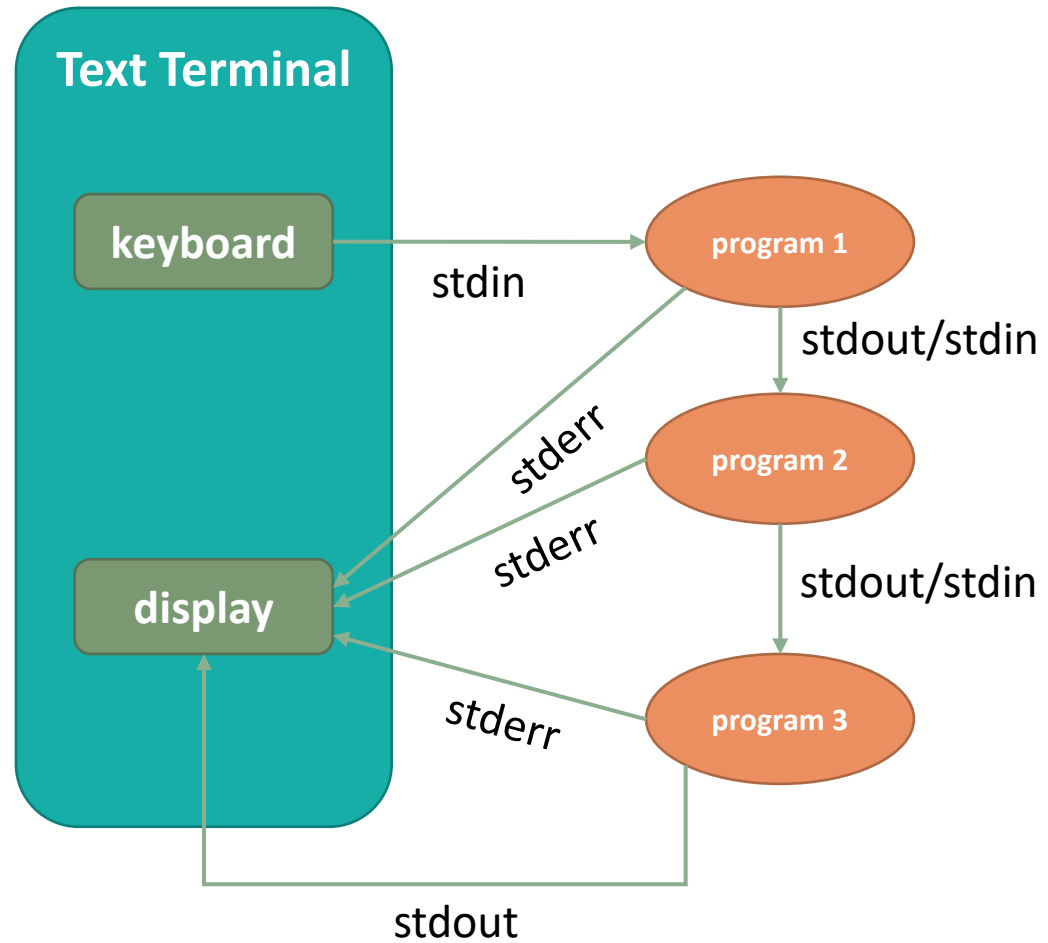
BASH 무엇인가?



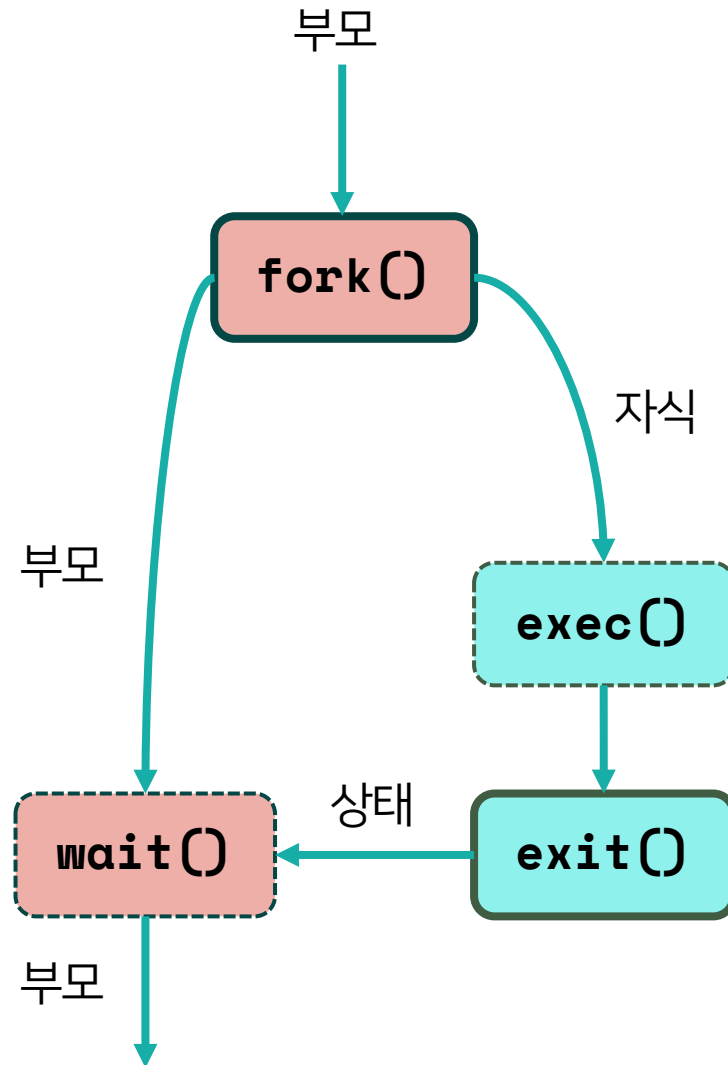
BASH 무엇인가?



BASH 무엇인가?

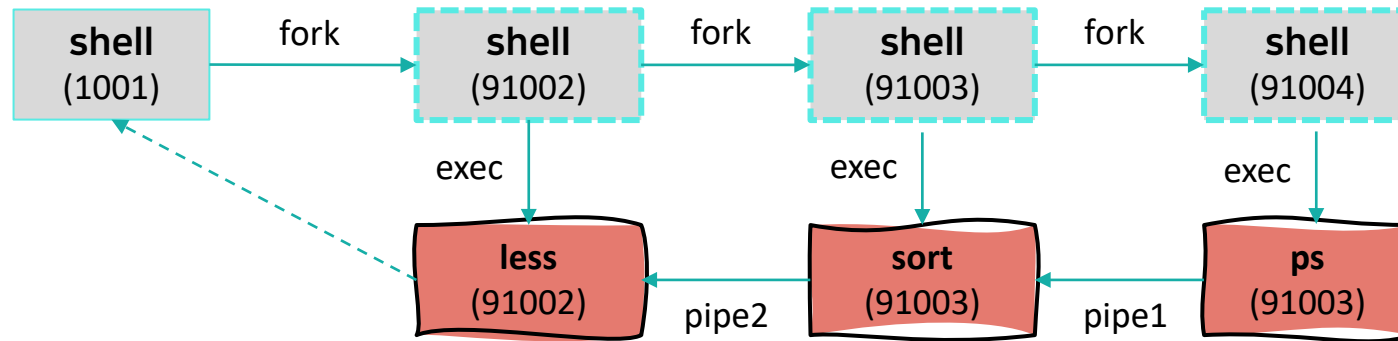


BASH 무엇인가?



BASH 무엇인가?

```
$ ps | sort | less
```



셸 기초

Bash 버전 확인하는 방법은 다음과 같다.

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

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

chsh에서 bash로 변경하는 경우 아래와 같이 변경이 가능하다.

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

셸 기초

리눅스 시스템에서 사용하는 셸 목록은 아래와 같이 확인이 가능하다.

```
$ chsh -l
```

설치가 안되어 있으면,

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

프로토타입

프로토타입

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

대화형 관리 시스템

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

오토메이션

자주 사용하는 명령어를 스크립트 기반으로 작성하여 간단하게 실행 및 수행이 가능한 방법.
이를 통해서 오토메이션 도구로 사용이 가능.

<https://opensource.com/article/19/12/automation-bash-scripts>

- **The advantages of automation with Bash shell scripts**
- Why using shell scripts is a better choice for sysadmins than compiled languages like C or C++
- Creating a set of requirements for new scripts
- Creating simple Bash shell scripts from command-line interface (CLI) programs
- Enhancing security through using the user ID (UID) running the script

프로그램 상화작용

한 개 이상의 프로그램을 쉘 스크립트를 통해서 동시에 사용 할 수 있도록 한다. 파일이나 혹은 표준 입출력을 통해서 데이터를 서로 전달하여 프로세싱 할 수 있다.

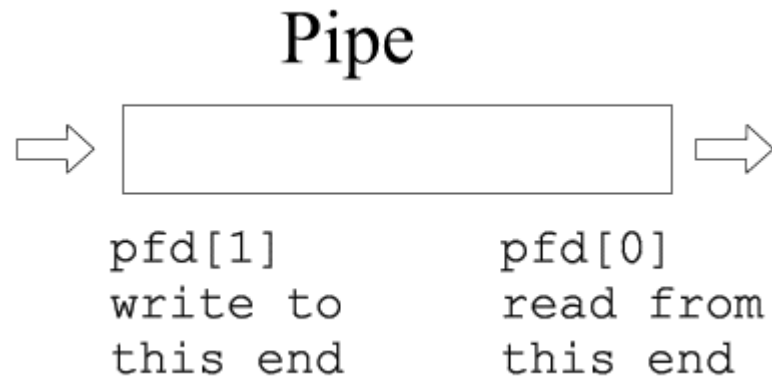
이를 보통 **파이핑(Piping)**라고 한다.

BASH가 할 수 있는 것.

필터링 그리고 입력 변환, 어떠한 프로그램은 동시에 데이터 처리에 대해서 고려하지 않고, 개발이 된 경우가 있다.

이러는 경우 bash는 다른 프로그램을 통해서 다시 데이터 재처리를 해준다. 예를 들어서 '**awk**', '**sed**'같은 프로그램들을 사용하여 원하는 형태로 출력 내용을 다시 재가공하여 전달한다.

BASH가 할 수 있는 것.



베이직 패턴 문자열 매칭

bash는 문자열 패턴 매칭 기반의 기능을 가지고 있다. 이를 통해서 파일이름, 경로 이름을 **파라미터(parameter)**로 확장이 가능하다. 이 기능을 **와일드카드(wildcard)**라고 부른다.

기본 셸 와일드 카드

1. Star or Asterisk (*)
2. Question mark (?)
3. Square brackets ([])

BASH 이동성

bash는 다양한 유닉스 및 리눅스 시스템에 패키징이 되어있다.

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

도움말

bash를 사용하다가 어려운 부분이 있으면 다음과 같은 방법으로 도움을 받을 수 있다.

내부 명령어를 통해서 사용법 확인이 필요한 경우 아래처럼 실행한다.

```
bash$ help printf
```

혹은 명령어 타입확인이 필요한 경우, 다음과 같은 명령어로 확인이 가능하다.

```
bash$ type echo
```

연습문제

간단하게 bash에서 다음과 같은 동작을 수행해서 어떠한 결과가 나오는지 확인한다.

1. 'cp', 'ls', 'type', '[' 명령어가 내부 명령어인지 외부 명령어인지 'type'명령어로 확인한다.
2. 현재 사용중인 셸 목록을 확인한다.
3. 간단하게 "pipe"를 사용하여 'ps'명령어서 현재 사용자가 사용중인 프로세스 목록을 출력한다.

연습문제

\$ type -a ls

ls is /usr/bin/ls

ls is /bin/ls

\$ type -a [

[is a shell builtin

[is /usr/bin/[

[is /bin/[

연습문제

\$ help ls

-bash: help: no help topics match `ls'. Try `help help' or `man -k ls' or `info ls'.

\$ help [

[: [arg...]

Evaluate conditional expression.

This is a synonym for the "test" builtin, but the last argument must be a literal `]', to match the opening `['.

연습문제

```
$ chsh -l
```

```
/bin/sh
```

```
/bin/bash
```

```
/usr/bin/sh
```

```
/usr/bin/bash
```

```
/usr/bin/zsh
```

```
/bin/zsh
```

```
$ ps -ef -ocmd,pid,user | grep $USER | awk '{ print $1 }' | uniq
```

```
-bash
```

```
\_
```

명령어 구조

bash

간단한 명령어

bash에서 명령어 사용 방법은 다음과 같다.

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

매우 간단하다! :

간단한 명령어

하지만, 이러한 방식으로만 사용하지 않는다. 앞에서 이야기 하였던 셸에서 사용하는 특수문자(**metacharacters**)가 있다. 예를 들어서 다음과 같다.

```
$ touch test file
```

```
$ ls -l
```

```
test
```

```
file
```

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

간단한 명령어

`$ touch report<SRV1>.txt`

`$ touch report;Final.txt`

위의 명령어를 실행을 하면 올바르게 실행이 되지 않는다.

왜?

bash 명령어 구조



설명해라 강사야!!

간단한 명령어

몇 가지 명령어를 더 테스트 해본다.

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

```
$ ls
```

```
12234likeMoney.txt
```

역시 올바르게 실행이 되지 않는다. 왜 이럴까요?

간단한 명령어

이러한 문제를 해결 하기 위해서 **쿼트(Quoting/Quote)**을 사용해야 한다. 보통 다음과 같은 방법이 있다.

'<string>'

"<string>"

\<escape>\<escape>

셸 기본

특수 문자

핸들링

특수문자 해결 방법

\$ touch report\<SRV1\>.txt

\$ touch report\;Final.txt

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



Escape Reference

A non-quoted backslash ‘\’ is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of newline. If a \newline pair appears, and the backslash itself is not quoted, the \newline is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

그러면 만약 백 슬래시(\)를 출력하는 방법?

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

```
\backslash\
```

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

그러면 만약 백 슬래시(\)를 출력하는 방법?

```
$ echo backslash\
```

```
> hello
```

```
> ygl
```

```
backslashhelloygl
```

특수문자

\$ touch 'report document'

\$ touch 'report<SRV1>.txt'

\$ touch 'report;Final.txt'

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

싱글 쿼트 줄 내림(Single Quote)

```
$ echo 'quotes
```

```
> foo
```

```
> bar'
```

```
quotes
```

```
foo
```

```
bar
```

싱글쿼트

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

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

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

어떤 내용이 출력이 될까요?

더블쿼트

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

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

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

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

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

백틱(backticks)

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

복합적 쿼트

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

```
$ $(echo echo hello)
```

위의 내용은 다음처럼 동작하게 된다.

첫번째 'echo'는 "echo hello"를 문자열로 출력하며, 이게 다시 콘솔(tty)에 전달하면서 'echo hello'라 명령어로 실행하면서 "hello"라는 메시지가 출력이 된다.

Shell Exit Values

간단하게...

\$ echo \$?

\$ true ; echo \$?

\$ false ; echo \$?

1: False

0: True

Shell Exit Values

명령어가 없는 경우...

```
$ thereisnocommand ; echo $?
```

127

Shell Exit Values

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

bash ctrl combination

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

bash ctrl combination

배시 프로세스 관리	
Ctrl + L	화면에 출력된 내용을 전부 지웁니다.
Ctrl + S	화면에 출력을 중지 합니다.
Ctrl + Z	현재 실행중인 프로세스를 일시적으로 백그라운드로 중지 합니다.
Ctrl + Q	일시적으로 중지된 프로세서를 재개 합니다.
Ctrl + C	프로세스에 SIGI시그널을 보내서 현재 실행중인 프로세스 중지 합니다.
Ctrl + D	현재 사용중인 터미널을 종료 합니다.

bash ctrl combination

배쉬 프로세스 관리	
Ctrl + R	배쉬 명령어 기록을 역순으로 검색 합니다.
Alt + P	콜론이 출력이 되면, 실행하였던 명령어 번호 혹은 명령어 일부분을 입력하면 됩니다.
Ctrl + P / Up arrow	과거에 사용했던 명령어를 조회 합니다.
Ctrl + N / Down arrow	과거 기준으로 사용하였던 명령어 조회 합니다.
Ctrl + S	화면을 잠깐 중지 합니다. 취소는 Ctrl + Q를 누르면 됩니다.

bash ctrl combination

배쉬 프로세스 관리	
Ctrl + O	Ctrl + R로 명령어 선택 후 실행 합니다.
Ctrl + G	Ctrl + G로 종료 합니다.
!!	마지막에 실행한 명령어를 수행 합니다.
!*	이전에 수행한 명령어를 실행 합니다.

bash ctrl combination

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

bash ctrl combination

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

bash ctrl combination

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

연습문제

다음 특수 문자를 화면에 올바르게 출력해보세요.

- 백틱 이용해서 사용하면 됩니다.
- 반드시 특수문자로 출력이 되어야 합니다.

This World sponsor @Bash.

I love to save \$\$USD\$\$ in my bank account!!!

변수

BASH

변수

셸에서 변수는 다음과 같이 선언한다.

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

tangshell를 변수 이름이 되며, 그 뒤에 홑 따옴표에 있는 내용은 셸 변수에서 사용하는 **자료(data)**가 된다. **BASH**에서는 이 부분을 "**string content**"라고 한다.

변수

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

```
tangshell = 'fish'
```

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

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

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

변수 목록

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

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

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

```
2022-06-26
```

```
$ declare -p today
```

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

변수 목록

'declare'에 표시되는 "--"는 특별한 속성이 아니라는 표시이다. 'declare -p'명령어를 실행하면 다음과 같은 내용을 볼 수 있다.

\$ declare -p

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

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

변수 네이밍

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

- test_var
- TestVar
- testvar
- Testvar
- _testvar
- testVar
- TestVar_

변수 네이밍

하지만 다음과 같은 이름은 허용하지 않는다.

- 1testVar
- test-var
- test.var
- test:var

변수 네이밍

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

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

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

```
grep -- teststring "$PATH"
```

결과는 다음과 같이 출력이 된다.

```
grep: command not found
```


변수 네이밍

이유는 간단하다. **\$PATH**는 시스템 변수이며, 기존에 설정된 경로 값에 덮어쓰기가 되어버린 상태다. 그래서 **PATH**라는 이름을 사용한다면 다음처럼 선언을 보통 한다.

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

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

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

변수 네이밍

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

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

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

```
export MYHOME
```

변수 취소

변수 초기화 하는 방법은 다음과 같다.

```
$ system_var =
```

그리고 초기화 된 내용을 확인 시 다음과 같은 방법으로 가능하다.

```
$ declare -p system
```

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

변수 확인

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

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

```
test_system=linux
```

```
system=windows
```

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

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

```
fi
```

변수 취소

변수를 완전히 제거하려면 unset명령어를 통해서 할 수 있다.

```
$ unset -v test_system
```

```
$ declare -p system
```

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

시스템 변수

```
$ echo $$
```

```
$ bash
```

```
$ echo $$
```

```
$ declare -p BASHPID
```

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

```
$ declare -p shellusername
```

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

시스템 변수

다시 이전 환경으로 돌아가기 위해서는 exit명령어 실행한다.

```
$ exit
```

```
$ declare -p BASHPID
```

```
$ declare -p shellusername
```

시스템 변수

만약 다른 환경에도 값을 전달하고 싶은 경우 POSIX명령어 export로 선언한다.

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

```
$ export USERNAME
```

```
$ declare -p USERNAME
```

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


시스템 변수

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

```
$ bash
```

```
$ declare -p USERNAME
```

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

```
$ exit
```

시스템 변수

환경변수는 bash에서만 사용하는 게 아니다. 모든 프로그램 및 운영체제 환경에서 적용이 되며, 다른 프로그램 언어 **Python, PHP, Awk**같은 명령어에서도 적용이 된다.

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

시스템 변수

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

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

```
$ export LC_COLLATE
```

```
$ sort words
```

```
helloworld
```

```
HelloTang
```

시스템 변수

\$ LC_COLLATE=C

\$ sort words

HelloTang

helloworld

\$ LC_COLLATE=C sort words

HelloTang

helloworld

일반변수

셸에서 변수를 확장하기 위해서 **\$1**를 사용한다. 일반적으로 **\$0**은 자신이 사용하는 명령어, **\$1**부터는 확장 파라미터라고 한다.

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

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

Hello, Tang.

일반변수

입력값 처리	설명
\$0	스크립트 자체 값. 보통 실행하는 명령어가 들어간다.
\$1	첫 번째 변수 값. 보통 명령어 및 스크립트의 첫 번째
\${10}	만약, 10개 이상의 변수가 들어오거나 혹은 한 자리 이상의 변수 순서 선언이 필요한 경우, 괄호로 표시한다.
\$#	인수에 필요한 개수를 설정 합니다.
\$*	모든 인수를 참조합니다.

일반변수

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

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

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

```
__Hello, Tang__.
```

일반변수

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

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

Hello, .

일반변수

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

```
$ set -u
```

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

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

일반변수

변수를 외부에서 받아 오기 혹은 읽어오기 할 때 내장함수인 **read**를 사용해서 사용한다. 이때 같이 사용하는 옵션은 **"-r"**를 같이 사용해서 변수에 할당한다.

```
$ read -r yourname
```

```
$ declare -p yourname
```

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

명령어 결과를 변수로

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

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

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

tang

명령어 결과를 변수로

두 개 이상의 명령어를 동시에 사용해야 되는 다음처럼 명령어를 구성한다.

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

기본값 설정

기본값 설정을 종종 필요하다.

예를 들어서 시스템에서 올바르게 기본값이 설정이 안되어 있으면 올바르게 동작이 안될 수 있다.
그래서 쉘에서는 "\${default_var:-default}"같은 형식을 제공한다.

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

혹은 다음과 같이 재선언이 가능하다.

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

LOOP/CONDITION

BASH SHELL

조건문

셸에서 조건문을 사용하기 위해서 다음과 같은 조건이 필요하다.

if

[, [[

((

case

for

while

조건문

if문을 사용하기 위해서는 다음과 같은 방법으로 구현이 가능하다.

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

먼저 중간에 있는 "&&"기호는 뒤에 두번째 명령어를 실행하는데, 이때 조건이 있다. "-q" bash 문자열이 /etc/shells에 있으면 true를 반환하면, 그 다음 명령어 printf를 실행한다. 만약, 해당 문자열이 없으면 false반환이 되기 때문에 뒤에 명령어가 실행이 되지 않는다.

조건문

이걸 if문으로 변경하면 다음처럼 된다.

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

좀 더 읽기 편하게 하기 위해서 다음처럼 구현이 가능하다.

```
if grep -q bash /etc/shells ; then
```

```
    printf 'bash is a system shell\n'
```

```
fi
```

조건문

if 명령어는 반드시 **fi**로 종료를 해야 한다.

```
$ if1.sh
```

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

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

```
fi
```

조건문

부정적인 조건은 아래처럼 사용이 가능하다. 간단한 조건문은 대 괄호로 감싸지 않아도 된다.

```
$ if2.sh
```

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

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

```
    exit 1
```

```
fi
```

조건문

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

```
$ if4.sh
```

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

```
    printf >&2 'Changed to primary directory'
```

```
elif cd /opt/mysql ; then
```

```
    printf >&2 'Changed to secondary directory'
```

```
else
```

```
    printf >&2 'Cound\' \"t find a directory!'
```

```
exit 1
```

```
fi
```

조건문

위와 같이 조건문을 사용하는 경우는 간단하게 true, false인 경우만 사용이 가능하다. 좀 더 고급적이고 복잡한 조건문을 사용하는 경우 아래와 같이 조건문을 사용해야 한다.

조건문

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

조건문

옵션	설명
[-g FILE]	파일이 존재하고 SetGID가 설정이 되어 있으면 참.
[-h FILE]	파일이 존재하고 심볼 링크면 참.
[-k FILE]	파일이 존재하고 Sticky이면 참.
[-p FILE]	파일이 존재하고 FIFO이면 참.
[-r FILE]	파일이 존재하고 읽기전용이면 참.
[-s FILE]	파일이 존재하고 파일 크기가 0보다 크면 참.

조건문

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

조건문

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

```
$ yourshell=bash
```

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

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

조건문

-e: file이라는 디렉터리나 파일이 있으면 참

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

```
$ echo $?
```

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

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

```
$ echo $?
```

if

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

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

```
fi
```

if elif else fi

'if'문에서 'else if'는 'elif'키워드로 사용이 가능하다. 또한 'else', 'elif'를 사용하는 경우 반드시 'then'키워드를 같이 사용해야 한다.

if [condition]; then

elif [other_condition]; then

else

fi

if elif else fi

```
$ vi elifelse.sh
```

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

```
second=$2
```

```
if [ "$seconds" -eq 0 ]; then
```

```
    timezone_string="Z"
```

```
elif [ "$seconds" -gt 0 ]; then
```

```
    timezone_string=$(printf "%02d:%02d" $((seconds/3600)) $(((seconds / 60) % 60)))
```

```
else
```

```
    echo "Unknown parameter"
```

```
fi
```

while/until

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

```
$ while.sh
```

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

```
COUNTER=0
```

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

```
    echo The counter is $COUNTER
```

```
    let COUNTER=COUNTER+1
```

```
done
```

while/until

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

```
$ until.sh
```

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

```
COUNTER=20
```

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

```
    echo COUNTER $COUNTER
```

```
    let COUNTER-=1
```

```
done
```

for

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

\$ for.sh

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

```
for i in $( ls ); do
```

```
    echo item: $i
```

```
done
```


루프 및 조건문

-d: 디렉터리 이름이 dir인 경우

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

```
$ echo $?
```

-s: 파일이 존재하며 파일이 비어 있지 않는 경우

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

```
$ echo $?
```

test, [, [[

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

쉘 스크립트의 키워드와 다르다.

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

if문은 grep, cd 명령어처럼 사용이 가능하다.

```
yourshell=bash
```

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

```
    printf 'Correct!\n'
```

```
fi
```

test, [, [[

이전 bash는 "["를 사용하였지만, bash 4.0이후부터는 "[["으로 변경되었다. 이 문법은 bash에서 특별한 문법이며 일반적인 빌트인 명령어가 아니다.

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

```
yourshell=bash
```

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

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

```
fi
```

test, [, [[

"[, "["하고 제일 큰 차이점은 다음과 같다.

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

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

"[" 경우에는 변수를 쌍 따옴표로 감싸야 한다.

test, [, [[

구체적인 차이는 다음과 같다.

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

이 부분은 다음처럼 사용이 가능하다.

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

test, [, [[

yourshell=\$SHELL

[[\$yourshell = "\$yourshell"]]

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

["\$yourshell" = "\$yourshell"]

"["방식과 비슷하지만, "["에서 크게 문제 없이 동작한다.

test, [, [[

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

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

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

test, [, [[

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

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

위의 명령어를 실행하면 마지막 문자가 sh로 끝나는 찾아서 화면에 출력한다.

test, [, [[

복잡한 오퍼레이션 사용이 필요한 경우 좀 더 간단하게 표현이 가능하다.

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

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

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

산술조건(Arithmetic condition)

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

["\$a" -eq "\$b"]: 같다

["\$a" -ne "\$b"]: 같지 않다

["\$a" -lt "\$b"]: 작거나

["\$a" -le "\$b"]: 작거나 혹은 같거나

["\$a" -gt "\$b"]: 크거나

["\$a" -ge "\$b"]: 크거나 혹은 같거나

산술조건(Arithmetic condition)

아래와 같이 100까지 카운터를 if문으로 제어가 가능하다.

```
count=100
```

```
if [ $count -eq 100 ]
```

```
then
```

```
    echo "Count is 100"
```

```
fi
```

산술조건(Arithmetic condition)

모든 산술조건이 올바르게 맞으면 보통 다음처럼 값을 반환한다.

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

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

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

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

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

활용하기 1

process utils

background

ps

- 1 UNIX options, which may be grouped and must be preceded by a dash.
- 2 BSD options, which may be grouped and must not be used with a dash.
- 3 GNU long options, which are preceded by two dashes.

ps

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

ps -ef

위의 명령어는 AIX(Unix)에서 사용하던 유닉스 스타일 명령어

ps aux

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

ps

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

ps aux or ps au

위의 명령어는 BSD스타일 명령어.

ps

ps aux or ps au

- "a" 옵션은 BSD 스타일의 "only yourself" 옵션
- "u" 옵션은 EUID(effective user ID)
- "x" 옵션은 x BSD 스타일의 "must have a tty" 옵션

```
[root@localhost ~]# ps -aux
USER      PID  %CPU  %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1   0.0   0.7 128144   6828 ?        Ss   Sep11   0:02 /usr/lib/systemd/systemd
root         2   0.0   0.0      0      0 ?        S    Sep11   0:00 [kthreadd]
root         4   0.0   0.0      0      0 ?        S<   Sep11   0:00 [kworker/0:0H]
root         5   0.0   0.0      0      0 ?        S    Sep11   0:00 [kworker/u32:0]
root         6   0.0   0.0      0      0 ?        S    Sep11   0:00 [ksoftirqd/0]
root         7   0.0   0.0      0      0 ?        S    Sep11   0:00 [migration/0]
root         8   0.0   0.0      0      0 ?        S    Sep11   0:00 [rcu_bh]
root         9   0.0   0.0      0      0 ?        R    Sep11   0:36 [rcu_sched]
root        10   0.0   0.0      0      0 ?        S<   Sep11   0:00 [lru-add-drain]
root        11   0.0   0.0      0      0 ?        S    Sep11   0:00 [watchdog/0]
```

ps

ps -ef

- "-f" 옵션은 "full-formatting list"
- "-e" 옵션은 "all processes"

위의 명령어는 Unix(AIX)스타일의 명령어

```
[root@localhost ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1         0  0 Sep11 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --
root           2         0  0 Sep11 ?        00:00:00 [kthreadd]
root           4         2  0 Sep11 ?        00:00:00 [kworker/0:0H]
root           5         2  0 Sep11 ?        00:00:00 [kworker/u32:0]
root           6         2  0 Sep11 ?        00:00:00 [ksoftirqd/0]
root           7         2  0 Sep11 ?        00:00:00 [migration/0]
root           8         2  0 Sep11 ?        00:00:00 [rcu_bh]
root           9         2  0 Sep11 ?        00:00:36 [rcu_sched]
root          10         2  0 Sep11 ?        00:00:00 [lru-add-drain]
```

ps

ps -x

```
[tang@www ~]$ ps -x
```

PID	TTY	STAT	TIME	COMMAND
69492	?	Ss	0:00	/usr/lib/systemd/systemd --user
69496	?	S	0:00	(sd-pam)
79289	?	Ss	0:05	tmux
79290	pts/15	Ss	0:00	-sh
79320	pts/21	Ss	0:00	-sh
79350	pts/22	Ss	0:00	-sh
80825	pts/15	S+	0:00	ssh root@192.168.90.171
80828	pts/15	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
80861	pts/21	S+	0:00	ssh root@192.168.90.3
80864	pts/21	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
80865	pts/22	S+	0:00	ssh root@192.168.90.168
80868	pts/22	S+	0:00	/usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.168

ps

ps -fu or -fU <UID> <UNAME>

ps -fU tang

```
[tang@www ~]$ ps -fU tang
UID      PID      PPID  C  STIME TTY          TIME CMD
tang     69492        1  0  Sep06 ?        00:00:00 /usr/lib/systemd/systemd --user
tang     69496     69492  0  Sep06 ?        00:00:00 (sd-pam)
tang     79289        1  0  Sep06 ?        00:00:05 tmux
tang     79290     79289  0  Sep06 pts/15    00:00:00 -sh
tang     79320     79289  0  Sep06 pts/21    00:00:00 -sh
tang     79350     79289  0  Sep06 pts/22    00:00:00 -sh
tang     80825     79290  0  Sep07 pts/15    00:00:00 ssh root@192.168.90.171
tang     80828     80825  0  Sep07 pts/15    00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.171
tang     80861     79320  0  Sep07 pts/21    00:00:00 ssh root@192.168.90.3
tang     80864     80861  0  Sep07 pts/21    00:00:00 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.3
```

ps

ps -U root -u root

"-u" 영향 받는 사용자 아이디(RUID)

"-U" 실제 사용자 아이디(EUID)

```
[tang@www ~]$ ps -u tang -U tang
  PID TTY          TIME CMD
 69492 ?            00:00:00 systemd
 69496 ?            00:00:00 (sd-pam)
 79289 ?            00:00:05 tmux: server
 79290 pts/15      00:00:00 sh
 79320 pts/21      00:00:00 sh
 79350 pts/22      00:00:00 sh
 80825 pts/15      00:00:00 ssh
 80828 pts/15      00:00:00 sss_ssh_knownho
```

ps

ps -Fg tang

ps -fG 1000

```
[tang@www ~]$ ps -Fg tang
UID      PID     PPID  C   SZ   RSS  PSR STIME TTY
tang    69492       1   0 22388 9528   4 Sep06 ?
tang    69496   69492   0 81902 3320   0 Sep06 ?
tang    79289       1   0  7246 4528   5 Sep06 ?
tang    79290   79289   0  6929 5140   3 Sep06 pts/15
tang    79320   79289   0  6929 5012   5 Sep06 pts/21
tang    79350   79289   0  6929 4988   3 Sep06 pts/22
tang    80825   79290   0 15448 6812   6 Sep07 pts/15
tang    80828   80825   0 24797 5588   6 Sep07 pts/15
tang    80861   79320   0 15448 6920   3 Sep07 pts/21
```

ps

ps -fp

```
[tang@www ~]$ sudo ps -fp 1
UID      PID    PPID  C  STIME TTY          TIME CMD
root         1        0  0 Sep03 ?        00:01:30 /usr/li
```

ps

ps -f --ppid 1

```
[tang@www ~]$ sudo ps -f --ppid 1
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	751	1	0	Sep03	?	00:00:11	/usr/lib/systemd/systemd-journald
root	799	1	0	Sep03	?	00:00:06	/usr/lib/systemd/systemd-udevd
rpc	899	1	0	Sep03	?	00:00:03	/usr/bin/rpcbind -w -f
root	908	1	0	Sep03	?	00:00:04	/sbin/auditd

kill/killall/pkill

- 1:** 프로세스를 재시작 한다. 모든 프로세스에서 적용되는 신호는 아니다.
- 9:** 현재 사용중인 프로세스를 강제로 메모리에서 종료한다. 이는 보호가 되지 않기 때문에, 프로세스는 종료하면서 모든 데이터를 소멸한다.
- 15:** 정상적으로 프로세스를 종료한다.

pgrep/pkill

'pgrep'은 프로세스의 이름 혹은 다른 속성을 확인하여 시그널 전달.

```
$ pkill <signal> <process_name>
```

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

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

pgrep/pkill

```
$ pgrep -c ssh
```

```
10
```

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

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

pgrep/pkill

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

```
80825
```

```
80828
```

```
80861
```

```
80864
```

```
80865
```

```
80868
```

```
131706
```

pgrep/pkill

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

```
12
```

```
$ pgrep -l ssh
```

```
1363 sssd_ssh
```

```
30317 sshd
```

```
79439 ssh
```

```
79442 sss_ssh_knownho
```

```
79459 ssh
```

```
79462 sss_ssh_knownho
```

pgrep/pkill

\$ pgrep -a ssh

79439 ssh root@192.168.90.178

79442 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.178

79459 ssh root@192.168.90.187

79462 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.187

80023 ssh root@192.168.90.183

80026 /usr/bin/sss_ssh_knownhostsproxy -p 22 192.168.90.183

131706 sshd: tang@pts/0

pkill/pgrep

ssh와 일치하지 않는 프로세스만 출력

```
$ pgrep -v ssh
```

정확하게 sshd이름과 일치하는 프로세스만 출력

```
$ pgrep -x sshd
```

최근에 생성된 ssh process만 출력

```
$ pgrep -n ssh
```

이전에 생성된 ssh process만 출력

```
$ pgrep -o ssh
```

pgrep/pkill

'ssh'로 시작하는 프로세스를 종료한다.

```
$ pkill '^ssh$'
```

'ping 8.8.8.8'으로 시작하는 명령어를 종료한다.

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

사용자 'mark'를 제거 합니다.

```
$ pkill -u mark
```


pgrep/pkill

사용자 "mark", "gnome"를 시그널 "-9"으로 종료한다.

```
$ pkill -u mark gnome -9
```

최근에 실행한 프로세스 'screen'를 종료 한다.

```
$ pkill -9 -n screen
```

pstree

이 명령어는 프로세스 종속성(혹은 의존성)을 트리형태로 화면에 출력한다.

\$ pstree -Ap

-A: 아스키 문자 기반으로 화면에 출력 합니다. 일반적으로 'pstree'사용시 아스키 문자로 출력한다.

-P: 프로세스의 PID를 화면에 같이 출력한다.

kill and killall

kill

PID로 프로세스 종료. 기본값은 **15번(TERM)**으로 되어 있다.

killall

kill과 동일하게 사용하는 명령어. 차이점은 **PID**가 아닌 **Process Name**으로 관리한다.

명령어 연속 실행 시 오류 중지

여러 명령어를 수행하는 방법은 두 가지가 있다.

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

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

첫 번째 방법은 오류가 발생하여도 다음명령어가 실행이 되며, 두 번째 방법은 첫 번째 명령어가 실행 중 문제가 발생하면 중지.

포그라운드/백그라운드

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

\$ sleep 10

\$ sleep 10 &

포그라운드/백그라운드

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

\$ bg

연습문제

1. 'sleep'명령어를 약 1000초 동안 백그라운드로 동작하게 한다.
2. 가동중인 백 그라운드 프로세스를 bg로 확인한다.
3. 백 그라운드로 동작중인 프로세스를 다시 포그라운드로 올린다.
4. 사용자 procuser를 추가한 다음에 procuser로 로그인 한다.
5. procuser에서 'sleep 1000'명령어를 실행한다.
6. 'ps'명령어로 procuser의 프로세스 실행을 확인한다.
7. 확인이 완료가 되면 root에서 'pkill'명령어로 procuser를 종료한다.

활용하기2

bash

bash 기본 명령어

셸에서 기본적으로 제공하는 내부 명령어(**internal command**)가 있다. 이 명령어를 통해서 좀 더 편하게 셸 환경을 꾸밀 수 있다.

빌트인 명령어는 다음과 같으며, 실제로 몇몇 명령어는 외부 파일로 구현이 되어 있기도 하다. 그러니 외부에(명령어가) 있어도 너무 신경 쓰지 말자.

빌트인 명령어

빌드인 명령어는 아래와 같다.

alias, bg, bind, break, builtin, case, cd, command, compgen, complete, continue, declare, dirs, disown, echo, enable, eval, exec, exit, export, fc, fg, getopts, hash, help, history, if, jobs, kill, let, local, logout, popd, printf, pushd, pwd, read, readonly, return, set, shift, shopt, source, suspend, test, times, trap, type, typeset, ulimit, umask, unalias, unset, until, wait, while.

빌트인 명령어

내부 명령어는 외부에 별도의 명령어 없이 실행이 가능한 내장형 명령어. 대표적인 명령어는 다음과 같다.

echo, type, source

아래서 자세한 내용은 뒤에서 설명.

런타임 명령어

런타임 명령어(runtime)는 기본적으로 셸 스크립트(bash)명령어 구성이 되어 있다.

여기 말하는 셸 스크립트 명령어는 'type', 'alias'같은 명령어이며, 특정 시스템 명령어 'ls', 'cp'같은 명령어는 'alias'로 재구성이 되어 있기도 하다. 몇몇 특정 명령어는 **shell function()**로 구성이 되어 자체적으로 실행은 불가능하며, 셸을 통한 조합 실행이 가능하다.

시스템 명령어

시스템 명령어는 파일 시스템에 독립적으로 구성이 되어 있는 프로그램. 어떠한 패키지가 설치가 되어 있느냐에 명령어가 조금 다를 수 있다. 대표적인 명령어는 다음과 같다.

grep, ping, rm, ls, nmcli

셸 대표 명령어

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

echo: 문자열 출력

printf: echo와 비슷하지만 포매팅 형식으로 출력

pwd: 셸의 현재 위치 출력

cd: 셸 위치 변경

set: 현재 셸에 설정된 값을 출력

declare: 변수나 함수(function)를 관리하는 명령어

test, [, [[: 정규식 구성, 대괄호로 구성이 된다.

대표 명령어

ls: 디렉터리 및 파일 출력

mv: 파일 이동 및 이름 변경

cp: 파일 복사

rm: 파일 및 디렉터리 제거, 다른 명령어로 rmdir이 있다.

grep: 문자열 검색하는 정규식 도구

cut: 행 단위로 자르는 명령어

wc: 줄 및 문자열 카운팅 도구

find: 파일이나 혹은 디렉터리를 특정 조건으로 찾는 도구

sort, uniq: 정렬 및 중복제거 하는 도구

type

bash에서 명령어 사용 방법이 궁금한 경우 아래와 같이 사용이 가능하다.

```
bash$ help type
```

type명령어를 사용하면 다음과 같이 출력한다.

```
bash $ type cp
```

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

```
bash $ type ls
```

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

```
bash $ help cp
```

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

```
bash $ type for
```

```
for is a shell keyword
```


type

function()를 type으로 확인하기.

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

```
bash $ type myfunc          ## myfunc 확인
```

```
myfunc is a function
```

```
myfunc()
```

```
{
```

```
:
```

```
}
```

```
bash $ alias myalias=:      ## alias 선언
```

```
bash $ type myalias         ## 선언된 alias내용 확인
```

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

type

간단하게 명령어가 내부/외부 명령어 확인이 필요할 때, 아래처럼 실행한다.

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

```
true is a shell builtin
```

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

echo

간단하게 메시지 출력을 'echo' 명령어로 한다.

```
bash $ echo hello
```

hello

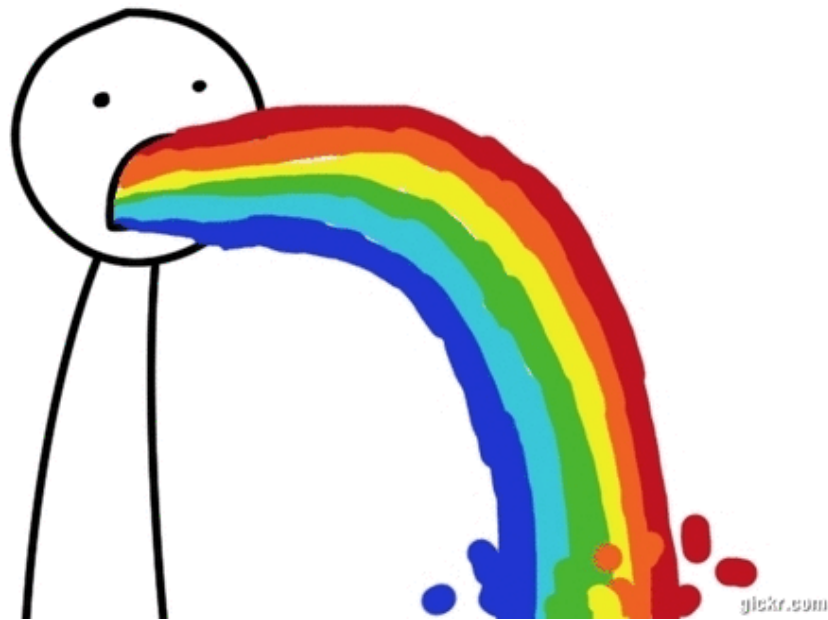
```
bash $ echo 'Hello, "$USER"'
```

Hello, bashuser!

기본 명령어

dommie95

잠깐 분명 명령어인데 왜 자꾸 빌트인이라고 하시나요?



printf

'echo'명령어 대신 'printf'명령어 사용이 가능하다.

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

Hello!

이 명령어는 C의 "printf"와 흡사한 포매팅 구조를 가지고 있다.

echo vs printf

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

echo

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

printf

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

echo vs printf

하지만, 쉘 버전이 업데이트 및 업그레이드가 되면서 현재는 'echo', 'printf'의 실행 속도에 큰 차이가 없지만 일반적으로 'printf'명령어가 더 빠르다. 성능 테스트 하기 위해서 다음과 같이 명령어를 실행한다.

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

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

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

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

echo vs printf

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

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

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

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


echo vs printf

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

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

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

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

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

빌트인?

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

type echo

echo is a shell builtin

whereis alias

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

빌트인?

```
# file /usr/bin/alias
```

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

```
# cat /usr/bin/alias
```

```
#!/usr/bin/sh
```

```
builtin alias "$@"
```

빌트인?

```
# file /usr/bin/alias
```

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

```
# cat /usr/bin/alias
```

```
#!/usr/bin/sh
```

```
builtin alias "$@"
```

printf

대표적인 빌트인 명령어 사용 예제는 다음과 같다.

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

```
-n
```

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

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

```
-n
```

printf

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

```
foo
```

```
bar
```

```
baz
```

pwd

현재 작업하는 위치를 확인 시 사용하는 명령어.

\$pwd

/home/bash/working

\$ ls secret/report.txt

\$ ls report_review.txt

\$ ls nonexistent

tildes(~)

경로 관련된 "~"(틸드, tild)특수 문자.

이 문자를 통해서 홈 디렉터리로 이동이 가능하다. 틸드 **path**이외 다른 특수 문자 '.', '..'등 있지만, 보통 홈 디렉터리 이동 시 틸드를 더 선호 한다.

```
$ echo ~
```

```
$ echo ~/secret
```

```
# echo ~
```

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


tildes

만약 존재하지 않는 경우 그냥 문자열로 출력이 된다.

```
$ echo ~nouser
```

```
~nouser
```

tildes

특수 문자로 출력하고 싶은 경우 아래처럼 실행한다.

```
$ echo ~bash
```

```
/home/bash
```

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

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

cd

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

```
$ pwd
```

```
/home/bash
```

```
$ cd /tmp
```

```
$ pwd
```

```
/tmp
```

cd

앞에서 언급하였던 .. 혹은 .를 통해서 디렉터리 이동이 가능함.

```
$ cd .
```

```
$ cd ..
```

```
$ pwd
```

cd

조금 더 자세히 ".", ".."에 대해서 알고 싶으면 'stat'명령어로 확인해보면 디렉터리 관계 확인이 가능하다.

```
$ stat .
```

```
$ stat ..
```

pushd/popd

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

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

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

```
$ pushd /opt/
```

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

pushd/popd

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

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

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

```
$ pushd +2
```

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

pushd/popd

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

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

```
$ pushd /opt/
```

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

```
$ popd
```


ls

\$ ls

adir bdir cdir

\$ ls -a

. .. .profile .bash_profile Downloads

ls

\$ ls -al

drwxrwxr-x. 4 tang tang 4096 May 24 12:19 kubectl-plugins

drwxrwxr-x. 10 tang tang 249 May 24 12:16 kubectx

drwxrwxr-x. 9 tang tang 251 May 24 12:15 kubelogin

drwxrwxr-x. 3 tang tang 80 May 24 12:26 kubergui

ls

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

\$ ls -l

\$ grep pattern -- *

ls

이걸 좀 더 보기 좋게 표현하기 위해서 다음과 같이 시도한다.

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

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

ls

위의 상황은 'ls'명령어 사용이 불가능한 경우이다.

'ls'명령어 사용이 가능한 경우에는 굳이 위와 같이 사용할 필요가 없다.

mv

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

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

```
$ mv file1 file2
```

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

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

```
$ touch skt.txt
```

```
$ mv skt.txt /tmp
```

```
$ touch skt.txt
```

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

mv

대화형으로 사용을 원하는 경우 아래와 같은 명령어로 사용이 가능하다.

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

mv

```
$ nano mv1.sh
```

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

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

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

```
    mv old_user new_user
```

```
fi
```

```
mv new_user user
```


cp

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

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

```
$ ls
```

```
dir
```

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

cp

```
$ cp -R
```

```
$ cp -r
```

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

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

rm, rmdir

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

```
$ rm testfile
```

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

```
$ rmdir testdir/
```

Directory not empty

grep

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

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

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

grep

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

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

grep

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

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

grep

중복된 출력 결과를 다음과 같은 명령어로 정리가 가능하다.

```
$ uniq
```

```
$ sort
```

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

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

```
$ echo $?
```

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

```
$ echo $?
```

grep

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

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

5

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

복합적으로 문자열을 확인이 필요한 경우는 다음과 같은 명령어로 확인이 가능하다.

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


grep

'-F' 옵션을 통해서 문자열 검색이 가능하다.

매뉴얼 설명은 다음처럼 나와있다.

Interpret PATTERNS as fixed strings, not regular expressions.

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

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

'^#' 옵션은 주석 문자인 #를 제외하고 화면에 출력.

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

grep

조금 더 복잡하게 특정 문자만 카운팅이 필요한 경우 다음과 같이 실행한다

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

10

-c: 카운팅

-F: 정규식 기호 문자열 취급

-v: 예외

-e: 복수 조건(정규식)

grep

'grep'명령어는 "OR"조건은 지원하지만, "AND"조건은 지원하지 않는다.

AND조건으로 문자열을 찾기 위해서는 다음과 같은 방법을 권장한다.

```
$ grep -e <STRING> -e <STRING>
```

혹은

```
$ grep <STRING> | grep <STRING>
```

cut

'cut'명령어는 말 그래도 자르는 명령어.

특정 필드나 문자열을 자르고 싶은 경우 cut명령어를 통해서 사용이 가능하다.

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

cut

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

root

bin

daemon

adm

lp

sync

shutdown

cut

만약, 맨 끝 필드를 출력하고 싶은 경우 아래처럼 실행한다.

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

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

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

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

WC

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

```
$ wc .bash_history
```

```
200 437 3373 .bash_history
```

위의 출력된 내용은 200줄, 437줄 그리고 총 3373바이트.

여기에서 특정 값만 출력을 원하는 경우 '-c', '-w', '-l'같은 옵션을 사용해서 출력이 가능하다.

'-l'옵션은 엄연히 따지고 보면 newline개수만 출력한다.

WC

"-c" 옵션 경우에는 바이트를 출력한다.

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

```
3373 .bash_history
```


WC

"-m"총 문자 개수를 카운팅 하기 위해서는 "-m"옵션을 사용한다.

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

```
3373 .bash_history
```

WC

\$ wc -m -c Korean

35 93 test

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

du

'wc'명령어로 파일의 크기를 알 수 있다.

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

하지만 전체적인 크기에 대해서는 확인이 불가능 하기 때문에 'du'명령어를 사용하면 된다.

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

```
1124  /bin/bash
```

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

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

find

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

기본적인 동작 방식은 다음과 같다.

```
$ find ~/
```

출력만 하고 싶은 경우 다음처럼 옵션을 지정한다.

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

find

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

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

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

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

find

test라는 문자열을 가지고 있는 파일 및 디렉터리

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

fish라는 문자열 가지고 있는 파일만 출력

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

수정한지 3일이 지난 파일

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

5일전에 수정한 파일

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

find

이름에 beef라는 문자열이 없는 것만 검색

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

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

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

find

find를 사용하면, 각각 결과마다 명령어를 같이 실행이 가능하다.

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

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

{}: 이 부분에 find에 찾을 출력 결과물로 대체

\;: 명령어 종료 알림

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

find

위 명령어를 나누면 다음과 같이 두개 구조로 나누어진다.

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

인자 끝
(명령어)

치환

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

/usr/share/doc/vim/map.vim

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

find

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

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

연속처리

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

첫번째 결과

두번째 결과

find -exec +

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

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

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

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

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

find with cp

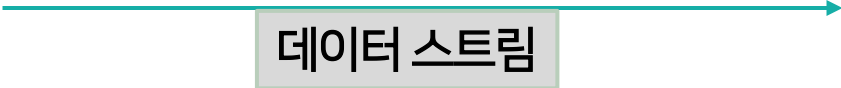
'find' 명령어로 파일 복사 시, 다음과 같이 활용한다.

```
# mkdir -p /tmp/md_files/
```

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

xargs

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



데이터 스트림

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

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

xargs

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

'find -print0' 혹은 'xargs -0'

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

xargs

만약, 출력되는 결과에 null-byte로 종료되는 경우가 있다.

올바르게 동작이 안될 수 있기 때문에 find에서 xargs를 사용시 -print0이라는 옵션을 사용한다.


'find -print0' 혹은 'xargs -0'

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

find + cp + xargs

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

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



표준 출력으로 전달 받은 입력 내용중 디렉터리 파싱

find+

추가적으로 'find'의 -exec 및 '{}' +에 대해서 추가적인 설명. 앞에서 말한 옵션 "-exec {} \", "-exec {} +", "| xargs" 총 3가지로 사용이 가능하다.

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

find+

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

find+

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

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

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

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

find+

2번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다.

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

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

find+

3번 경우에는 실행이 되는 1:N 인자 값으로 실행이 된다. 삭제 대상 파일이 test1.txt test2.txt파일이라고 하면 다음처럼 동작한다.

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

하지만 3번 경우에는 문제가 있다.

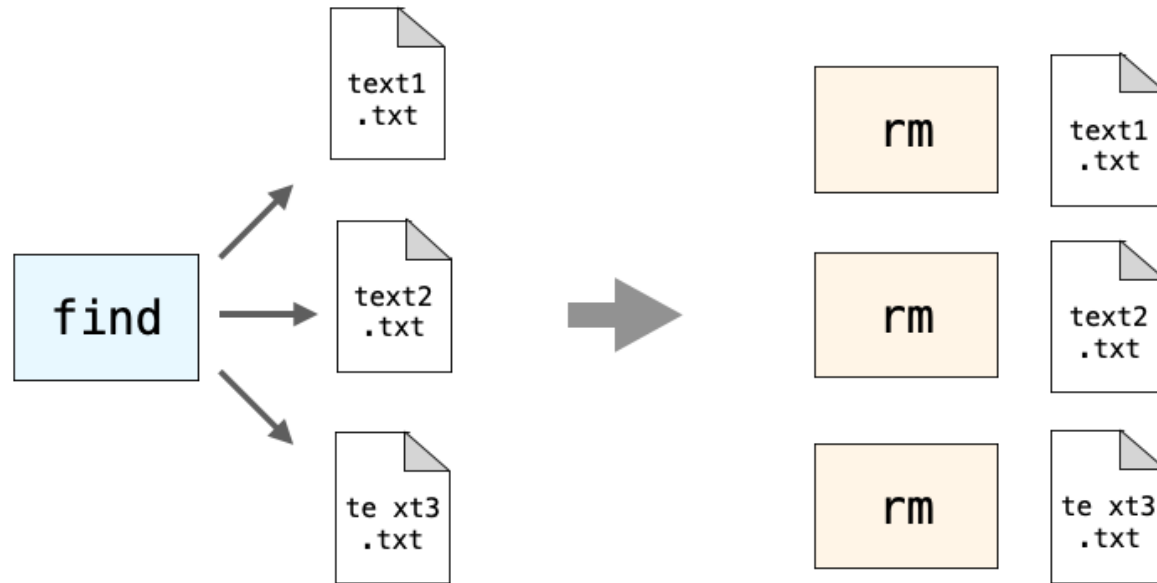
find+

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

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

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

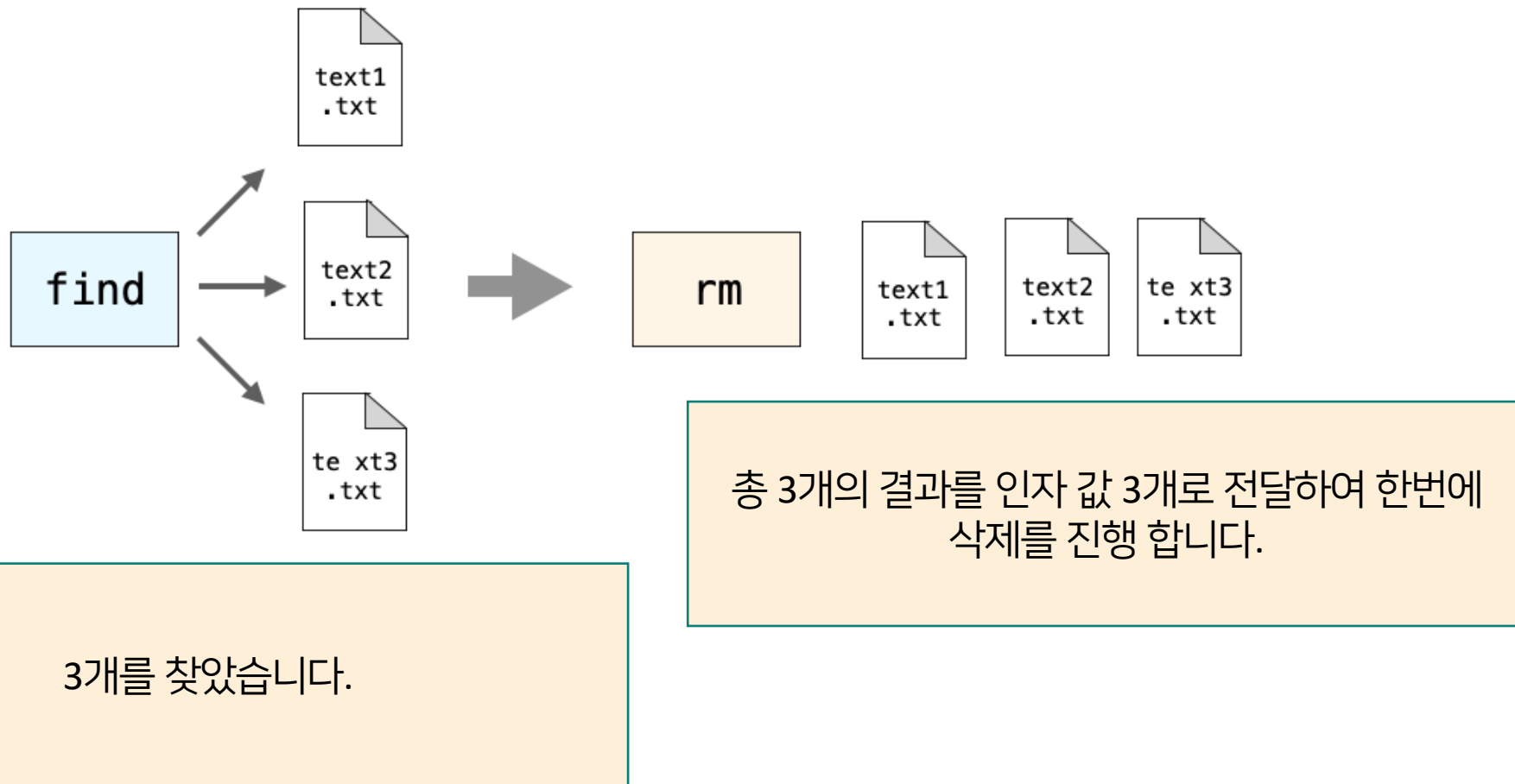
find+



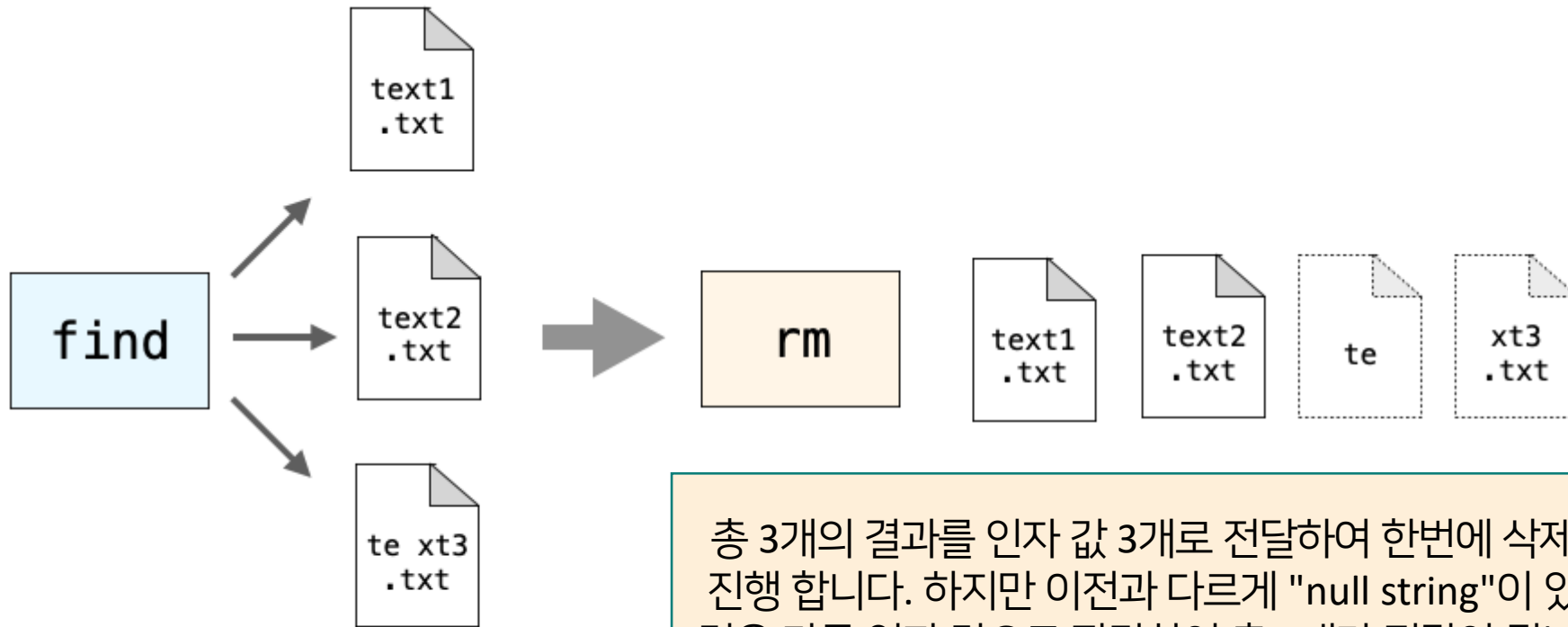
3개를 찾았습니다.

find에서 대상을 찾을 때마다 1번씩 명령어 수행,
총 3번 실행을 합니다.

find+



find+



3개를 찾았습니다.

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

find+

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



\0

위의 test1.txt\0 "null string"이 존재 합니다.

引数 1 : ./test1.txt

引数 2 : ./te st2.txt

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

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

find+

■ -pr

-print0 옵션을 사용하기 전

find



./test1.txt
./test2.txt

■ -p

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

find



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

Null String vs Empty String

positive value



1



0



negative value



Infinity



NaN



null



undefined



sort

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

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

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

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

sort

또한 같이 사용하는 옵션은 다음과 같다.

-n: 숫자가 알파벳 순서보다 먼저 출력이 된다.

-r: 역 정렬, 맨 마지막 결과가 맨 처음에 출력

uniq

```
$ cat ipaddress.txt
```

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

연습문제

1. 'printf'명령어를 통해서 "Hello World" 문자열을 출력 합니다.
2. 'echo'명령어를 통해서 "Hello World2"문자열을 출력 합니다.
3. 'echo \$USER', echo '\$HOME', 'echo ~'명령어를 수행하여 차이점을 확인 합니다.
4. 'pushd'명령어를 사용하여 다음처럼 디렉터리를 이동한다.
 - -> /tmp
 - -> /var/tmp/
 - -> /usr/share/doc
 - -> /var/logs
5. 'popd'명령어를 통해서 어떻게 동작하는지 확인한다.

함수 및 선언

bash

함수 및 별칭

bash

Function

함수는 명령어를 기능별로 묶을 때 사용하는 기능. C언어의 "void function(){} return 0"과 비슷한 기능.

함수는 일반적으로 여러가지 명령어를 사용하여 하나의 세트로 만든 다음에 그 기반으로 동작 순서(**routine**)를 구성한다.

Function

함수는 다음과 같은 구조로 생성한다.

```
function HelloWorldSkt(){  
  echo "Hello SKT"  
}
```

호출은 다음과 같은 명령어로 함수를 호출한다.

HelloWorldSkt

Function

생성한 함수를 확인 하고 싶은 경우, 다음명령어로 확인이 가능하다.

```
$ declare -F <function>
```

```
$ delclare -F HelloWorldSkt
```

Function

함수 생성 후 함수를 제거(**unset**)하고 싶은 경우 다음과 같은 명령어로 함수 선언을 해제한다.

```
$ unset <function>
```

```
$ unset HelloWorldSkt
```

Function

함수에서 변수 선언은 매우 단순하다.

```
var1=1
```

```
var2=2
```

```
function change(){
```

```
    echo internal function
```

```
    echo "Variable 1 is: $var1"
```

```
    echo "Variable 2 is: $var2"
```

```
    local var1=5
```

```
var2=5
```

```
    echo
```

```
    echo "After change internal function
```

```
    echo Variable 1 is locally $var1
```

```
    echo Variable 2 is gobally $var2
```

```
}
```

Function Arguments

bash에서 함수 인자 값은 다음처럼 다룬다.

\$: 예약된 쉘 함수 인자 값. 보통 **\$0**번은 **명령어(스크립트)**실행 위치를 표시

\$1, \$2, etc: **\$1~\$9**번까지는 일반적으로 받는 변수 이름

\$#: 현재 가지고 있는 인자 값 출력

Function

bash에서 함수 인자 값은 다음처럼 다룬다.

\$@, \$*: 둘 다 기능은 비슷하다. 차이점은 리스트이냐 혹은 문자열이냐 차이다.

\$@: 문자열을 리스트 형태로 변환한다. 스페이스 기반으로 리스트 배열을 구성한다.

\$*: 리스트를 하나의 문자열로 만든다. 스페이스 기반으로 분리한다. \$1, \$2

종료코드

bash에서 사용하는 리턴 코드는 보통 다음과 같다.

- 1: 일반적인 오류 코드.
- 2: 빌트인 키워드 올바르게 찾지 못함, 예를 들어서 올바르지 않는 함수.
- 126: 명령어 실행이 되지 못함. 예를 들어서 /dev/null를 명령어로 접근.
- 127: 명령어를 찾을 수 없음. \$PATH나 혹은 오타 typo)
- 128: 올바르지 않는 리턴 값. 예를 들어서 exit 3.14159.
- 128+n: 오류 코드. 예를 들어서 프로그램 종료 시 kill -9한 경우 128+9.
- 130: 스크립트 종료. 예를 들어서 ctrl+c 128코드에 +2.
- 255*: 프로그램이 종료가 되었으나, 범위에 포함이 되지 않음.

alias

'alias'는 긴 명령어 혹은 명령어 "+"옵션을 짧게 줄여서 사용하기 위해서 사용한다.
설정 되어있는 'alias'를 확인하기 위해서 다음 명령어로 확인이 가능하다.

\$ alias

\$ alias -p

alias

'alias'설정은 다음처럼 한다.

```
$ alias move='mv -l'
```

```
$ alias Documents='/tmp/documents/'
```

alias

해제하기 위해서는 다음 명령어로 'alias'해제가 가능하다.

\$ unalias -a

\$ unalias move

alias

해제하기 위해서는 다음 명령어로 'alias'해제가 가능하다.

\$ unalias -a

\$ unalias move

function

뒤에서 함수에 대해서 더 자세히 다룸.

여기에서 다루는 함수는 간단한 함수 선언 방법. C언어 함수와 동일한 기능을 하며, 사용하는 쉘 마다 함수 선언 방법 혹은 방식을 다를 수 있음.

function

Function in FISH

```
function ll
```

```
  ls -l $argv
```

```
end
```

Function BASH

```
function quit {
```

```
  exit
```

```
}
```


function

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

```
bash$ type FirstFunction
```

FirstFunction is a function

```
FirstFunction ()
```

```
{
```

```
    :
```

```
}
```

```
bash$ alias FirstAlias=:
```

FirstAlias is aliased to `:'

set

'set'명령어는 빌트인 명령어.

'set'은 현재 사용중인 셸에 설정되어 있는 값을 화면에 변수 이름과 함께 출력한다. 기존에 사용하던 값을 'set'명령어를 통해서 설정 및 변경도 가능하다.

알파벳 순서대로 정렬해서 화면에 출력한다. 여기에 출력된 변수들은 전부 셸에서 명시된 값이다.

set

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

set

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

set명령어는 "-v"옵션을 통해서 설정된 값 확인이 가능하다. 확인 하는 방법은 **printenv**, **env**, **compgen** 같은 명령어 사용을 권장한다.

set

-e: 프로그램이 종료가 되면 0 상태 값을 반환한다.

하지만 이 방법을 스크립트에서 사용하는 경우 거의 대다수가 올바르지 않는 값을 반환한다. 그 이유는 -e 옵션은 결과값 예측이 어렵기 때문이다.

그래서 대다수 스크립트는 set -e를 사용하지 않고 자체적으로 오류 값 핸들링을 더 선호한다.

set -e

```
$ nano set1.sh
```

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

```
set -e
```

```
function hello()
```

```
{
```

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

```
}
```

```
hello
```

set -e

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

```
$ nano set2.sh
```

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

```
function hello()
```

```
{
```

```
    set -e
```

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

```
}
```

```
hello
```

set -e

보통 아래와 같이 'trap'명령어와 같이 사용하는 것을 권장한다.

```
$ nano set3.sh
```

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

```
set -e
```

```
function Error()
```

```
{
```

```
    echo "error occur at line $1"
```

```
}
```

```
trap 'Error $LINENO' ERR
```

```
errorss
```


set -e

```
$ sh set3.sh
```

```
set3.sh: line 9: errorss: command not found
```

```
error occur at line 9
```

"-e"가 올바르게 동작하면 아래처럼 어느 라인에 문제가 있는지 메시지를 출력한다. 하지만, 해당 오류는 정확한 정보가 아닌 경우가 더 많다. 'trap'를 사용하지 않는 경우, 어느 부분에서 문제가 발생하였는지 확인이 어렵다.

set -n

명령어를 읽어 오지만 실행하지 않는다. 보통 셸 스크립트 문법 확인 시 사용한다.

```
$ nano set4.sh
```

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

```
set -n
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -u

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

```
$ nano set5.sh
```

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

```
set -u
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -o

"-u"하고 비슷하지만 조금 더 자세하게 출력이 된다.

```
$ nano set6.sh
```

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

```
set -o nounset
```

```
os="linux"
```

```
echo $os
```

```
echo $unix
```

set -f

글러빙 끄기. 글러빙 기능을 비활성화 한다.

```
$ set -f
```

```
$ ls *
```

```
$ set -f
```

```
$ ls *
```

set -a

변수를 외부에서도 사용할 수 있도록 한다. 보통 셸은 다른 환경 혹은 PID가 다른 경우 이전 셸에서 사용하던 변수에 접근이 불가능 하다.

```
$ echo $$
```

```
$ set -a
```

```
$ number1=23
```

```
$ number2=34
```

```
$ bash
```

```
$ echo $$
```

```
$ echo number1
```

set -v

읽어오는 값을 그대로 화면에 출력한다. 디버깅시 매우 유용하지만, **-x**와는 다른 정보를 출력한다. 실행 시, 발생한 메시지를 자세히 출력한다.

```
$ bash -v set1.sh
```

set -x

이 옵션은 각각 인자 값이 실행하는 걸 출력한다. 이 옵션을 실행하면 bash스크립트 디버깅에 도움이 된다. 어떤 부분에서 bash프로그램이 입력을 받아서 확장이 되는지 확인이 가능하다.

```
$ bash -x set3.sh
```


set -x

디버깅시 다음과 같은 방법으로 진행이 가능하다. **set -x** 를 CLI에 같이 선언 혹은 매직-뱅(Magic-Bang)에 다음과 같이 선언한다.

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

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

set -x

```
$ nano set7.sh
```

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

```
var = 5
```

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

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

```
    echo $var
```

```
    sleep 2
```

```
done
```

"-x"를 선언 전. 일반적인 결과만 출력한다.

set -x

```
$ nano set8.sh
```

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

```
set -x
```

```
var = 5
```

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

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

```
    echo $var
```

```
    sleep 2
```

```
done
```

set -x

혹은 다음처럼 실행하여도 된다.

```
bash$ -x <SCRIPT>
```

declare

declare는 내장 명령어이다.

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

declare

\$ declare -p | head -5

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

\$ declare -r

```
BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote:force_ignores:histappend:interactive_comments:login_shell:progcomp:promptvars:sourcepath"
```

declare -r

변수를 읽기 전용으로 선언. 읽기 전용으로 선언이 되면 해당 변수는 변경이 불가능하다.

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

```
$ echo $test
```

```
ok
```

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

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

declare -i (integer)

정수형 변수를 선언한다. 소수점 선언은 불가능하다.

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

```
$ echo $x
```

```
$ x=ok
```

```
$ echo $x
```

```
0
```


declare -i (integer)

```
$ x=15
```

```
$ echo $x
```

```
15
```

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

```
$ echo $x
```

```
20
```

```
$ x=15.1
```

declare

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

-f: 쉘의 함수(function)만 출력

-p: 변수의 속성 출력

변수 속성을 제어 할 때, "+, -"기호로 속성 제어가 가능.

unset

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

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

```
$ unset <value_name>
```

```
$ unalias <alias_name>
```

재-출력

redirection

Input, Output, Redirection

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

Hello World!

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

```
$ cat myfile
```

Input, Output, Redirection

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

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

```
0
```

디렉터리 위치로 재 전달 혹은 출력하는 경우 다음처럼 한다.

```
$ cd
```

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

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

Input, Output, Redirection

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

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

이어 붙이기

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

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

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

```
$ cat test_doc.txt
```

First Command

Second Command

이어 붙이기

스크립트 기반으로 이어 붙이기. 실제로 이러한 방식으로 사용할 필요가 없지만, 아래와 같이 구성 및 사용이 가능하다.

```
$ vi append.sh
```

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

```
printf 'Starting script\n' >> log
```

```
printf 'Creating test directory\n' >> log
```

```
mkdir test || exit
```

```
printf 'Changing into test directory\n' >> log
```

```
cd test || exit
```

```
printf 'Writing current date\n' >> log
```

```
date > date || exit
```

이어 붙이기

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

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

"||" 연산자는 or역할을 한다.

일반 시스템 명령어(cut)

여러 개 필드를 동시에 처리를 하려면 아래와 같이 하면 된다.

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

```
root:/root
```

```
bin:/bin
```

```
daemon:/sbin
```

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

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

```
sync:/sbin
```

입출력 전환

표준 입출력 오류는 각각 숫자를 가지고 있다.

출력은 "**stdout(standard output)**", "**표준출력**"이라고 말한다.

오류는 "**stderr(standard error)**" 그리고 "**표준오류**"라고 부른다.

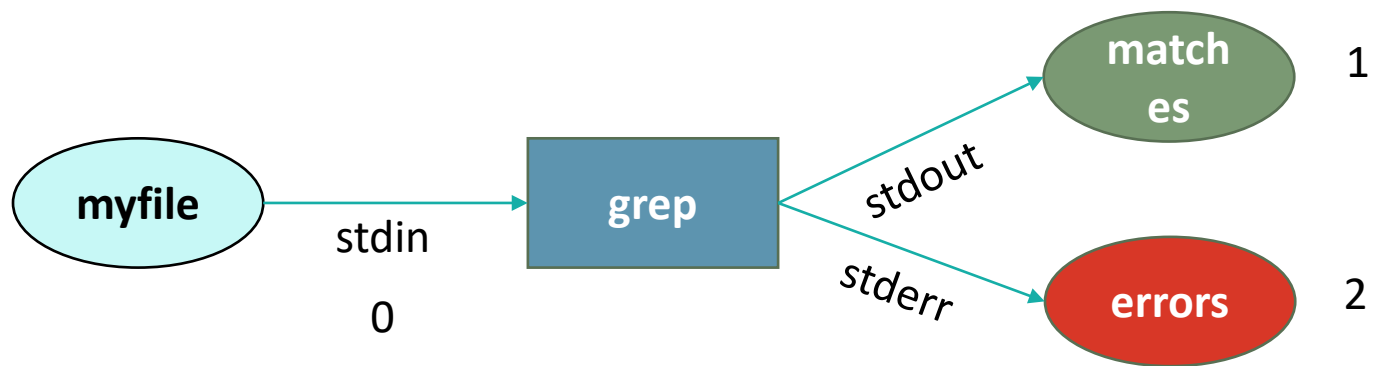
이 옵션은 매우 bash에서 스크립트에서 매우 일반적으로 사용한다.

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

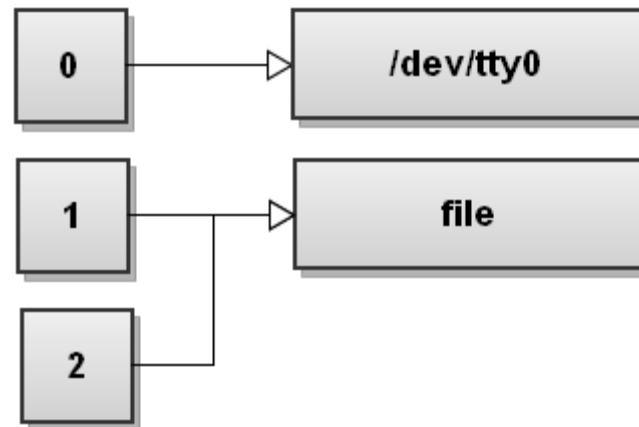
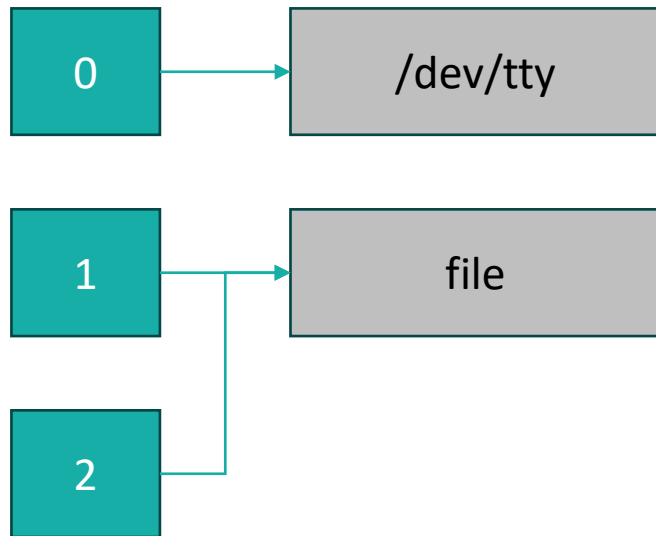
입출력 전환

- 0, 표준 출력 및 오류에 대한 동시 출력, 번호로는 0번 사용.
- 1, 표준출력, 오류가 정상 메시지를 화면에 출력한다.
- 2, 표준오류, 오류 부분에 대해서만 화면에 출력한다.

입출력 전환

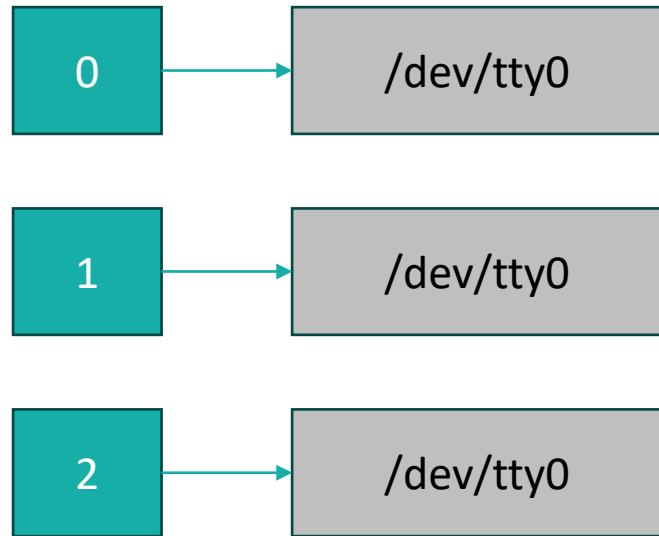


입출력 전환



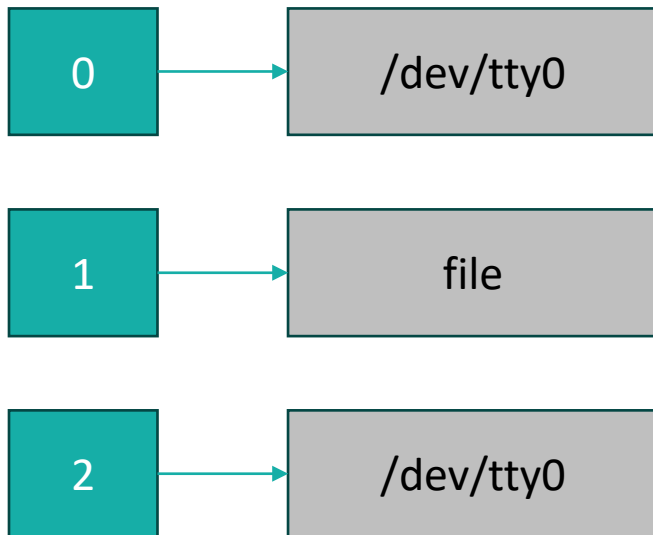
\$ ls >file 2>&1

입출력 전환



\$ ls

입출력 전환



\$ ls > file > 1

입출력 전환

오류와 표준 출력을 같이 하는 경우 보통 다음처럼 사용한다.

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

위에 처럼 사용하는 경우 현재 오류 메시지만 matches_error.txt파일에 기록 정상적인 메시지는 matches에 저장되어 있다. 그래서 같이 출력을 하기 위해서는 **&(ampersand)**를 통해서 1에 대한 출력을 명시 해주어야 한다.

입출력 전환

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

위와 같이 실행하면 위의 실행 결과는 표준 출력과 표준 오류를 동시에 파일에 저장한다. 주의 할 부분은 동일한 작업을 수행하기 위해서는 ">"앞에다가 **2>&1**을 넣을 수 없다. 리디렉션은 왼쪽에서 오른쪽으로 항상 이루어 진다.

입출력 전환

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

"/dev/null"

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

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

입출력 전환

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

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

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

```
$ echo $?
```

입출력 전환

```
$ ls -ld /tmp /tnt >/dev/null
```

```
ls: cannot access '/tnt': No such file or directory
```

```
$ ls -ld /tmp /tnt 2>/dev/null
```

```
drwxrwxrwt 1 bebop bebop 0 Jul 18 19:04 /tmp/
```

```
$ ls -ld /tmp /tnt >/dev/null
```

```
ls: cannot access '/tnt': No such file or directory
```

입출력 전환

```
$ ls -ld /tmp /tnt >/dev/null 2>&1
```

```
$ ls -ld /tmp /tnt 2>&1 >/dev/null
```

```
ls: cannot access '/tnt': No such file or directory
```

```
$ ls -ld /tmp /tnt | sed 's/^.*$/<-- & --->/'
```

```
ls: cannot access '/tnt': No such file or directory
```

```
<-- drwxrwxrwt 1 bebop bebop 0 Jul 18 19:04 /tmp --->
```

입출력 전환

```
$ ls -ld /tmp /tnt 2>&1 | sed 's/^.*$/<-- & --->/'
```

```
<-- ls: cannot access '/tnt': No such file or directory --->
```

```
<-- drwxrwxrwt 1 bebop bebop 0 Jul 18 19:04 /tmp --->
```

```
$ ls -ld /tmp /tnt >/dev/null | sed 's/^.*$/<-- & --->/'
```

```
ls: cannot access '/tnt': No such file or directory
```

```
$ ls -ld /tmp /tnt >/dev/null 2>&1 | sed 's/^.*$/<-- & --->/'
```

```
$ ls -ld /tmp /tnt 2>&1 >/dev/null | sed 's/^.*$/<-- & --->/'
```

```
<-- ls: cannot access '/tnt': No such file or directory --->
```


입출력 전환

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

하지만 꼭 tee명령어를 사용할 필요는 없다.

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

```
Copy Copy rom rom
```

입출력 전환

기존 명령어에 다음처럼 파일 이름만 더 추가한다.

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

```
Copy Copy rom rom
```

```
$ ls
```

```
myfile1 myfile2 myfile3
```

입력 재 전달

여기에서 사용한 정보는 다음 'man'페이지에서 참고 하였음.

```
$ man -Len -Pless\ +/^REDIRECTION bash
```

입력 재 전달

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

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

```
Hello, world!
```

```
HELLO, WORLD!
```

입력 재 전달

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

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

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

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

입력 재 전달

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

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

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

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

Multi Pipe

여러 개의 텍스트 파일을 하나로 통합할 때 다음처럼 스트리밍 처리한다.

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

```
$ date > output
```

```
$ hostname >> output
```

```
$ cat output
```

Multi Pipe

그룹 명령어는 중괄호를 사용하여 처리한다. 아래처럼 실행하면 모든 출력이 대문자로 출력이 된다. 또한 변수로 처리가 가능하다.

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

위의 명령어는 아래처럼 사용이 가능하다. 다만 방법은 다르다.

```
(date; hostname)
```


Multi Pipe

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

```
Mon Sep 19 17:17:19 KST 2022
```

```
'applen' 'carrotn' 'bananan'
```

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

```
Mon Sep 19 17:17:19 KST 2022
```

```
'applen' 'carrotn' 'bananan'
```

stream editor

에디터가 아닌 표준 입력을 통해서 편집하는 방법. 보통 스크립트 처리시 아래와 같은 방법으로 작업한다. 자주 사용하는 방법은 아니지만, 가끔 스크립트 작성 시 사용한다.

```
$ cat /usr/share/doc/lshw/README.md
```

```
$ sed "/usr/share/doc/lshw/README.md | head -2
```

```
lshw: HardWare LiSter for Linux
```

```
=====
```

```
$ sed '1,2d' /usr/share/doc/lshw/README.md | head -2
```

```
lshw is a small tool to provide detailed information on the hardware configuration of the machine. It
can report exact memory configuration, firmware version, mainboard configuration, CPU version and
speed, cache configuration, bus speed, etc. on DMI-capable x86 or EFI (IA-64) systems and on some ARM
and PowerPC machines (PowerMac G4 is known to work).
```

stream editor

간단하게 편집을 테스트 해보려면 다음과 같이 명령어를 실행한다.

첫번째는 "CONFIG_SYSFS_DEPRECATED" 문자열이 있으면 제거한다.

```
$ sed '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/systemd/README
```

두 번째는, 제거 후 해당 문자열이 있는지 명령어로 확인한다.

```
$ sed '1,/CONFIG_SYSFS_DEPRECATED/d' /usr/share/doc/systemd/README | grep CONFIG_SYSFS_DEPRECATED
```

마지막 세번째는 found를 not_found로 바꾼다.

```
$ sed 's/found/not_found/' /usr/share/doc/systemd/README
```

스트리밍, 핸들링

문자열

긴 문자열 처리

```
$ longstring1.sh
```

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

```
case $1 in
```

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

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

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

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

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

```
EOF
```

```
    exit 0
```

```
;;
```

```
esac
```

긴 문자열 처리

긴 문자열 처리가 필요 할 때는 연속적으로 받기 위해서 입력 재출력을 통해서 "<<" 추가적으로 계속 받는다. 이때 어느 부분이 파일의 끝 부분인지 표시하기 위해서 'EOF'라는 키워드를 사용한다. "cat << 'EOF'" 명령어는 입력라인에 EOF라는 문자열이 들어오면 더 이상 입력을 중지한다.

긴 문자열 처리

```
$ longstring2.sh
```

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

```
cat << EOF
```

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

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

```
EOF
```

긴 문자열 처리

```
$ longstring3.sh
```

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

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

```
    leading tabs
```

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

```
EOF
```

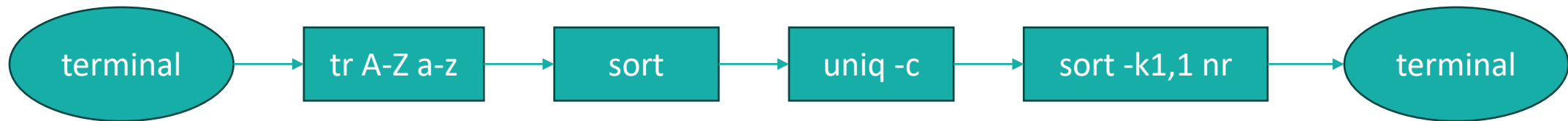

긴 문자열 처리

여러가지 명령어를 동시에 사용하기 위해서 셸 스크립트 형식으로 명령어를 작성한다. 이를 동시에 처리하기 위해서 파이프를 사용한다. 예제로 **tr**명령어 기반으로 간단하게 구성을 하였다.

긴 문자열 처리

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

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



긴 문자열 처리

아래처럼 문자열을 입력 후, "ctrl+d" 누르면 아래와 같이 내용이 출력이 된다.

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

```
sdlkjskljieurwjldklskdj
```

```
akljsnkcjnnjhdsiakjslia
```

```
38473904889orjksdjm,snbns
```

```
1 ($&$
```

```
1 2948
```

```
1 38473904889orjksdjm,snbns
```

긴 문자열 처리

위에서 사용한 내용을 셸 스크립트로 변경하면 다음처럼 변경한다.

```
$ longstring4.sh
```

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

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

```
sort |
```

```
uniq -c |
```

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

포맷팅

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

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

```
Type: json
```

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

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

```
Field name: type
```

자르기

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

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

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

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

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

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

길이측정

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

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

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

```
11
```

문자열 바꾸기

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

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

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

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

Yes, tomorrow, I'll do it today.

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

Yes, tomorrow, I'll do it tomorrow.

문자열 바꾸기

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

```
$ printf '%s\n' "${notice/today*/I\'ll do it soon.}"
```

Yes, I'll do it soon.

대소문자 변경

```
$ text='helloWorld'
```

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

```
HELLOWORLD
```

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

```
HelloWorld
```

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

```
helloWorld
```

외부 변수 합치기

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

```
temp="${test:-foo}"
```

```
printf '%s\n' "${temp#.ext}"
```

계산

four arithmetic operations

계산

```
$ tmp_a=3
```

```
$ tmp_b=2
```

```
$ printf 'The sum of the a + b is %u\n' $((tmp_a + tmp_b))"
```

The sum of the a + b is: 5

계산

```
$ diff=$((tmp_a - tmp_b))
```

```
$ printf 'The difference of the two numbers is: %u\n' "$diff"
```

The difference of the two number is: 1

```
$ printf '%u\n' "$(3**2)"
```

```
$ printf '%u\n' "$((180/60))"
```

```
$ printf '%u\n' "$((2>1))"
```

```
$ printf '%u\n' "$((30>40))"
```

계산

```
$ printf '%.2f\n' "$((3/2))"
```

```
1.00
```

```
$ bc <<'EOF'
```

```
scale=2
```

```
3/2
```

```
EOF
```

```
1.50
```

```
$ awk 'BEGIN { printf "%.2f\n", 3/2 }'
```

```
1.50
```

글로빙

globbing

글로브(*) 사용하기

글로브, 별표 혹은 눈이라고 말하는 문자이다. 셸에서는 글로브 즉 별표를 사용하면 기본적으로 모든 것(all of match)라고 표현한다. 매치가 되는 자원은 거의 모든 자원에 적용이 된다.

```
$ ls -a
```

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

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

```
hello
```

```
world
```

```
Documents
```

```
Downloads
```

글로브(*) 사용하기

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

Documents

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

test1.txt

test2.txt

글로브(*) 사용하기

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

Documents

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

jsues.txt

sjwue.txt

글로브(*) 사용하기

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

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

```
$ shopt -s nullglob
```

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

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

글로브(*) 사용하기

```
$ shopt -s nocaseglob
```

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

```
apple
```

```
Aux
```

글로빙 확장하기

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

?(pattern): 하나라도 패턴에 맞으면 출력

+(pattern): 최소 한 개로도 맞으면 출력

***(pattern):** 숫자 하나라도 맞으면 출력

@(pattern): 명시한 조건만 맞으면 출력

!(pattern): 조건에 맞지 않는 것 만 출력

글로빙 활용하기

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

/var/log에서 .gz로 끝나지 않는것만 출력

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

haw혹은 mess단어가 맞으면 화면에 출력

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

배열

bash

배열 사용하기

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

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

위의 배열은 매우 기본적인 배열 선언이다.

배열 사용하기

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

iphone

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

winphone

배열 사용하기

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

```
winphone
```

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

```
iphone
```

```
android
```

```
winphone
```

배열 사용하기

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

IPHONE

ANDROID

WINPHONE

배열 사용하기

```
$ homefiles=("$HOME"/*)
```

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

```
/home/tang/1.txt
```

```
/home/tang/2.txt
```

```
/home/tang/3.txt
```

```
/home/tang/4.txt
```

배열 사용하기

bash4.0 이후에 새로 할당된 배열 할당이 있다. 파이선처럼 기능을 제공한다.

```
$ declare -A colors
```

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

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

```
yellow
```

앤서블 모듈

셸 스크립트 응용

설명

앤서블 모듈은 파이썬 언어 기반으로 작성 권장. 하지만, 대 다수가 많이 사용하는 쉘 언어 기반으로도 사용이 가능.

이 목차에서는 짧지만 어떻게 쉘 스크립트 기반으로 작성하는지 그리고 집적 만들어서 테스트 해본다.

설명

여러 언어들을 사용하여 모듈을 만들 수 있지만, 간단한 지식을 통해서 빠르게 만들 수 있는 언어는 아마 쉘 스크립트가 유일하다.

Perl, Python, Go Lang, PHP같은 언어로 만들려고 하는 경우 다시 언어에 대해서 기본적인 학습이 필요하기 때문에 그러한 과정을 최대한 줄이고 간결하게 사용할 수 있도록 리눅스에서 많이 사용하는 bash script기반으로 작성한다.

vimrc

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

```
autocmd FileType yaml setlocal et ts=2 ai sw=2 nu sts=0
```

```
set cursorline
```

설명

앤서블 모듈은 다음과 같은 인자 값 핸들링 한다.

sh.check_agent:

name: cisco

type: network

위와 같은 구조를 가지고 있다.

설명

셸 변수 핸들링은 앞에서 학습 하였던 방식으로 입력을 받아온다.

```
$ source $1
```

```
$ echo $1 $2
```

1, 2번째 인자 값을 셸을 통해서 전달 받는다

출력

앤서블 모듈은 다음과 규칙이 적용이 된다.

changed 성공적으로 작업이 완료가 된 경우 출력되는 메시지. 올바르게 수행이 되면 **true**, 수행이 되지 않으면 **false**으로 반환한다.

failed: 작업 수행이 실패한 경우, 해당 작업에 대해서 실패하면 true를 반환한다.

msg: 성공/오류/변경 같은 작업이 문제없이 수행이 되는 경우 메시지를 반환 합니다.

출력

위의 기능은 JSON기반으로 동작하면 메시지는 다음과 같은 형식으로 전달 합니다.

echo '{"changed": true, "msg": "Updated some thing}"'

반드시 모든 데이터는 JSON형태, 그리고 중괄호 안에서 홑 따옴표 및 쌍 따옴표로 전달이 되어야 한다. 다만, 안쪽에 있는 데이터 형식은 딕셔너리는 형태로 반드시 쌍 따옴표로 구별이 되어 있어야 한다.

출력 시 주의사항

하지만, 셸 스크립트에서는 특수문자 핸들링 하기 위해서는 다음과 같이 두 가지 해결 방법이 있다.

`echo "{\“changed\”: true, \“msg\”: \“Updated some thing\”}"`

혹은 다음과 같이 bash에서 "printf"를 사용하여 출력이 가능하다.

`printf '{\“changed\”: true, \“msg\”: \"%s\"}' \"$msg"`

셸에서 JSON형태로 출력 시 위의 같은 형태로 구현만 해주면 된다.

출력 시 주의사항

하지만, bash또한 JSON형태로 출력이 불가능하기 때문에 파이썬 같은 도구를 사용해서 출력 도움을 받아야 한다.

```
msg="{\"changed\": true, \"msg\": \"$msg\"}"
```

```
echo $(echo "$msg" | python -c 'import json,sys; print json.dumps(sys.stdin.read())')
```

셸에서 실행하면 다음과 같은 결과가 출력이 된다.

출력

간단하게 정리하면, **echo**, **printf** 둘 중 아무거나 사용하여도 됨. 올바르게 JSON 형태로 출력하면 된다.

간단한 문법

앤서블 모듈 작성시 꼭 필요한 기본 함수. 명시된 함수는 없으나 모듈에서 사용하는 옵션 그대로 함수를 생성 해주어야 함.

예를 들어서 다음과 같은 옵션이 있으면 다음처럼 함수를 만들어야 함.

backup:

src: /usr/share/docs

dest: /tmp/docs.tar

label: 20220202_backup_ansible

case문 예외처리

이 문법을 통해서 예외 처리가 필요함. 아래는 모듈에서 사용하는 옵션 핸들링 하는 부분.

```
case $type in
    network)
        network
        ;;
    os)
        os
        ;;
```

앤서블 모듈 예제 및 함수 구성

sh.check_agent:

name: cisco

type: network

function network{}

function os{}

코드 내려받기

https://github.com/tangt64/training_memos/blob/main/opensource/bash-101/ansible_modules_examples/sh.agent_check

코드 구성

```
#!/bin/bash

function network
{
if [ $type == "network" ] && [ $name == "cisco" ]; then
    result_cisco=$(ps -ef | grep cisco | awk '{ print $NF }' | uniq | wc -l)
    if [ result_cisco -gt 1 ]; then
        contents=$(printf "$result_cisco" | python2 -c 'import json,sys; print json.dumps(sys.stdin.read())')
        output=$(printf "$result_cisco" 2>&1 | python2 -c 'import json,sys; print json.dumps(sys.stdin.read())')
        changed="false"
        msg="the cisco agent is alive"
    fi
}
```

코드 구성

```
else
```

```
    msg="the cisco agent is dead"
```

```
    output=$(printf "$result_cisco" 2>&1 | python2 -c 'import json,sys; print  
    json.dumps(sys.stdin.read())')
```

```
    error="Heelo"
```

```
    echo "{ \"failed\": true, \"msg\": \"$msg\", \"output\": \"$output\" }"
```

```
    exit 1
```

```
fi
```

```
}
```

코드 구성

```
source $1 $2  
if [ -z "$type" ]; then  
    printf '{"failed": true, "msg": "missing required arguments: name"}'  
    exit 1  
fi  
  
changed="false"  
msg=""  
contents=""
```


코드 구성

```
case $type in
    network)
        network
        ;;
    os)
        os
        ;;
    *)
        printf '{"failed": true, "msg": "invalid type: %s"}' "$type"

        exit 1
        ;;
esac

printf '{"changed": %s, "msg": "%s", "contents": %s}' "$changed" "$msg" "$contents"

exit 0
```

모듈 테스트

모듈 테스트 하기 위해서는 다음과 같은 명령어로 확인이 가능하다.

```
$ ansible localhost, -m sh.check_agent -a 'name=cisco type=network'
```

모듈 테스트

디버깅 하기 위해서는 bash의 -x옵션을 통해서 디버깅이 가능하다.

```
$ echo 'name=cisco type=network' > args
```

```
$ bash -x bashmod args
```