쉘 (Shell)

쉘 소개

쉘(Shell)이란 무엇인가?

- 쉘의 역할
  - 쉘은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어
  - 명령어 처리기(command processor) 혹은 명령어 해석기
  - 사용자로부터 명령어를 입력받아 이를 처리한다
  - 사용자 인터페이스

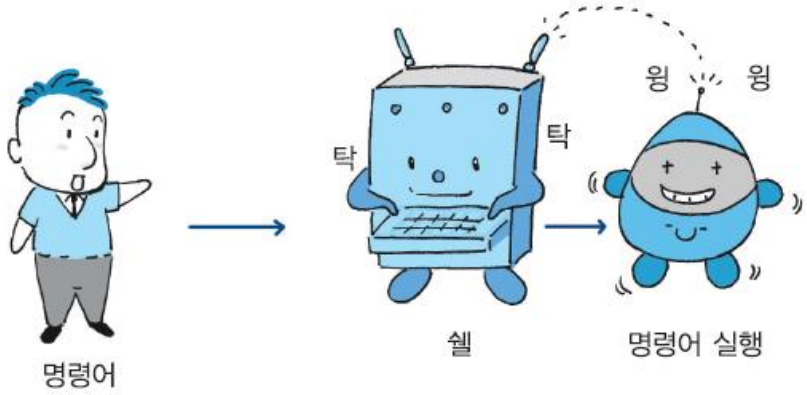
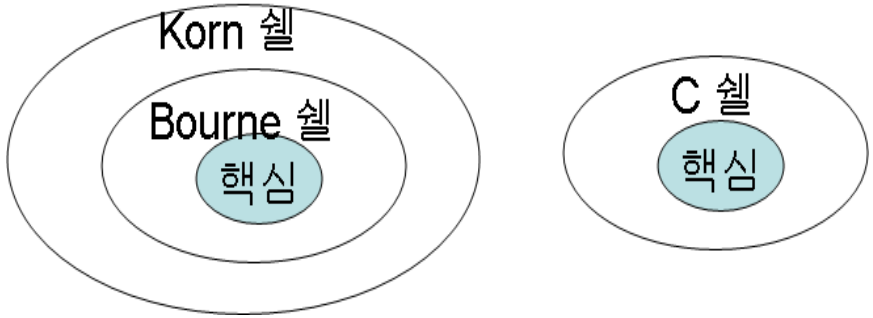


그림 6.1 쉘의 역할

쉘의 종류

- 유닉스/리눅스에서 사용 가능한 쉘의 종류
  - 모든 쉘은 공통의 핵심 기능은 공유
  - 추가로 각 쉘마다 고유의 특징을 가짐

쉘의 종류	쉘 실행 파일
본 쉘	/bin/sh
콘 쉘	/bin/ksh
C 쉘	/bin/csh
Bash	/bin/bash
tcsh	/bin/tcsh



기능 조금씩 다름

쉘의 종류

- 본 쉘(Bourne shell) (기본 $ 프롬프트)
  - 벨연구소의 스티븐 본(Stephen Bourne)에 의해 개발 됨
  - 가장 먼저 만들어진 쉘로서, 유닉스에서 기본 쉘로 사용됨
- 콘 쉘(Korn shell)
  - 1980년대에 역시 벨연구소에서 본 쉘을 확장해서 만들었음.
- C 쉘(C shell) (기본 % 프롬프트)
  - 버클리대학의 빌 조이(Bill Joy)에 의해 개발 됨
  - 쉘의 핵심 기능 위에 C 언어의 특징을 많이 포함함
  - BSD 계열의 유닉스에서 많이 사용됨
  - 최근에 이를 개선한 tcsh이 개발 되어 사용됨
- Bash(Bourne again shell)
  - GNU에서 본 쉘을 확장하여 개발한 쉘
  - 리눅스 및 맥 OS X에서 기본 쉘로 사용되면서 널리 보급됨
  - Bash 명령어의 구문은 본 쉘 명령어 구문을 확장함 → 따라서 대부분의 본쉘 스크립트는 Bash에서도 정상 동작

로그인 쉘(login shell)

- 로그인 하면 자동으로 실행되는 쉘
- 보통 시스템 관리자가 계정을 만들 때 로그인 쉘 지정

  /etc/passwd

  ...

  chang:x:109:101:Byeong-Mo Chang:/user/faculty/chang:/bin/csh

  user 정보 로그인 shell

- 로그인 쉘 변경(리눅스)

  $ chsh //change shell

  Changing login shell for chang

  Old shell : /bin/sh

  New shell : /bin/csh

  $ logout

  login : chang

  passwd:

  % (로그아웃 후 재로그인 하면 csh로 변경됨을 확인)

  사용 도중에도 임의로 쉘 변경 가능

  (단, 쉘 위에 또 다른 쉘이 실행되는 형태 → 서브 쉘, next slide)

로그인 쉘(login shell)

% logout

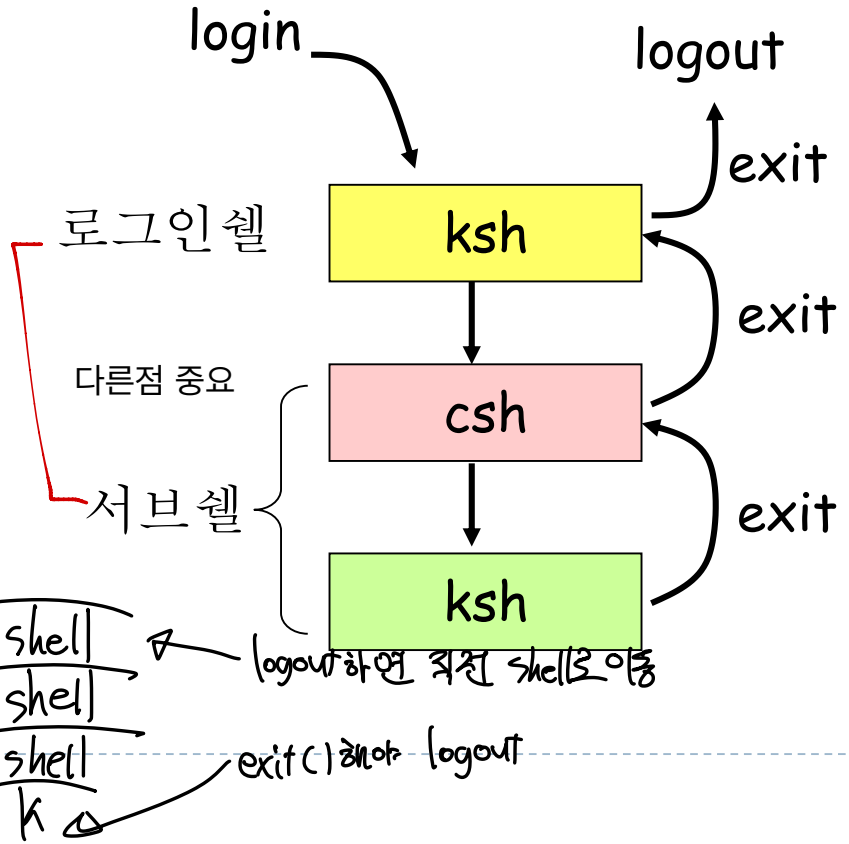
% exit

- 로그인 쉘과 서브 쉘
  - 로그인 쉘 : 사용자가 로그인한 직후 자동으로 생성되는 쉘
  - 서브 쉘 : 사용자가 직접 실행한 쉘

login → exit

```
telnet hanbit.co.kr
$ csh
% ksh
$ exit
% exit
$
```

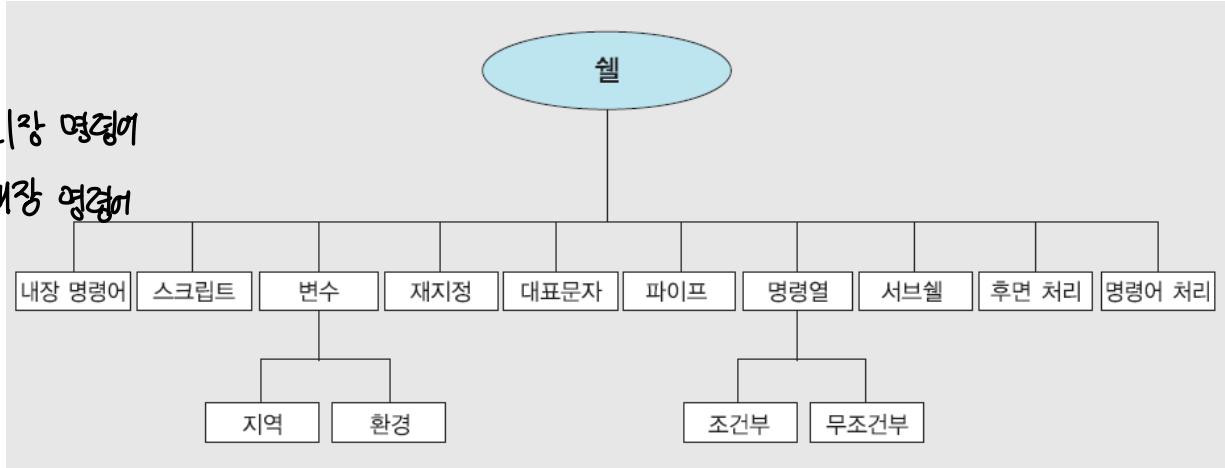
shell 위에 shell 실행



쉘의 기본 기능

쉘의 기본 기능

- 명령어 처리
  - 사용자가 입력한 명령을 해석하고 적절한 프로그램을 실행
- 시작 파일 역할
  - 로그인할 때 실행되어 사용자별로 맞춤형 사용 환경 설정
- 스크립트 작성 및 실행 기능 포함
  - 쉘 자체 내에서 스크립트 프로그래밍 기능을 가지고 있음
- 추가로 다음 그림과 같은 여러 기능 수행



외장 명령어

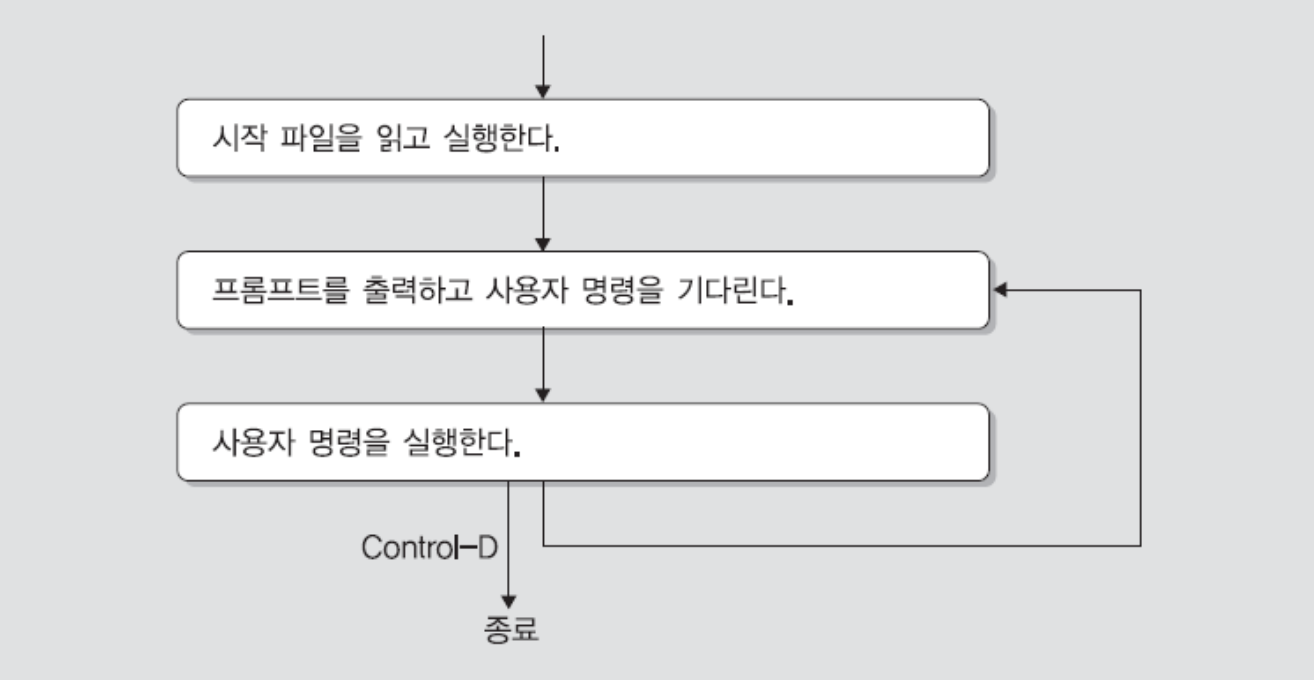
/bin, /usr/bin

- ls (디렉토리 목록 출력)
- grep (패턴 검색)
- find (파일 검색)
- cat (파일 내용 출력)
- tar (파일 압축/해제)

내장 명령어

- cd (디렉토리 이동)
- exit (쉘 세션 종료)
- echo (텍스트 출력)
- export (환경 변수 설정)
- pwd (현재 디렉토리 출력)

쉘의 실행 절차



쉘은 기본적으로 실행 대기 상태를 계속 유지하고 있다.

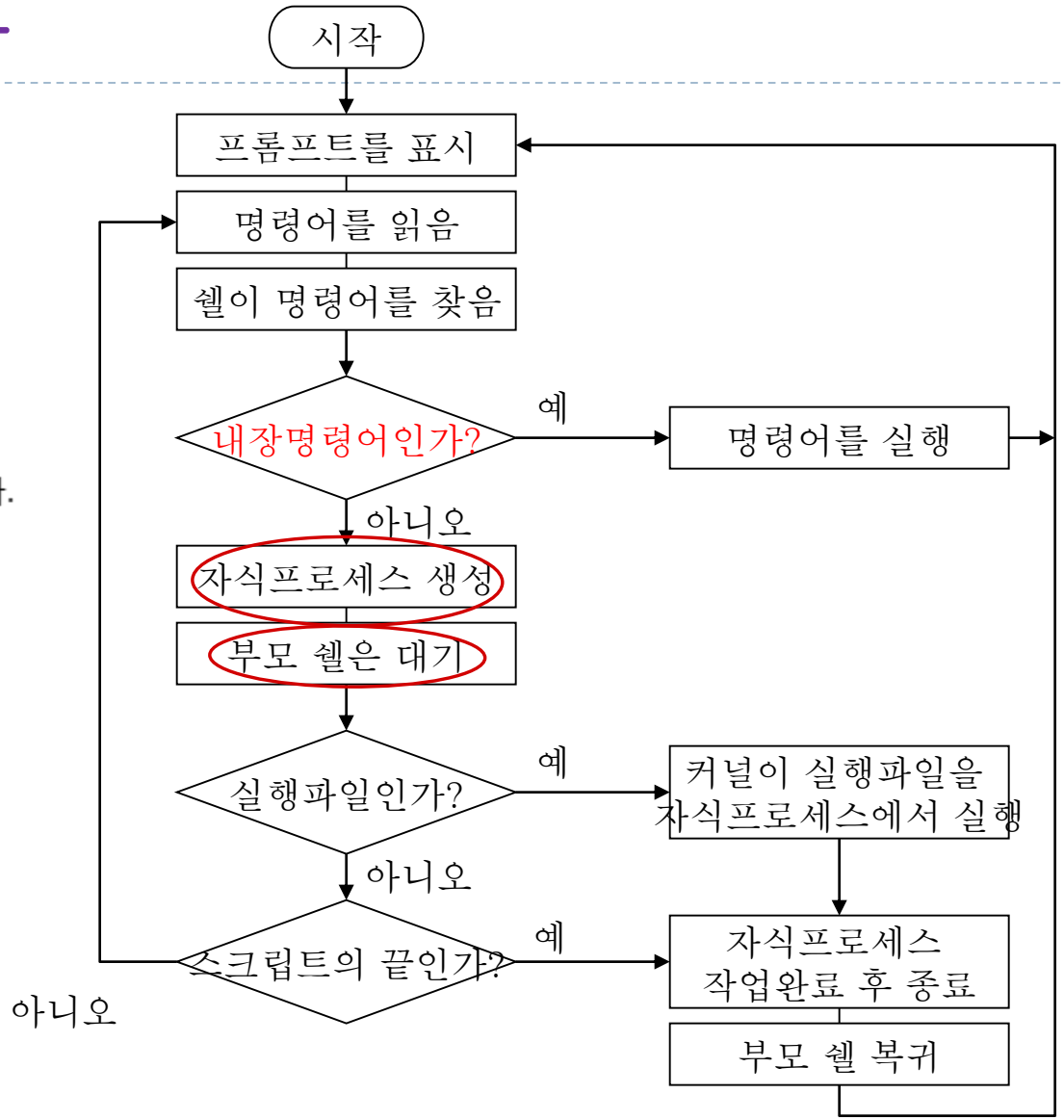
쉘의 실행 절차

외장 명령어는 자식 프로세스 생성

내장 명령어라면 현재 프로세스
내부에 존재하는 명령이므로
추가적으로
자식 프로세스를 실행 시킬 필요가 없다.

내장 명령어의 종류
- cd
- echo
- eval
- exec
- shift

참고) ls는 제외



시작 파일(start-up file)

- 쉘마다 시작 시, 자동으로 실행되는 고유 시작 파일이 존재
  - 다음 슬라이드 참조
- 시작 파일의 주요 역할 :
  - 사용자 환경을 설정하는 역할
- 환경설정을 위해서 환경변수에 적절한 값을 설정한다.
- $> 환경변수명=문자열 (공백 금지)

```
$> TERM=xterm
$> echo $TERM
xterm
```

(변수 이름 앞에 $ 사용 : 변수의 값)

- export 명령어

```
$ export TERM
```

(export 명령을 통해 변수를 환경 변수로 만들수 있으며, 이는 자식 쉘에서도 접근 가능)

- 환경 변수 확인

```
$ env  (혹은 set 명령어)
HOSTNAME=CS2
HOST=CS2
TERM=xterm
SHELL=/bin/sh
GROUP=faculty
USER=chang
...
```

쉘의 시작 파일(start-up file)

- 시작 파일
  - 쉘마다 시작될 때 자동으로 실행되는 고유의 시작 파일
  - 주로 사용자 환경을 설정하는 역할
  - 환경설정을 위해서 환경변수에 적절한 값을 설정한다.

  - 시스템 시작 파일 vs. 사용자 시작 파일 → 2가지로 구분됨
- 시스템 시작 파일
  - 시스템의 모든 사용자에게 적용되는 공통적인 설정
  - 환경변수 설정, 명령어 경로 설정, 환영 메시지 출력, ...
- 사용자 시작 파일
  - 사용자 홈 디렉터리에 있으며 각 사용자에게 적용되는 설정
  - 개별 환경변수 설정, 사용자 프롬프트 설정, 명령어 경로 설정, 명령어 이명(alias) 설정, ...

시작 파일(start-up file)

쉘의 종류	시작파일 종류	시작파일 이름	실행 시기
본 쉘	시스템 시작파일	/etc/profile	로그인
	사용자 시작파일	~/.profile	로그인
Bash 쉘	시스템 시작파일	/etc/profile	로그인
	사용자 시작파일	~/.bash_profile	로그인
	사용자 시작파일	~/.bashrc	로그인, 서브쉘
	시스템 시작파일	/etc/bashrc	로그인
C 쉘	시스템 시작파일	/etc/.login	로그인
	사용자 시작파일	~/.login	로그인 overwrite, 추가 설정
	사용자 시작파일	~/.cshrc	로그인, 서브쉘
	사용자 시작파일	~/.logout	로그아웃

숨긴파일

파일 위치 /etc 디렉터리
사용자 홈 디렉터리

실행 시점 사용자가 로그인할 때 시스템 전체 설정 적용
사용자가 로그인할 때 또는 서브쉘 실행 시 사용자의 개인 설정 적용

시작 파일 예 (/etc/profile)

telnet hanbit.co.kr

```
$ cat -n /etc/profile
     1  #ident  "@(#)profile 1.17 95/03/28 SMI" /* SVr4.0 1.3 */
     . . .
     6  export LOGNAME PATH
     . . .
    14                  TERM=sun
    15          fi
    16          export TERM
    . . .
    30                  /bin/cat -s /etc/motd
    . . .
    33                  /bin/mail -E
    34                  case $? in
    35                  0)
    36                          echo "You have new mail."
    37                          ;;
    . . .
    45  umask 022
```

단말기설정

메시지출력

메일 확인

기본사용 권한설정

슈퍼유저가 umask 022로 만들어달라고 함 변경하면 다음에 적용

시작 파일 예 (~/.profile)

- .profile

```
PATH=$PATH:/usr/local/bin:/etc    //사용자 개개인의 추가 경로 설정
TERM=vt100
export PATH TERM
stty erase ^h          (stty : 터미널 기능 설정)
```

- 시작 파일을 재로그인 없이 바로 적용 (. 마침표 명령어)

```
$ . .profile
```

쉘 명령어 처리

쉘 명령어 종류

- 내장 명령어
  - 쉘 내에 내장되어 있는 명령어
  - 별도의 실행 파일이 존재하지 않음

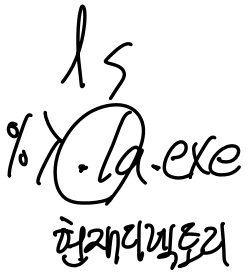
  $ echo
  $ cd

- 외장 명령어 : 유틸리티 프로그램
  - 명령어를 위한 실행 파일이 별도로 존재하는 명령어

    $ ls /bin/ls
  - 환경변수 PATH 상에서 경로 설정하여 사용

    PATH = .:/bin:/usr/bin:/usr/local/bin:/etc

    (쉘은 환경변수 PATH에서 지정된 순서대로 ls를 찾아 앞서 나온 /bin/ls 를 실행)



입출력 재지정 및 파이프

- 출력 재지정

  $ 명령어 > 파일
- 출력 추가

  $ 명령어 >> 파일
- 입력 재지정

  $ 명령어 < 파일
- 문서 내 입력

  $ 명령어 << 단어

  ...

  단어
- 파이프

  $ 명령어1 | 명령어2

복합 명령어

- 여러 개의 명령어를 묶어서 실행

- 명령어 열(command sequence)

  $ 명령어1; … ; 명령어n

  $ date; who; pwd (3개의 명령어를 순차적으로 실행)

- 명령어 그룹(command group) : 괄호를 사용하면 그룹이 됨

  $ (명령어1; … ; 명령어n)

  $ date; who; pwd > out1.txt (pwd만 출력 재지정)

  $ (date; who; pwd) > out2.txt (전체의 출력 재지정)

  (하나의 명령어처럼 사용되어 표준 입력/출력/오류를 공유함)

조건 명령어 열(conditional command sequence)

- $ 명령어1 && 명령어2

  $ gcc myprog.c && a.out (앞선 명령어 성공 시, 뒤 명령어 실행)

- $ 명령어1 || 명령어2

  $ gcc myprog.c || echo 컴파일 실패

조건 명령어 [
- 앞선 명령어가 실패 시, 뒤 명령어 실행 → 일반적인 || 의미와 다름
- 앞선 명령어가 성공 시, 뒤 명령어 실행되지 않음

무시

명령어 열(command sequence)

- 세미 콜론(;)의 사용: 명령어 열
  - 나열된 명령어들을 순차적으로 실행한다.
- 사용법

> $ 명령어1; ... ; 명령어n
>
> 나열된 명령어들을 순차적으로 실행한다.

- 예

```
$ date; pwd; ls
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
list1.txt list2.txt list3.txt
```

명령어 그룹(command group)

- 명령어 그룹
  - 나열된 명령어들을 하나의 그룹으로 묶어 순차적으로 실행한다.
- 사용법

> $ (명령어1; ... ; 명령어n)
>
> 나열된 명령어들을 하나의 그룹으로 묶어 순차적으로 실행한다.

- 예

```
$ date; pwd; ls > out1.txt
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
$ (date; pwd; ls) > out2.txt
$ cat out2.txt
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
...
```

조건 명령어 열(conditional command sequence)

- 조건 명령어 열
  - 첫 번째 명령어 실행 결과에 따라 다음 명령어 실행을 결정할 수 있다.

- 사용법

> $ 명령어1 && 명령어2
>
> 명령어1이 성공적으로 실행되면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

- 예

```
$ gcc myprog.c && a.out
```

조건 명령어 열

- 사용법

> $ 명령어1 || 명령어2
>
> 명령어1이 실패하면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

- 예

```
$ gcc myprog.c || echo 컴파일 실패
```

여러 개 명령어 사용: 요약

명령어 사용법	의미
명령어1; … ; 명령어n	나열된 명령어들을 순차적으로 실행한다.
(명령어1; … ; 명령어n)	나열된 명령어들을 하나의 그룹으로 묶어 순차적으로 실행한다.
명령어1 && 명령어2	명령어1이 성공적으로 실행되면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.
명령어1 || 명령어2	명령어1이 실패하면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

핵심

파일 이름 대치 (wild character)

- 대표문자(wild character)를 이용한 파일 이름 대치
  - 대표문자를 이용하여 한 번에 여러 파일들을 나타냄
  - 명령어 실행 전에 대표문자가 나타내는 파일 이름들로 먼저 대치하고 실행

이스케이프 \ , 따음표 " 사용하기

```
$ gcc *.c
$ gcc a.c b.c test.c
```

파일이 *.C일때 이떻게 될까?

정상 컴파일 됨

- 대표문자
  - `*` 빈 스트링을 포함하여 임의의 스트링을 나타냄
  - `?` 임의의 한 문자를 나타냄
  - `[..]` 대괄호 사이의 문자 중 하나

```
$ ls *.txt  (임의의 txt 파일)
$ ls [ac]*  (a 혹은 c로 시작하는 파일)
```

명령어 대치(command substitution)

- 명령어를 실행할 때 다른 명령어의 실행 결과를 이용
  - 역따옴표(`)로 에워싼 다른 명령어 부분
  - 그 명령어의 실행 결과로 대치된 후에 실행

$ echo 현재 시간은 `date`

```
echo 현재 시간은 date
-> 현재 시간은 date
```

```
현재 시간은 2024. 10. 26. (토) 22:48:56 KST
```

$ echo 현재 디렉터리 내의 파일의 개수 : `ls | wc -w`

현재 디렉터리 내의 파일의 개수 : 32

```
echo 현재 디렉터리 내의 파일의 개수 : ls | wc -w
7
```

Back quote is command substitution.

명령어 치환

Shell Escape

echo 현재 시간은 `date`
echo 현재 디렉터리 내의 : `ls | wc -w`
echo 3 * 4 = 12 //*이 현재 디렉토리로 의도 됨
echo "3 * 4 = 12"
echo 3*4 =12

- 따옴표를 이용하여 대치 기능을 제한 (shell escape)

```
$ echo 3 * 4 = 12
3 cat.csh count.csh grade.csh invite.csh menu.csh test.sh 4 = 12
$ echo "3 * 4 = 12"
3 * 4 = 12
$ echo '3 * 4 = 12'
3 * 4 = 12
$ echo 3*4=12        (띄어 쓰기 없을 경우 그대로 출력)
3*4=12
```

wild character

3과 4=12 사이에 * 기호에 의해 현재 디렉토리 내의 파일이 출력

쉘은 *을 모든(임의의) 이라고 해석하여 echo에게 전달한다.

작은 따음표 X
`
O

Shell이 처다보지 않음

```
$ name=나가수  띄어쓰기 x
$ echo '내 이름은 $name 현재 시간은 `date`'
내 이름은 $name 현재 시간은 `date`
$ echo "내 이름은 $name 현재 시간은 `date`"
내 이름은 나가수 현재 시간은 2011년 12월 8일 목요일 오후 03시 43분 12초
```

뿐만으로 Shell이 처다봄

Shell Escape

- Shell Escape별 용도
  - ` ` back quote : 명령어 치환
  - ₩ backslash : 특수한 한문자의 shell escape
  - ‘ ’ single quote : 특정 범위의 shell escape
  - “ ” double quote : 특정 범위의 shell escape + shell escape 범위 가운데 shell 파트 지정
    - 파트 지정 키워드 : $char, ₩char, `char`

Shell Escape

- 다음 결과를 예측해 봅시다.
  - %>echo $PATH
  - %>echo '$PATH' 문자열
  - %>echo "$PATH"
  - %>echo ₩$PATH
  - %>echo date 문자열
  - %>echo `date`
  - %>echo '오늘은 `date`입니다.' 문자열
  - %>echo "오늘은 `date`입니다."

쉘 스크립트

쉘 스크립트 (shell script)

- 쉘 스크립트는?
  - 명령어 및 유틸리티들을 적절히 사용하여 작성한 간단한 형태의 프로그램
  - 스크립트는 주로 interpretation 됨 (c.f., compilation)

한줄씩 해석

- 스크립트 파일 내에서 쉘 스크립트 종류 명시

  1. 첫 번째 줄에 사용할 쉘을 #!경로명 형태로 지정

     #!/bin/csh

     #!/bin/ksh 콘쉘

     #!/bin/bash

     #!/bin/sh

  쉘 경로 지정 없이 첫 번째 줄이 #으로 시작되면, C쉘 스크립트로 간주
  그 외는 본 쉘 스크립트로 간주한다.

쉘 스크립트 작성 및 실행

- 에디터를 사용하여 스크립트 파일을 작성한다.

  test.sh

```
#!/bin/sh    (본 쉘임을 명시)
echo 현재 시간:
date
echo 현재 사용자:
who
echo 시스템 현재 상황:
uptime
```

- chmod를 이용하여 실행 모드로 변경한다.

```
$ chmod +x test.sh   (스크립트 파일에 실행 속성 부여)
```

- 스크립트 이름을 타입핑하여 실행한다.

```
$ test.sh
```

```
export PATH=$PATH:.
```

작업 제어

프로세스 상태: ps

- ps [-옵션] 명령어

시스템 종류마다 옵션이 다른 대표적 명령어

  - 현재 존재하는 프로세스들의 실행 상태를 요약해서 출력

```
$ ps
PID TTY TIME CMD
25435 pts/3 00:00:00 csh
25461 pts/3 00:00:00 ps
```

- $ ps -aux (BSD 유닉스)
  - -a: 모든 사용자의 프로세스를 출력
  - -u: 프로세스에 대한 좀 더 자세한 정보를 출력
  - -x: 더 이상 제어 터미널을 갖지 않은 프로세스들도 함께 출력
- $ ps -ef (시스템 V)
  - -e: 모든 사용자 프로세스 정보를 출력
  - -f: 프로세스에 대한 좀 더 자세한 정보를 출력

kill

- kill 명령어
  - 현재 실행중인 프로세스를 강제로 종료

$ kill [-시그널] 프로세스번호

```
$ (echo 시작; sleep 5; echo 끝) &
1230
$ kill 1230
```

기본적으로 15번 시그널을 보냄
-9 시그널 사용 : 강제 종료 (sure kill)
kill -l 로 시그널 목록 확인

시그널 리스트

- $ kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR

주요 시그널

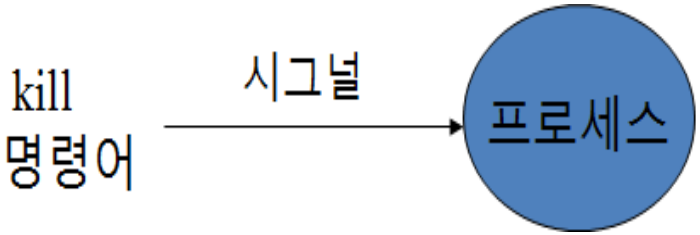
시그널 이름	의미	기본 처리
SIGABRT	abort()에서 발생되는 종료 시그널	종료(코어 덤프)
SIGALRM	자명종 시계 alarm() 울림 때 발생하는 알람 시그널	종료
SIGCHLD	프로세스의 종료 혹은 중지를 부모에게 알리는 시그널	무시
SIGCONT	중지된 프로세스를 계속시키는 시그널	무시
SIGFPE	0으로 나누기와 같은 심각한 산술 오류	종료(코어 덤프)
SIGHUP	연결 끊김	종료
SIGILL	잘못된 하드웨어 명령어 수행	종료(코어 덤프)
SIGIO	비동기화 I/O 이벤트 알림	종료
SIGINT	터미널에서 Ctrl-C 할 때 발생하는 인터럽트 시그널	종료
SIGKILL	잡을 수 없는 프로세스 종료시키는 시그널	종료
SIGPIPE	파이프에 쓰려는데 리더가 없을 때	종료
SIGPIPE	끊어진 파이프	종료

주요 시그널

SIGPWR	전원고장	종료
SIGSEGV	유효하지 않은 메모리 참조	종료(코어 덤프)
SIGSTOP	프로세스 중지 시그널	중지
SIGSTP	터미널에서 Ctrl-Z 할 때 발생하는 중지 시그널	중지
SIGSYS	유효하지 않은 시스템 호출	종료(코어 덤프)
SIGTERM	잡을 수 있는 프로세스 종료 시그널	종료
SIGTTIN	후면 프로세스가 제어 터미널을 읽기	중지
SIGTTOU	후면 프로세스가 제어 터미널에 쓰기	중지
SIGUSR1	사용자 정의 시그널	종료
SIGUSR2	사용자 정의 시그널	종료

시그널 보내기: kill 명령어

- kill 명령어
  - 한 프로세스가 다른 프로세스를 제어하기 위해 특정 프로세스에 임의의 시그널을 강제적으로 보낸다.



- 사용법

```
$ kill [-시그널] 프로세스번호
$ kill [-시그널] %작업번호
```

프로세스 번호(혹은 작업 번호)로 지정된 프로세스에 원하는 시그널을 보낸다.

시그널을 명시하지 않으면 SIGTERM 시그널을 보내 해당 프로세스를 강제 종료

시그널 보내기: kill 명령어

- 종료 시그널 보내기

```
$ kill -9 프로세스번호
$ kill -KILL 프로세스번호   //signal list에서 접두어 SIG의 뒷부분
```

- 다른 시그널 보내기

```
$ 명령어 &
[1] 1234
$ kill -STOP 1234                    // 중지
[1] + Suspended (signal) 명령어
$ kill -CONT 1234                    // 재실행
```

wait 명령어

이와 같은 wait 명령어는
스크립트 내에서의 명령어 실행 순서 제어도 가능하며
시스템 함수로도 구현되어 프로그래밍에서도 활용 가능하다

$ wait [자식프로세스번호]

- 해당 프로세스 번호를 갖는 특정 프로세스가 종료될 때까지 기다린다.
- 프로세스 번호를 지정하지 않으면 모든 자식 프로세스를 기다린다.

```
$ (sleep 10; echo 1번 끝) &
//가령 echo가 1231번 프로세스라 하자

$ echo 2번 끝; wait 1231; echo 3번 끝
2번 끝
1번 끝
3번 끝
```

```
$ (sleep 10; echo 1번 끝) &
$ (sleep 10; echo 2번 끝) &
$ echo 3번 끝; wait; echo 4번 끝
3번 끝
1번 끝
2번 끝
4번 끝
```

프로세스 우선순위

- 실행 우선순위 nice 값
  - 19(제일 낮음) ~ -20(제일 높음) ← 작은 숫자가 높은 우선순위
  - 보통 기본 우선순위 0으로 명령어를 실행

- nice 명령어

```
$ nice [-n 조정수치] 명령어 [인수들]
주어진 명령을 조정된 우선순위로 실행한다.
```

- 예

```
$ nice                          // 현재 우선순위 출력
0
$ nice -n 10 ps -ef             // ps 명령을 조정된 우선순위로 실행
```

프로세스 우선순위 조정

- 사용법

```
$ renice [-n] 우선순위 [-gpu] PID
이미 수행중인 프로세스의 우선순위를 명시된 우선순위로 변경한다.
  -g : 해당 그룹명 소유로 된 프로세스를 의미한다.
  -u : 지정한 사용자명의 소유로 된 프로세스를 의미한다.
  -p : 해당 프로세스의 PID를 지정한다.
```

프로세스의 사용자 ID

- 프로세스는 프로세스 ID 외에
- 프로세스의 사용자 ID와 그룹 ID를 갖는다.
  - 그 프로세스를 실행시킨 사용자의 ID와 사용자의 그룹 ID
  - 프로세스가 수행할 수 있는 연산을 결정하는 데 사용된다.

- id 명령어

```
$ id [사용자명]
```

사용자의 실제 ID와 유효 사용자 ID, 그룹 ID 등을 보여준다.

```
$ id
uid=1000(chang) gid=1002(cs) groups=1002(cs)
context=system_u:unconfined_r:unconfined_t:s0
```

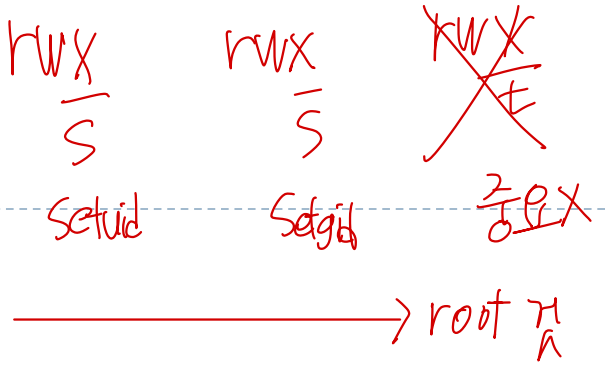
프로세스의 사용자 ID

- 프로세스 사용자 ID
  - 실제 사용자 ID (real user ID)
  - 유효 사용자 ID (effective user ID)

- 프로세스의 실제 사용자 ID(real user ID)
  - 해당 프로세스를 실행한 실제 사용자의 사용자 ID로 설정된다.
  - 예를 들어 chang이라는 사용자 ID로 로그인하여 어떤 프로그램을 실행시키면 그 프로세스의 실제 사용자 ID는 chang이 된다.

- 프로세스의 유효 사용자 ID(effective user ID)
  - 새로 파일을 만들 때 그 파일의 소유자 결정 용도로 사용
  - 혹은 파일에 대한 접근 권한을 검사하는 용도로 사용된다.

프로세스의 사용자 ID



- 유효 사용자 ID 사용법
  - 일반적으로 유효 사용자 ID와 실제 사용자 ID는 특별한 실행파일을 실행할 때를 제외하고는 동일하다.
  - Setuid가 지정되어 있으면, 해당 프로그램 실행 시, 파일을 실행한 사용자가 아닌, 파일 소유주의 uid를 파일 실행자의 effective uid에 부여

    → 즉, 프로그램 실행 시 현재 사용자의 EUID(Effective UID)를 프로그램 소유자의 RUID(Real UID)로 설정

set-user-id 실행파일

set uid의 필요성

강우에 따라 일반사용자가 시스템관리를 사용해야 함

- set-user-id(set user ID upon execution) 실행권한
  - set-user-id가 설정된 실행파일을 실행하면
  - 이 프로세스의 유효 사용자 ID는 그 실행파일의 소유자로 바뀜.
  - 이 프로세스는 실행되는 동안 그 파일의 소유자 권한을 갖게 됨.

1. 제한된 권한 환경에서 특정 작업 실행
2. 특정 프로그램에 권한을 제한적으로 부여
3. 보안 유지
4. 권한 상승을 통한 효율적 운영
5. 시스템 안정성 및 관리 효율성

- 예

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 27000 2010-08-22 12:00 /usr/bin/passwd
```

root권한(시스템)

real : me
effective : root

  - set-user-id 실행권한이 설정된 실행파일이며 소유자는 root
  - 일반 사용자가 이 파일을 실행하게 되면, 파일의 실행 동안, 이 프로세스의 유효 사용자 ID는 실제 소유주인 root가 됨
  - /etc/passwd처럼 root만 수정할 수 있는 파일의 접근 시 활용
  - 즉, 일반 사용자가 root 만 접근, 수정 할 수 있는 파일을 접근, 수정 할 수 있게 됨

set-group-id 실행파일

- set-group-id(set group ID upon execution) 실행권한
  - 실행되는 동안에 그 파일 소유자의 그룹을 프로세스의 유효 그룹 ID으로 갖게 된다.
  - set-group-id 실행권한은 8진수 모드로는 2000으로 표현된다.

- set-group-id 실행파일 예 : wall 명령어

  write wall

```
$ ls -l /usr/bin/wall
-r-xr-sr-x. 1 root tty 15344 6월 10 2014 /usr/bin/wall
```

rwx ← s

rw- ← S (x)

6666 ?

6666
실행권한이 없기 때문에 setuid/setgid의 효과는 적용X

setuid는 8진수 **4**로 표현됩니다.

setgid는 8진수 **2**로 표현됩니다.

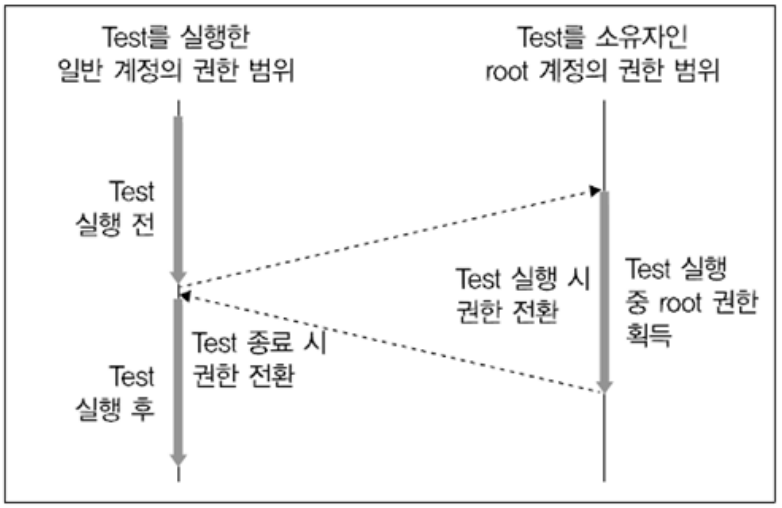
set-user-id/set-group-id 설정

- set-user-id 실행권한 설정

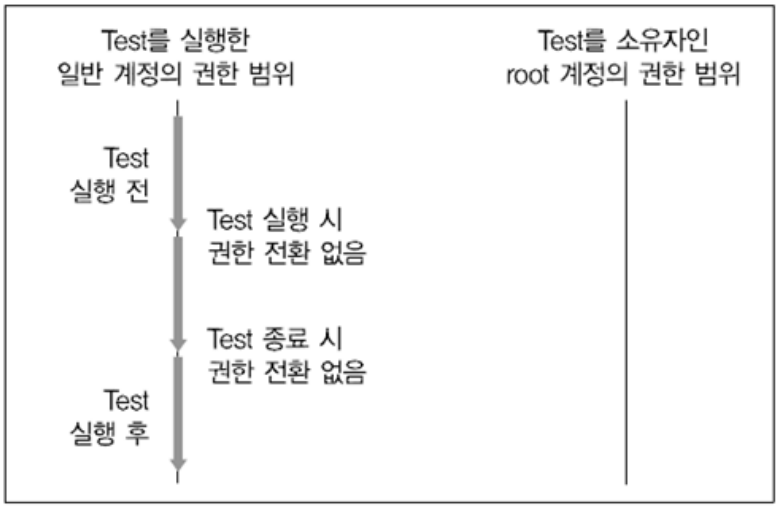
  $ chmod 4755 파일 혹은 $ chmod u+s 파일

- set-group-id 실행권한 설정

  $ chmod 2755 파일 혹은 $ chmod g+s 파일



(a) Test에 SetUID 비트가 있는 경우



(b) Test에 SetUID 비트가 없는 경우

(참고) Sticky Bit

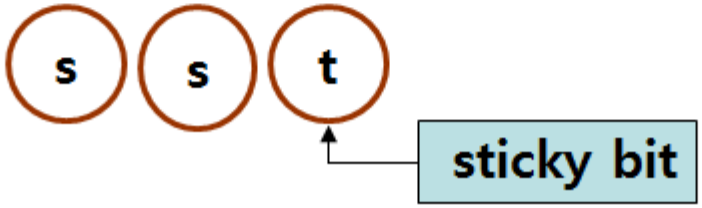
Sticky Bit

- 디렉터리에 sticky bit이 붙으면 디렉터리 내의 파일 삭제에 제한
  - 해당 디렉터리의 쓰기 권한이 있는자
  - 해당 파일의 소유자
  - 해당 디렉터리의 소유자
  - 슈퍼유저

  [예] /tmp

디렉토리 접근 권한에 ----wx-wx 가 포함될 경우
타인이 해당 디렉토리를 수정 및 삭제를 할 수 있음.
디렉토리 생성은 가능하나, 삭제는 루트만 가능하도록 설정

- 정규 파일에 sticky bit이 붙으면,
  - 가상 메모리에 로드된 프로그램을 가상 메모리에 존속하게 함



프로그램 종료 후에도,
SWAP 영역에서 제거되지 않음
다음 실행 시, 메모리에 빠르게 로딩
빈번히 쓰이는 프로그램에 유용
수퍼 유저만 설정 가능

강제 종료

- 강제종료: Ctrl-C

```
$ 명령어
^C
```

- 예

```
$ (sleep 100; echo DONE)
^C
$
```

- 실행 중지: Ctrl-Z

```
$ 명령어
^Z
[1]+ Stopped 명령어
```

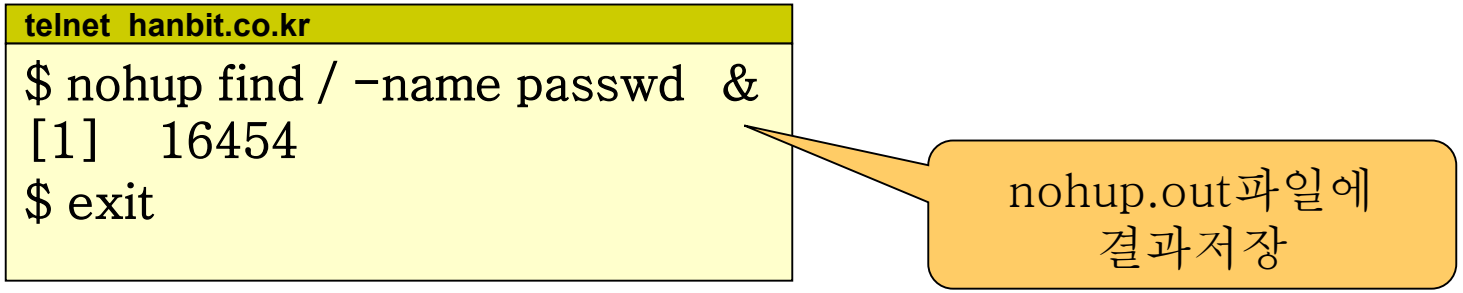
Ctrl-D
입력의 끝을 알림

기타 명령어

- exit
  - 쉘을 종료하고 종료값(exit code)을 부모 프로세스에 전달

  $exit [종료값]
- nohup
  - 로그아웃 이후에도 프로세스를 계속 실행하고자 하는 경우

  $ nohup 명령어 [인수] &
  - 명령어의 결과는 시스템 내의 nohup.out 파일에 기록



내장 명령어

eval, exec

- **eval**
  - eval은 명령어의 출력을 쉘 명령어로 인식하여 실행시킨다.
  - 즉, 출력을 실행하는 과정에 이를 명령어로 인식하여 실행

```
$ eval `echo x=5`
$ echo $x   ← 출력을 명령어로 처리하라!
5
```

eval 'x=5'

echo에 의해 x=5를 출력(echo)하면서 ← 실제 출력은 없음!
즉, x=5가 출력되지는 않으나
주어진 명령어 x=5를 실행시킨다.
따라서 echo $x로 값을 찍으면
쉘 변수 x가 5로 출력됨

- **Exec**
  - Exec는 쉘을 명령어로 대치하고 명령어를 실행한다.
  - 명령어 실행 후, 쉘 종료
    - Exec 의 정확한 메커니즘은 운영체제(시스템프로그램) 시간에 별도로 학습

```
$ exec date
```

(로그인 쉘이었다면 date 출력 후 로그아웃)

shift

- shift 명령어
  - 명령줄(command line) 인수를 하나씩 왼쪽으로 이동한다.

- shift.sh

```
#!/bin/sh
echo 첫 번째 인수 $1, 모든 인수 $*
shift
echo 첫 번째 인수 $1, 모든 인수 $*
```

- $ shift.sh a b c d

```
첫 번째 인수 a, 모든 인수 a b c d
첫 번째 인수 b, 모든 인수 b c d
```

별명 및 히스토리 기능

별명, alias (Bash shell)

- alias 명령어
  - 스트링이 나타내는 기존 명령에 대해 새로운 단어를 별명으로 정의

$ alias 단어=스트링

$ alias dir='ls –aF'

$ dir 띄어쓰기 x

$ alias h=history

$ alias list='ls –l'

- 현재까지 정의된 별명들을 확인 (인자없이 명령어만 실행)

$ alias

dir ls -aF

h history

list ls –l

- 이미 정의된 별명 해제

$ unalias 단어

별명 (C shell)

- alias 명령어
  - 스트링이 나타내는 기존 명령에 대해 새로운 단어를 별명으로 정의

```
% alias 단어 스트링          (BASH의 경우, alias 단어=스트링)
% alias dir ls -aF
% dir

% alias h history
% alias list ls -l
```

- 현재까지 정의된 별명들을 확인 (bash와 동일)

```
% alias
dir ls -aF
h history
list ls -l
```

- 이미 정의된 별명 해제 (bash와 동일)

```
% unalias 단어
```

히스토리, history (Bash shell)

- 입력된 명령들을 기억하는 기능
  $ history [option] [번호]

- 기억할 히스토리의 크기
  $ HISTSIZE=100

  따로 설정하지 않을 경우: 50
  (최신 버전은 1000이 기본 값)

- 로그아웃 후에도 히스토리가 저장되도록 설정
  $ HISTFILESIZE=100

  홈 디렉토리의 .bash_history 생성
  및 해당 파일에 저장

$ history 지금까지 사용된 명령들을 리스트 함.
1 ls
2 who
3 env
4 vi test.sh
5 chmod +x test.sh
6 test.sh
7 ls
8 date
9 history ← 가장 최근 명령어

히스토리 (C shell)

- 입력된 명령들을 기억하는 기능
  % history [option] [번호]
- 기억할 히스토리의 크기
  % set history = 40
  (배쉬의 경우, $ HISTSIZE=100)
- 로그아웃 후에도 히스토리가 저장되도록 설정
  % set savehist = 32
  (배쉬의 경우, $ HISTFIESIZE=100)
- 홈 디렉토리의 ~/.history 생성 및 해당 파일에 저장
  (배쉬의 경우 .bash_history 파일)

```
% history
1 ls
2 who
3 env
4 vi test.sh
5 chmod +x test.sh
6 test.sh
7 ls
8 date
9 history
```

재실행 (history 기능 활용)

형태	의미
!!	바로 전 명령 재실행
!n	이벤트 번호가 n인 명령 재실행
!시작스트링	시작스트링으로 시작하는 최후 명령 재실행
!?서브스트링	서브스트링을 포함하는 최후 명령 재실행

- 예

```
$ !!          # 바로 전 명령 재실행
$ !20         # 20번 명령어 재실행
$ !gcc        # gcc로 시작하는 최근 명령 재실행
$ !?test.c    # test.c를 포함하는 최근 명령 재실행
```

Bash Shell and C Shell 모두 동일

출력 재지정 (C shell)

- 표준출력 재지정

  % 명령어 > 파일

  % gcc a.c > errors

- 표준출력과 표준오류까지도 재지정 (stdout, stderr 모두 파일에 저장)

  % 명령어 >& 파일 오류 없으면 파일만 생성

  % gcc a.c >& errors (error 파일이 stdout과 stderr 모두 포함)

  컴파일 결과로 생성되는
  표준출력과 표준오류를 하나의 파일로 확인

- 표준출력 및 표준오류를 분리하여 저장

  % (명령어 > 파일1) >& 파일2

  % (gcc a.c > out) >& errors (out에 stdout, errors에 stderr 구분 저장)

  표준출력과 표준오류를 분리하여 저장

일반적으로 오류를 별도로 로깅하여 확인하고 싶을 때 사용

(참고) 출력 재지정 (bash의 경우)

```
$ cat x y 1> hold1 2> hold2          // 1> 과 >은 동일
$ cat hold1
This is y
$ cat hold2
cat: cannot open x

$ cat x y 1> hold 2>&1                    //& 띄어쓰기 없음 주의
```

표준 출력은 hold, file descriptor 2는 file descriptor 1의 사본
즉, 표준출력과 표준에러는 모두 hold에 출력

파이프 (C shell 추가 기능)

bash에서는 없음

- 명령어1의 표준출력이 파이프를 통해 명령어2의 표준입력

  % 명령어1 | 명령어2

  % gcc a.c | wc

  표준출력은 파이프로 전달되나
  표준에러는 모니터로 그대로 출력

- 표준출력 및 표준오류 둘 다 파이프 보내기 //Bash & C shell

  % 명령어1 |& 명령어2

  % gcc a.c |& wc

- 표준출력 및 표준오류를 분리 (구분하여 보내기) //C shell only

  % (명령어1 > 파일) |& 명령어2

  % (gcc a.c > out) |& wc (표준 오류만 wc에 전달됨)

핵심 개념

- 쉘은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어로 사용자로부터 명령어를 입력받아 이를 처리하는 명령어 처리기 역할을 한다.
- 입력 재지정은 명령어의 표준입력을 키보드 대신에 파일에서 받게 한다. 출력 재지정은 명령어의 표준출력을 모니터 대신에 파일에 하게 한다.
- 유닉스 명령어 및 유틸리티들을 적절히 사용하여 프로그램을 작성할 수 있는데 이러한 프로그램을 쉘 스크립트라고 한다.
- 쉘에서 사용할 수 있는 명령어는 쉘 내에 내장되어 있는 내장 명령어와 명령어를 위한 실행 파일이 별도로 존재하는 유틸리티 프로그램이다.