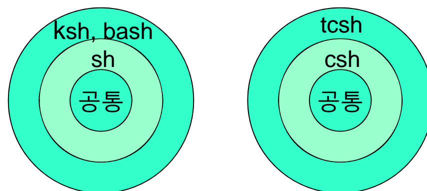
5. 셸(shell)

5.1 셸의 기능과 종류

- 셸(shell)
  - 사용자와 OS 사이의 인터페이스 프로그램
- 셸의 기본 기능
  - 명령어 해독기(command interpreter) 역할 수행
  - 셸의 종료 – ^D (입력끝), exit 명령어, 또는 logout (로그인 셸 만 해당)
- 셸의 추가 기능
  - 셸 프로그램 처리 기능 → shell script
  - 표준 입출력 방향 전환, 파이프 등의 다양한 기능 …
- shell의 종류
  - sh (Bourne shell) → bash (GNU Bourne again shell)
  - csh (C shell) → tcsh (enhanced C shell)
  - ksh (Korn shell) → zsh
  - ash (Almquist shell) → dash (Debian ash)

셸의 특징

- 각 shell의 특징
  - sh은 original shell로서 Bell Lab의 Steve Bourne이 개발함
    - 사용자 편의 기능에 부족하여 후에 많은 개선된 셸들이 등장
  - csh는 C언어 스타일의 script 언어 제공,
    - 작업제어, 히스토리, 별명 등의 사용자 편의 기능 추가
  - ksh은 David Korn이 sh을 확장하여 만든 shell, csh의 추가 기능 제공
  - bash은 sh을 대체하기 위해 GNU 프로젝트로 개발됨.
    - ksh, csh의 추가기능 제공,현재 가장 널리 사용됨 (Linux의 기본 셸)
  - ash, dash는 용량이 작은 shell로서 빠르게 실행됨



기본 로그인 셸 변경 및 서브셸

- chsh – 기본 셸 변경

  $ chsh … 새로운 셸의 경로명을 입력
  $ chsh –s /bin/bash … 셸의 경로명을 인수로 지정
  - 새로 로그인하여 기본 셸 변경 확인

  $ chsh –l … 사용가능한 셸 확인
  - /sbin/nologin을 기본 셸로 지정하면 로그인이 거절됨

- subshell 실행
  - 잠깐 동안 다른 셸을 사용하고자 할 경우에는 해당 shell을 명령어로 입력하여 서브셸을 실행

  $ csh
  % … csh 생성, csh 프롬프트 출력,
  …
  % exit … csh 종료

5.2 실행파일과 내장 명령어

- 내장 명령어(built-in command)
  - shell이 자체적으로 처리하여 실행할 수 있는 명령어
    (ex) cd, exit, logout, pwd 등

  $ man cd … BASH_BUILTINS 매뉴얼 출력

  $ man bash … bash 내장 명령어 설명
- 실행 파일(executable file) – 외부 명령어
  - 실행 허가권이 있는 파일. 실행파일 이름을 명령어로 사용함
  - 유틸리티 프로그램, 외부 명령어라고도 함
  - 일부 내장 명령어는 같은 이름의 유틸리티 프로그램이 존재함
    (ex) pwd, echo 등 (내장 명령어가 먼저 사용됨)
- 셸의 명령어 처리
  - 명령어가 **내장 명령어**이면 직접 처리하여 실행
  - 아니면 명령어 이름의 **실행 파일**을 찾아서 실행
    - 찾지 못하거나 실행허가권이 없으면 에러 메시지 출력
  - 명령어 실행 후 프롬프트 출력 – 다음 명령어 입력 기다림

출력 명령어

- echo (내장명령어)

  $ echo hello linux … 인수를 출력

  $ echo –n hello linux … 인수를 줄바꿈 없이 출력

  $ echo –n “hello linux ” … 마지막에 빈칸 출력
  - 유틸리티 프로그램도 존재(/bin/echo)
- printf – C언어의 printf 함수와 유사

  $ printf “hello linux\n”

  $ printf "%d + %d = %d\n" 10 20 30 ... 첫째 인수 = 출력 형식

실행 파일 경로

- 외부 명령어 실행
  - 명령어가 **절대경로이름** 또는 **상대경로이름**이면 해당위치의 파일 실행
  - 아니면, **실행파일 경로**(path)에 설정된 디렉토리를 순서대로 검색하여 명령어 이름의 실행파일을 찾아서 실행
- 실행파일 경로
  - 명령어에 대한 실행파일을 검색할 디렉토리들의 집합
  - **환경 변수** PATH 값으로 나타냄

$ echo $PATH … 현재의 경로 출력

5.3 셸의 메타 문자

- 메타 문자(meta character) – 셸에서 특수한 용도로 사용되는 문자
  - >, <, >> 표준입출력 방향전환(redirection)
  - *, ? , […] 파일 이름 대치, 대표 문자(wildcard)
  - | pipe
  - & background 처리
  - $ 변수 대치(변수 접근)

  . . .

  (교과서 80쪽 참조)
- 메타 문자의 특수 용도 제거
  - 메타문자 앞에 \ (backslash)를 함께 사용 (예: \>)

$ echo hello > world ... 출력을 파일 world에 저장(redirection)

$ echo hello \> world ... >도 출력

5.4 표준 입출력 방향전환

- 표준 입출력의 기본 설정
  - 표준입력(stdin): keyboard
  - 표준출력(stdout): 화면
  - 표준에러출력(stderr): 화면
- 표준 출력 방향전환 (output redirection) : **>, >>**
  - 표준출력으로 화면대신에 file을 사용 (file 출력)
  - 표준에러출력은 변화 없음

1) crate a file (file이 있으면 overwrite)
    $ data **>** stamp .. 출력을 파일에 저장

2) append to a file (file이 없으면 create, write 허가권이 없으면 error)
    $ data **>>** stamp .. 출력을 파일에 추가(append)

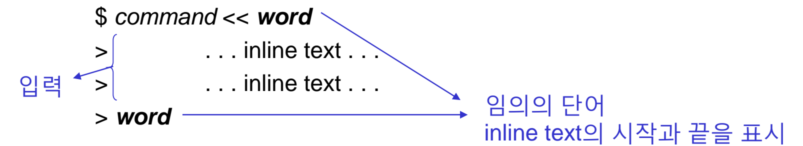
- 표준 출력 제거하기 – 특수파일 /dev/null로 redirection
    $ *command* > **/dev/null**

표준 입력 방향 전환

- 표준 입력 방향전환(input redirection) : **<**
  - 표준 입력으로 키보드 대신 file을 사용 (file 입력)
  - 입력 file이 없거나 read 허가가 없으면 error

    $ *command* < *infile*

- 표준 입력 방향전환 – Here Document
  - 표준 입력으로 명령어 행에 포함되는 **inline text**를 사용
  - 이 기능을 here document 라고도 부름

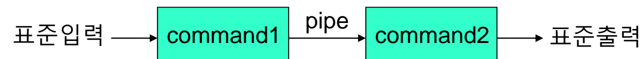


마지막 word까지 입력해야 명령어 입력이 종료되어 실행됨

5.5 파이프

- 파이프(pipe): |
  - 두 process들을 연결하여 한 process의 표준 출력을 다른 process의 표준 입력으로 사용함

    $ command1 **|** command2

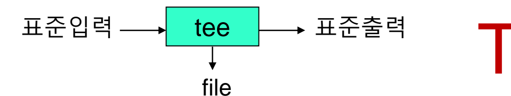


- 파이프라인 – 파이프로 연결한 process의 시퀀스

  $ ls | wc –w ; 현재 디렉토리의 파일/디렉토리 개수
  $ ls –l /bin | more ; 긴 출력을 화면 단위로 출력

파이프와 tee 유틸리티

- tee 명령어 - pipe 중간에 사용하여 pipe 중간 출력 저장
  - tee [-ia] file ; 표준입력을 표준출력과 file에 복사
    -i interrupt를 무시
    -a append to the file



$ ls | **tee** out2 | wc -w ; ls 출력이out2에 저장되고 wc의 입력으로 사용됨

(cf) $ ls | wc –w

$ date | tee out1 ; 명령어 출력을 화면으로 보면서 파일에 저장함

bc – 계산기

- bc (basic calculator) – 계산기
  - 무한 정밀도를 갖는 계산기
  - 대화형으로 수식을 입력, 계산 결과 출력
  - 기본적으로 정수 계산, scale을 지정하면 소수점 이하 자릿수 지정

```
$ bc
10+20       (입력)
30
scale=3     (입력)
1/3         (입력)
.333
 ^D
```

  - pipe를 사용한 수식 입력 - 비대화형

  $ echo 100 + 200 - 50 | bc

5.6 파일이름 대치 – wildcard

- 파일이름 대치(filename substitution)
  - **대표문자(wildcard)**를 사용하여 파일이름 패턴을 지정할 수 있으며
  - 인수들은 파일이름 패턴과 매칭되는 파일이름의 정렬된 목록으로 대치되어 명령어에 인수로 전달됨
- 대표문자(wildcard)
  - ***** 임의의 길이의 문자열 (ex) a* b*.c
  - **?** 임의의 한 문자 (ex) a? ???
  - **[ ]** 대괄호 내의 문자들 중 한 문자

    (ex) [A-Z] : 대문자 (**-** 는 문자 범위)

    [aeiou] : 모음 소문자

    $ echo **??? file? a***

    $ ls –l **/bin/[x-z]***

  - 매칭되는 파일이 **없으면** 대치되지 않고 **그대로** 인수로 사용

    $ echo z* ; z로 시작하는 파일이 없으면 z*를 출력

중괄호 파일이름 대치

- 중괄호 확장 (csh부터 제공)
  - abc{def, pqr}xyz → abc**def**xyz abc**pqr**xyz 으로 대치

    $ echo /usr/include/{stdio,signal}.h

  - 중괄호 안에 wildcard 사용 가능

    $ ls -l /usr/include/{std*,signal}.h

(프로그램 예) wildcard 동작 확인

- (예) 모든 인수들을 출력하는 프로그램

  $ vi arg.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i<argc; i++)
        printf("%d : %s\n", i, argv[i]);
    return 0;
}
```

$ cc arg.c ; a.out 생성

$ ./a.out a* b? ; 작성한 프로그램 실행

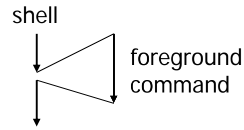
0 : **a**rg.c ; 프로그램에 전달된 인수는 파일이름 대치된 것임

1 : **a**.out

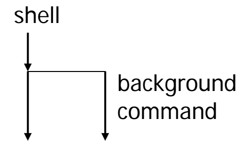
…

5.7 백그라운드 처리

- foreground 작업
  - 셸은 명령어 수행 종료 **후에** prompt를 출력하고, 다음 명령어를 입력
  - 보통의 명령어 수행 방법
    - 한 터미널에서 한번에 한 개의 작업 수행



- background 작업
  - 셸은 명령어 수행 종료와 **관계없이** 즉시 prompt를 출력하고 다음 명령어를 입력
  - 한 터미널에서 여러 작업을 동시 수행 가능
  - 실행시간이 긴 명령어 실행 시에 주로 사용



- background 실행 - &

  $ sleep 100 **&**

  [1] 13050 ; 1은 작업번호, 13050은 프로세스 번호

  $ _

백그라운드 작업과 redirection

- background 작업 종료 보고
  - 종료 후 shell에서 Enter가 입력될 때에 (명령어 없이도 가능) 출력
- background 작업의 표준입력
  - 터미널 입력(키보드)를 제어하지 않아서 키보드 입력을 사용할 수 없음
  - 키보드 입력이 필요하면 일시 중지(stop)되거나 에러 발생

    $ bc &

    …

    [1]+ Stopped bc

  - 반드시 표준입력을 미리 준비된 입력 파일로 redirection해야 함

    $ bc –l < bc_pi &

    "bc_pi"

scale=10000	.. 소수점 이하 개수
4 * a(1)	.. 원주율 계산

  - 간단한 입력은 pipe로 전달 가능

    $ echo "scale=1000; 4*a(1)" | bc –l &

백그라운드 작업과 redirection (2)

- background 작업의 표준 출력
  - 표준 출력은 화면(tty)에 출력 → foreground 작업의 출력과 섞일 수 있음
  - 표준 출력 redirection을 사용하는 것이 바람직

    $ bc -l < bc_pi > out_pi &

    $ echo "scale=10000; 4*a(1)" | bc -l > out_pi2 &

5.8 명령어 시퀀스와 명령어 그룹

- 명령어 시퀀스 ;
  - 여러 개의 명령어를 순서대로 실행

    $ ls; pwd; date

  - redirection은 각 명령어 별로 개별적으로 적용

    $ ls > ls.txt ; pwd ; date > date.txt

  - background 작업의 시퀀스는 ; 없이 &만 사용

    $ sleep 100 & sleep 60 &

    $ sleep 100 & pwd

    $ pwd ; sleep 100 &

명령어 그룹

- 명령어 그룹
  - 괄호 안에 있는 명령어 시퀀스로 구성
  - 서브셸을 생성한 후 서브셸에서 명령어 실행

```
$ (cd /bin; pwd)            ; 서브셸에서 디렉토리 이동
/bin
$ pwd                       ; 현재 디렉토리 그대로
```

  - 명령어 그룹의 명령어들은 표준입출력, 표준에러출력을 공유

```
$ (ls -C; pwd ; date) > out.txt
```

- 복잡한 명령어 그룹 사용

```
$ (cmd1; cmd2) & cmd3
$ cmd1 & cmd2 & cmd3
$ (cmd1; cmd2) & (cmd3; cmd4) &
```


중괄호 명령어 그룹

- 중괄호 명령어 그룹
  - 입출력 방향전환 공유 – ( ) 명령어 그룹과 같음
  - 현재셸에서 실행 – ( ) 명령어 그룹과 다름

```
$ { cd /bin; pwd; }          ; 디렉토리 이동을 포함
/bin
$ pwd                        ; 이동된 디렉토리 그대로
```


5.9 종료 코드와 조건부 명령어 시퀀스

- 종료코드(exit값)
  - 프로세스는 종료 코드를 갖고 종료
    - 성공 : exit값 = 0
    - 실패 : exit값 = non-zero (내장명령어는 1)
      - (**교과서 p93의 표 참조**)
  - C언어 프로그램에서
    - main() 함수의 종료 : exit 값 = main함수의 return 값
    - exit() 함수를 사용한 종료 – exit값 = exit() 함수의 인수
  - 셸의 exit 명령어

```
$ exit 1        ; exit 값=1 을 반환하고 셸을 종료
$ exit          ; 이전 명령어의 exit값을 반환하고 셸을 종료
```

- 종료코드 보기 - $?
  - echo $?   ; 이전 명령어의 exit값을 출력

```
$ date; echo $?          ; 정상 실행
$ data –k; echo $?       ; 잘못된 옵션
```


그룹의 종료코드와 조건부 명령어 시퀀스명령어

- 명령어 그룹의 종료코드
  - 마지막에 실행한 명령어의 exit값
  - exit 명령어를 사용하여 명시적 지정 가능
    - $ (echo hi; **echo linux**); echo $?
    - $ (echo hi; echo linux; **exit 10**); echo $?
- 조건부 명령어 시퀀스
  - 성공 조건부 시퀀스 : &&
    - $ **true** && echo success
    - $ cc prog.c && a.out
  - 실패 조건부 시퀀스 : ||
    - $ **false** || echo failure
    - $ cc prog.c || echo compilation error
  - true/false 명령어 – 항상 성공/실패인 명령어

5.10 작업 제어

- 작업제어
  - UNIX/Linux는 multitasing 기능 제공
  - 여러 작업에 대한 제어 동작이 필요
    - 작업의 포그라운드 – 백그라운드 간에 전환
    - 작업 일시 중지
    - 작업 종료
- ps – 프로세스 상태 보기 (process status)

  $ ps ; 현재 터미널에서 수행한 프로세스 정보
  $ ps –f ; 상세정보
  $ ps –ef ; 전체 프로세스 정보(-e)
  $ ps –u gdhong ; 특정 사용자 프로세스 정보
  $ ps axu ; BSD UNIX 옵션(모든 프로세스, 상세정보)
  - 상세정보 내역 – p96, p97 표 참조

Jobs

- jobs – (백그라운드/일시정지) 작업 목록 보기 (csh 이후)

  $ jobs
  [1] Running sleep 1000 &
  [2]- Running sleep 200 & -: 마지막에서 두 번째 참조
  [3]+ Running bc -l < bc_pi & +: 마지막 참조
- 작업 지정
  - 작업제어 명령어에서 작업을 지정하는 데 사용

    %2 ; 작업번호 2인 작업
    %prefix ; prefix로 시작한 작업
    %+, %% ; 마지막 참조 작업
    %- ; 마지막에서 두 번째 참조 작업

프로세스 종료

- Ctrl-C : 포그라운드 작업 종료
- kill : 백그라운드 작업 종료
  - kill은 프로세스에게 signal을 보내는 동작 수행
  - 주로 프로세스를 종료시킬 때 사용
    - default signal = TERM(15번) : 종료(terminate)

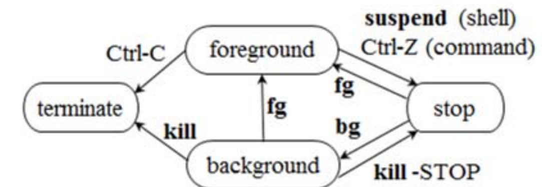
    $ kill 23202
    $ kill %2
  - TERM signal에 의해 종료되지 않는 프로세서는 KILL(9번) signal 사용

    $ sleep 1000 &
    $ kill -9 *pid* 또는 kill –KILL *pid*
- signal 목록 출력

  $ kill –l

작업 상태 전환

- 일시 중지
  - **^Z** (포그라운드 작업),
  - kill –STOP (백그라운드 작업) (csh에서는 stop 명령어 제공)
  - suspend (셸) – 잠시 부모 셸을 사용하고자 할 때 사용
- fg – 포그라운드 전환
  - fg [%job] ; 인수가 없으면 마지막 참조 작업
- bg – 백그라운드 전환
  - bg [%job]



5.11 변수와 변수 대치

- 변수
  - 문자열을 저장
  - 변수 이름 – 숫자, 문자, _ 로 구성, 숫자로 시작하지 않음
- 변수의 종류
  - **환경 변수** – 서브셸/자식 프로그램에게 변수 값이 복사되어 전달됨
  - **지역 변수** – 현재 셸에서만 사용
- 변수 값 할당 및 접근
  - 할당: $ var=value (csh은 set var = value)
  - 접근: $ echo $var
    $ echo ${var}ing
  - "$변수"는 변수 값으로 대치됨 → **변수 대치**(variable substitution)

환경 변수

- 환경변수 지정
  $ export var ; 기존 변수를 환경변수로 지정
  $ export var=value ; 변수값을 할당하면서 환경변수로 지정
  (csh에서는 setenv var value)
- 모든 환경변수 출력
  $ export (csh에서는 setenv)
- 모든 변수 출력
  $ set (csh에서는 set은 지역변수 출력)

5.12 미리 정의된 변수

변수	의미
HOME	홈 디렉토리의 경로. 로그인 할 때의 작업 디렉토리로 사용됨
PATH	명령어를 검색할 디렉토리 경로들의 :으로 구분된 리스트
CDPATH	cd 명령어에서 이동할 디렉토리를 검색할 디렉토리 경로들의 리스트
MAIL	메일박스 파일의 경로이름
PS1	1차 프롬프트 문자열
PS2	2차 프롬프트 문자열
IFS	셸에서 명령어와 인수들의 구분자로 사용되는 문자들
SHELL	셸의 경로이름
SHLVL	현재 셸의 레벨 단계
LANG	프로그램 메시지가 여러 언어로 준비된 경우에 출력에 사용할 언어

PATH – 실행파일 경로

- PATH – 실행파일 경로 디렉토리 지정 (:로 구분)
  $ echo $PATH
  - 디렉토리들은 콜론(:) 으로 구분, 앞의(왼쪽) 디렉토리부터 검색
- PATH 변경/추가
  $ PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin ; 절대 지정
  $ PATH=${PATH}:~/bin ; 뒤에 추가
  $ PATH=~/bin:$PATH ; 앞에 추가 (우선권)
- 유틸리티 overload
  - 같은 이름의 명령어 파일이 두 개 이상 존재하면
    - PATH의 앞(왼쪽)에 위치한 디렉토리의 명령어가 수행됨
  - 추가되는 경로가 표준 경로보다 뒤에 위치하는 것이 바람직함
    - 그렇지 않으면 기존 유틸리티가 overload되어 사용 못할 수 있음