Lecture 15 - บรรยายครั้งสุดท้ายในภาคเรียนนี้

ฟังก์ชันใน bash

รูปแบบ

```
[function] ชื่อฟังก์ชัน ( )
{
      คำสั่งของฟังก์ชัน
      .
      .
      return ค่า            # เฉพาะในกรณีที่ต้องคืนค่าให้ผู้เรียก
}
```

ผู้เรียกผ่าน parameter ไปยังฟังก์ชัน ผ่าน command line parameter

Scope ของ ตัวแปร

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
  x=2
}

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

The script, when called as **scope.sh a b c**, gives the following output:

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

หากจะให้ x เป็น local variable ใน function เราต้องเติม local เข้าไปตอนประกาศตัวแปร x

การจัดการกับแฟ้มและ File descriptor

โพรเซสต้องทำการเปิดแฟ้มก่อน จึงจะสามารถอ่าน/เขียนแฟ้มนั้นได้ เมื่อโพรเซสเปิดไฟล์ ระบบปฏิบัติการ Unix จะกำหนดตัวเลขจำนวนเต็มเป็นตัวแทนแฟ้มนั้น ตัวเลขนี้เรียกว่า file descriptor (ตัวบอก-ราชบัณฑิตยสถาน) ซึ่งใช้เป็นดัชนีไปตารางของแฟ้มที่โพรเซสเปิดใช้งานอยู่ในขณะนั้น ตารางนี้เรียกว่า File descriptor table เมื่อเปิดแฟ้มแล้วการเขียน/อ่านทำได้โดยการอ้างอิง file descriptor เมื่อปิดแฟ้ม file descriptor ของแฟ้มนั้นจะว่างลง สามารถนำไปหมุนเวียนเพื่อใช้ในการเปิดแฟ้มต่อไปได้ โพรเซสแต่ละโพรเซสมีตารางแฟ้มที่เปิดใช้งานอยู่เป็นของตัวเอง ไม่เกี่ยวข้องกับโพรเซสอื่น

โดยปกติเมื่อมีการสร้างโพรเซส ระบบปฏิบัติการจะเปิดแฟ้มให้สามแฟ้มโดยอัตโนมัติ ได้แก่ standard input (file descriptor 0), standard output (file descriptor 1), และ standard error (file descriptor 2) ซึ่งโพรเซส เกือบทั้งหมดจำเป็นต้องใช้ และสามารถเปลี่ยนทิศทางได้

- การเปลี่ยนทิศทาง input ใช้เครื่องหมาย <
- การเปลี่ยนทิศทาง output ใช้เครื่องหมาย > หรือ 1>
- การเปลี่ยนทิศทาง error ใช้เครื่องหมาย 2>

สำหรับการเปลี่ยนทิศทางของแฟ้มอื่นๆ นอกเหนือจากนี้ไม่สู้จะมีความหมายและไม่ค่อยมีประโยชน์ในทางปฏิบัติ

การเปิดแฟ้มของเชลล์

การเปิดแฟ้มของ bash ได้โดยใช้คำสั่ง exec ซึ่งเป็นคำสั่งภายในของ bash เช่น

```
exec n> outfile
exec m < infile
```

บรรทัดแรกเป็นการเปิดแฟ้ม outfile สำหรับส่งข้อมูลออก(output) หรือเป็นการเปิดแฟ้มสำหรับเขียน และกำหนดให้มี file descriptor เป็น n ส่วนบรรทัดที่สองเป็นการเปิดแฟ้ม infile สำหรับรับข้อมูลเข้า(input) หรือเป็นการเปิดแฟ้มสำหรับอ่าน และกำหนดให้มี file descriptor เป็น m

Duplicating a file descriptor

เครื่องหมาย <& ใช้สำหรับ duplicate file descriptor ของแฟ้ม input และเครื่องหมาย >& ใช้สำหรับ duplicate file descriptor ของแฟ้ม output การ duplicate file descriptor ช่วยให้ผู้ใช้สามารถอ้างอิงแฟ้มโดยใช้ file descriptor ของแฟ้มที่เปิดอยู่แล้วได้ เช่นแฟ้ม log มี file descriptor เป็น n สามารถ duplicate ให้มี file descriptor เป็น 0 เพื่อให้การอ่าน input จากแป้นพิมพ์เปลี่ยนเป็นการอ่านข้อมูลจากแฟ้ม log ได้

```
exec n <&m
```

แฟ้มที่เปิดอยู่แล้ว (และมี file descriptor แล้ว) สามารถใช้แฟ้มสำหรับการ input และ output ได้สองแบบคือ แบบแรกเป็นการเปลี่ยนทิศทาง input/output จาก command line

- เปลี่ยนทิศทาง standard output ไปยังแฟ้มที่มี file descriptor เป็น n, >&n
- เปลี่ยนทิศทาง standard input มาจากแฟ้มที่มี file descriptor เป็น n, <&n

และ แบบที่สองคือการใช้คำสั่ง read และคำสั่ง echo ซึ่งเป็นคำสั่งภายในของเชลล์
เมื่อเชลล์มีการเรียกใช้คำสั่ง รวมทั้งฟังก์ชัน เชลล์ย่อยที่สร้างขึ้นเพื่อทำงานจะ inherit แฟ้มและ file descriptor ที่มีอยู่ไปด้วย - การปิดแฟ้มที่ใช้งานแฟ้มเสร็จแล้ว (ที่มี file descriptor เป็น n) ทำได้โดยใช้

```
exec n<&-
```

เมื่อมีการเรียกใช้ฟังก์ชันของเชลล์ชื่อ mycp โดยมีชื่อแฟ้ม 2 แฟ้มเป็น argument ฟังก์ชันจะทำการคัดลอกแฟ้มแรกไปยังแฟ้มที่สอง หากมีชื่อแฟ้มเพียงชื่อเดียว ฟังก์ชันจะทำการคัดลอกข้อมูลในแฟ้มออกทาง standard output และในกรณีที่ไม่มีการกำหนด argument เลย จะเป็นการคัดลอกข้อมูลที่ป้อนเข้ามาทาง standard input ไปยัง standard output

ตัวอย่างโปรแกรมพร้อม comment

ตัวอย่างที่ 1 – mycp

```
$ cat mycp
function  mycp ()
{
     case $# in
          0)
               # ไม่มี command line argument ใดเลย
               # กำหนดให้ file descriptor หมายเลข 3 ควบคู่ (duplicates) กับ standard
               # input และ กำหนดให้ File descriptor หมายเลข 4 ควบคู่กับ standard output
               exec 3<&0 4<&1
               ;;
          1)
               # มี argument เพียงตัวเดียว -- argument นี้คือชื่อแฟ้ม
               # เปิดแฟ้มที่กำหนด เป็นแฟ้มข้อมูลเข้า (เปิดสำหรับการอ่าน) โดยให้มี file descriptor เป็น 3
               # เนื่องไม่มีการกำหนดแฟ้มข้อมูลออก แสดงว่าผู้ใช้ต้องการคัดลอกข้อมูลในแฟ้มที่กำหนดออก
               # หน้าจอ จึงทำการควบรวม file descriptor ของแฟ้มข้อมูลออกเข้ากับ standard output
               exec 3< "$1" 4<&1
               ;;
          2)
               # มี argument สองตัว -- มีทั้งชื่อแฟ้มต้นฉบับ และชื่อแฟ้มที่จะใช้เป็นสำเนา เปิดแฟ้มต้นฉบับสำหรับการอ่าน
               # โดยกำหนดให้มี file descriptor เป็น 3 และเปิดแฟ้มสำเนาสำหรับการเขียนโดยกำหนดให้มี
               # file descriptor เป็น 4
               exec 3< "$1"  4> "$2"
               ;;
          *)
               echo "Usage: mycp [source [destination]]"     # กรณีอื่นเป็น error
               return 1
               ;;
     esac

     # เรียกโปรแกรม cat โดยให้ข้อมูลเข้ามาจากแฟ้มที่มี file descriptor เป็น 3 และ
     # ให้ส่งผลลัพธ์ออกไปยังแฟ้มที่มี file descriptor เป็น 4
     cat <&3  >&4

     # ปิดแฟ้มข้อมูลเข้าและข้อมูลออกโดยใช้ descriptors
     exec 3<&-  4<&-
}

# หากต้องการดูการทำงานของฟังก์ชันต้องมีผู้เรียก โดยส่ง command-line argument เป็น parameters ไปให้
mycp "$@"
```

ตัวอย่างที่ 2 - sortmerg

โปรแกรมต่อไปนี้ ใช้สำหรับเรียงข้อมูลในแฟ้มชนิด text file จำนวน 2 แฟ้ม และนำข้อมูลจากทั้งสองแฟ้มมาผสาน (merge) เข้าด้วยกัน และแสดงผลลัพธ์ทาง standard output

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

usage ()
{
     if [ $# -ne 2 ]; then
          echo "Usage: $0 file1 file2" 2>&1
          exit 1
     fi
}

# กำหนดไดเรกทอรีสำหรับใช้เก็บแฟ้มชั่วคราว ในกรณีที่ผู้ใช้ไม่ได้กำหนดค่าในตัวแปร TMPDIR หรือ
# กำหนด Default temporary directory เป็นไดเรกทอรี tmp ในไดเรกทอรีปัจจุบัน
# ในการใช้งานจริง โดยทั่วไปควรกำหนดเป็น /tmp # ความหมายของคำสั่งจะอธิบายในห้องเรียน
: ${TEMPDIR:=./tmp}

# ตรวจสอบจำนวน argument โดยเรียกฟังก์ชัน usage
usage "$@"

# สร้างแฟ้มชั่วคราวสำหรับใช้ในการเรียงข้อมูล
file1=$TEMPDIR/$$.file_1
file2=$TEMPDIR/$$.file_2

# เรียงลำดับข้อมูลด้วยโปรแกรม sort -- เปลี่ยนทิศทางผลลัพธ์ไปยังแฟ้มชั่วคราว
sort $1 > $file1
sort $2 > $file2

# เปิดแฟ้มชั่วคราว (ตัวแปร $file1 และ $file2) สำหรับอ่าน กำหนดให้มี file descriptors เป็น 3 และ 4
exec 3<$file1
exec 4<$file2

# อ่านข้อมูลบรรทัดแรกจากแต่ละแฟ้ม เพื่อเริ่มต้นทำงาน สังเกตการเปลี่ยนข้อมูลเข้าจากแป้นพิมพ์เป็นแฟ้มที่มี
# file descriptor เป็น 3 และ 4
read Line1 <&3
status1=$?
read Line2 <&4
status2=$?

# หากข้อมูลของทั้งสองยังไม่หมด เปรียบเทียบข้อมูลทั้งสองบรรทัด และเขียนข้อมูลที่มีลำดับมาก่อน
# จากนั้นจึงอ่านข้อมูลจากแฟ้มมาแทนบรรทัดที่เขียนแล้ว
while [ $status1 = 0 -a $status2 = 0 ]; do
     if [[ "$Line2" > "$Line1" ]]; then              # สังเกตการใช้วงเล็บซ้อน [[ ... ]]
```

```
          echo -e "1.\t$Line1"                          # สังเกตเงื่อนไขของคำสั่ง test
          read -u3 Line1                          # สังเกต option -u ของ read
          status1=$?
     else
          echo -e "2.\t$Line2"
          read -u4 Line2
          status2=$?                              # สังเกตการใช้ $? เพื่อนำไปใช้เป็น end-
of-file

     fi
done

# เมื่อออกจาก while loop แสดงว่าข้อมูลในแฟ้มใดแฟ้มหนึ่ง หรือทั้งสองแฟ้มหมดลง จากนั้นจึงอ่านข้อมูล
# จากแฟ้มที่เหลือ ส่งออกทาง output จนหมด -สังเกตการใช้ประโยชน์จาก zero-or-more ของ while

# แฟ้มแรก - File1:
while [ $status1 -eq 0 ]; do
     echo -e "1.\t$Line1"
     read -u3 Line1
     status1=$?
done

# แฟ้มที่สอง - File2:
while [ $status2 -eq 0 ]; do
     echo -e "2.\t$Line2"
     read -u4 Line2
     status2=$?
done

# ปิดแฟ้ม และกำจัดแฟ้มชั่วคราว Close and remove both input files
exec 3<&- 4<&-
rm -f $file1 $file2
exit 0
```

คำถาม

---

1. คำสั่ง : ${TEMPDIR:=./tmp} ใช้สำหรับทำงานใด?
2. คำสั่ง read Line1 <&3 และ read -u3 Line1 มีการทำงานเหมือนกันหรือไม่? จะทดสอบได้อย่างไร?
3. หากเปลี่ยนคำสั่ง

```
        if [[ "$Line2" > "$Line1" ]]; then
```

เป็น

```
        if [ "$Line2" > "$Line1" ]; then
```

ผลลัพธ์ของโปรแกรมเหมือนกันหรือไม่? อย่างไร?

4. หากเปลี่ยนโครงสร้าง

```
    while [ $status1 -eq 0 ]; do
         ... ...
    done
```

เป็นโครงสร้าง `until..do..done` การทำงานของโปรแกรมจะเหมือนเดิมหรือไม่? อย่างไร?

1. คำสั่ง : `${TEMPDIR:=./tmp}` ใช้สำหรับทำงานใด?
2. คำสั่ง `read Line1 <&3` และ `read -u3 Line1` มีการทำงานเหมือนกันหรือไม่? จะทดสอบได้อย่างไร?
3. หากเปลี่ยนคำสั่ง

```
if [[ "$Line2" > "$Line1" ]]; then
```

เป็น

```
if [ "$Line2" > "$Line1" ]; then
```

ผลลัพธ์ของโปรแกรมเหมือนกันหรือไม่? อย่างไร?

4. หากเปลี่ยนโครงสร้าง

```
while [ $status1 -eq 0 ]; do
    ... ...
done
```

เป็นโครงสร้าง `until..do..done` การทำงานของโปรแกรมจะเหมือนเดิมหรือไม่? อย่างไร?

Expanding null and unset variables

เมื่อ `shell script` ทำงาน, ตัวแปร `$name` หรือ `${name}` จะถูกแทนที่ด้วย "ค่า" ในกรณีที่ตัวแปรนั้นไม่ได้ถูกกำหนดค่ามาก่อน หรือมีการยกเลิกค่าด้วยคำสั่ง `unset` จะทำให้ค่าของตัวแปรเป็น `null` ซึ่งผู้เขียนโปรแกรมสามารถดำเนินการกับตัวแปรที่มีค่าเป็น `null` ได้สามแบบคือ

- ใช้ค่า `default` สำหรับตัวแปรนั้น (:-)
- กำหนดค่า `default` ให้แก่ตัวแปรนั้น (:=)
- แสดง `error message` (:?)

```
:- Use a default value
----------------------
     ${name:-default}

     if  name is null or unset  then
         expand default and use the expanded value in place of name
     else
         use name
     end{if}
```

การ `expand default` ไม่มีผลทำให้ค่าของตัวแปรเปลี่ยนไป ตัวแปรนั้นยังคงอยู่และมีค่าเป็น `null` ต่อไป

```
:= Assign a default value
-------------------------
     ${name:=default}
```

ไม่ว่าตัวแปรนั้นจะมีค่ามาก่อนหรือไม่ตาม

```
     expand default and assign the expanded value to name
     use name

     :(null) buitin
```

หากต้องการทดสอบก่อนว่าตัวแปรนั้นเป็น null หรือไม่ ทดสอบด้วย :(null)

```
:  ${name:=default}

if  name is null or unset  then
    expand default and assign the expanded value to name
    use name
end{if}
```

```
:? Display an error message
---------------------------
     ${name:?message}
```

เป็นพียงการแสดง error ไม่มีผลให้ shell หยุดทำงาน ในกรณีที่ไม่กำหนด message จะได้ default error message ของ shell

```
$ echo ${var:?}
-bash: var: parameter null or not set

$ echo ${var:?$(date +%T) error, variable not set}
-bash: var: 05:54:23 error, variable not set
```