第 2 回 UNIX ゼミ

ゼミ担当者　：　吉形 允晴，澁谷 翔吾
指導院生　：　山川 望，山崎 弘貴，木田 清香，渡辺 崇文
開催日　：　2007 年 5 月 18 日

1　はじめに

本ゼミでは，研究活動での UNIX の使用において，最低限必要となる知識および操作のスキル取得を目的とする．本研究室では研究活動において並列計算機を使用する．並列計算機を利用，管理するためには Linux の知識が必要となる．本ゼミでは，それらを利用できるように，シェル，コマンド，およびシェルスクリプトについて学ぶ．

2　シェル (Shell)

2.1　シェルの概要

UNIX において，ユーザは OS の核となるカーネルに直接アクセスして操作することができない．このため，ユーザーが入力したコマンドを解釈して，ユーザとカーネルの橋渡しをするシェルが必要となる．シェルはユーザの操作を受付けて与えられた指示をカーネルに伝える役割を持つ．シェルにより，ユーザはカーネルと対話しているように OS を操作できる．

シェルには主にコマンドインタプリタとスクリプト言語の 2 つの機能がある．UNIX を利用するにあたって，コマンド入力をシェルが解釈しカーネルに伝えるというこの一連の動作は，インタプリタとしてシェルが動作している状態である．

このように対話的に操作を行う一方で，シェルにはバッチ処理を行わせるプログラミングの機能がある．シェルの記述法に基づいてスクリプトを書くことによって，複数のコマンドを 1 つに組み合わせることができる．また，繰り返しや分岐などの制御構造を持つことで，単なる順次処理だけではなく，より複雑な処理を記述できる．これら 2 つの機能から，シェルは UNIX を利用する上で必要不可欠なものである．

2.2　シェルの有用性

シェルの有用性としては，以下の 2 点が挙げられる．

- ルーティンワークを単純化できる
- 独自に組合わせた特別なコマンドを作成できる

ルーティンワークとは同じ処理を繰り返す作業である．シェルでは処理手順をシェルスクリプトとして記述することで, 複雑な作業，処理を自動化することができる．また頻繁に利用する処理などを独自に記述してコマンド化しておくこともできるため，ユーザの作業効率を上げることができる．

このように作成したプログラムは, シェルスクリプトの汎用性から, ほとんどの UNIX 上で同じように動作できるという利点もある．

2.3　代表的なシェル

シェルにはいくつか種類があるが，ここでは代表的なものを 4 つ紹介する．

- sh(Bourne Shell)
  現在利用できる最も古いシェルで，B シェルとも呼ばれている．このシェルは AT&T のベル研究所で開発され，開発者の 1 人である Steven Bourne 氏に因んで名付けられた．様々なシェルの中で共通項的な位置にあり，ほとんど全ての UNIX で利用できる標準的なシェルである．

- bash(Bourne Again Shell)
  B シェルを拡張したシェルである．ユーザーインターフェイスとしての機能を強化するため，ヒストリー機能，エイリアスなどが追加されている．Linux では，デフォルトのシェルとしてこの bash を使えるようになっている．本ゼミでは，bash を基に進めていく．

- C シェル (csh)
  シェルプログログラミングの方法が C 言語の構造に似ているため,「C Shell」と呼ばれている．B シェルにはない便利な機能が多くあるが，B シェルと C シェルではかなり非互換部分がある．

- Korn シェル (ksh)
  このシェルは，David G. Korn によって開発され，B シェルと互換性がある．C シェルのよう，組み込み演算機能，配列，文字列操作などのプログラミング機能がある．

3　シェルコマンド

シェルコマンドは第 2.1 節で説明したシェルのコマンドインタプリタの機能である．第 1 回 UNIX ゼミで扱っ

た「mv」,「cd」などのコマンドもこれに当たる．本節以降で説明するスクリプト言語に関して，出力やファイル操作などを行う際に便利となるコマンドを紹介する．

3.1 基本コマンド

下記に各コマンドとその特徴，実行例を示す．

- echo
  文字列をそのまま出力する．

```
$ echo 文字列
```

ex:echo

```
$ echo aiueo
aiueo
```

- cat
  ファイルの内容を出力する．

```
$ cat ファイル名
```

ex:cat

```
$ cat example1.txt
kakikukeko
sasisuseso
```

- grep
  ファイル内の文字列を検索する．

```
$ grep 検索文字列 ファイル名
```

ex:grep

```
$ grep kaki example1.txt
kakikukeko
```

- sed
  ファイル内の文字列を置換して出力する．

```
$ sed s/置換前文字列/置換後文字列/ ファイル名
```

ex:sed

```
$ sed s/kakikukeko/aiueo/ example1.txt
aiueo
sasisuseso
```

- expr
  数値演算を行う．
  基本の演算子は和「+」差「-」積「*」商「/」である．

```
$ expr 数式
```

ex:expr

```
$ expr 5 + 3
8
$ expr 8 - 2
6
```

- cut
  ファイルの一部を出力する．下記のようにxとyに数値を入れることにより，ファイル内の各行のファイルのx文字目からy文字目までを表示する．

```
$ cut -c x-y ファイル名
```

ex:expr

```
$ cut -c 1-3 sample1.txt
kak
sas
```

これらはファイルを開かずに目的に応じて，ターミナルなどに結果を出力することができる．これらのコマンドを次節に述べるシェルの機能と組み合わせることで，効率的な作業が行える．

4 シェルの便利な機能

4.1 リダイレクト機能

リダイレクトとは「入出力の切り替え」という意味で，標準入力をキーボード以外のファイルに指定したり，標準出力をディスプレイ以外のファイルに指定することである．Table 1にリダイレクト機能を用いるための記号を示し，リダイレクト機能の簡単な例を示す．

Table 1　リダイレクト機能

コマンド	機能
<	標準入力のリダイレクト
>	標準出力のリダイレクト
>>	標準出力をファイルへ追加する
>&	エラー情報もファイルへ出力する
>>&	上記の追記版
2>	エラー情報のみをファイルへ出力

ex:redirect

```
$ echo Hello! > out1.txt
$ cat out1.txt
Hello!
```

この例では，標準入力の内容を指定したファイルに出力した．指定したファイルに出力すると同時に，画面出力も行いたい場合は，tee コマンドを用いる．

```
$ コマンド | tee 出力ファイル名
```

ex:tee

```
$ echo hello! | tee out2.txt
hello!
$ cat out2.txt
hello!
```

この例では，標準出力とファイル出力を同時に行っている．

4.2 コマンドの連続実行

4.2.1 条件なし連続実行

基本的に複数のコマンドを連続実行するには，次の記述を用いることで行える．

```
$ コマンド 1; コマンド 2; …
```

ex:連続実行

```
$ pwd;ls
/home/mitsuharu
example1.txt sennryaku smap.sh test
```

この例では，pwd コマンドと ls コマンドを連続して実行している．

4.2.2 条件あり連続実行

条件ありの連続実行では，前の実行コマンドが正常終了，異常終了かを判断して次の処理を進める．

コマンドを下記のように「||」で区切ることにより，前のコマンドが異常終了した場合に次のコマンドを実行する．

```
$ コマンド 1||コマンド 2|| …
```

また「&&」で区切ることにより，前のコマンドが正常終了した場合に次のコマンドを実行する．

```
$ コマンド 1&&コマンド 2&& …
```

連続実行の利点として，出力を一括してファイル出力できるため，リダイレクトを用いる場合に逐次追記する必要がないなどが考えられる．

4.3 出力の利用

シェルの機能としてコマンドの出力を利用できる．出力を利用するためのコマンドを$() で囲むことにより，その出力がコマンドや引数として動作する．

```
$  $(コマンド)
```

ex:出力の利用

```
$ cat example2.txt
ls
$ $(cat example2.txt)
example1.txt sample2.txt sennryaku
smap.sh test
```

4.4 パイプ

シェルにはパイプという機能があり，あるコマンドの標準出力を別のコマンドの標準入力に連結することが可能である．

```
$  コマンド 1|コマンド 2|コマンド 3…
```

上記のように入力することによりコマンド 1 の出力をコマンド 2 の引数として用いられる．

ex:パイプ

```
$ cat example3.txt
c
b
a
$ cat example3.txt|sort
a
b
c
```

この例では，cat コマンドで「c，b，a」を出力し，それをパイプ機能を用いて，sort コマンドでソートし「a，b，c」を出力している．

4.5 正規表現

正規表現とは，文字列パターンを指定できる表現方法である．任意の文字を「.」，直前文字の繰返しを「*」などの記号で表現できるため，あらゆる文字列のパターンの指定ができる．そのため，コマンドによる検索や置換など，複数のファイルやディレクトリを同時に指定し操作することが可能である．Table 3 に代表的な正規表現の記号を示す．

Table 2　正規表現

記号	意味
.	任意の 1 文字または 0 文字
*	直前文字の 0 回以上の繰返し
+	直前文字の 1 回以上の繰返し
[ ]	列挙された任意の 1 文字
^	行頭を表す
$	行末を表す

ex:正規表現

```
$ cat a.txt
bbbb
99dlakj
$ sed s/^[0-9][0-9]*// a.txt
bbbb
dlakj
```

この例では，正規表現を用いて，文字列の先頭が数字の場合は，その数字が続く限り，それらの数字を消去している．

5　シェル変数と環境変数

5.1　シェル変数

シェル変数とは，動作中のシェル内でのみ有効な変数である．ユーザはシェル変数を利用することによって，シェルやコマンドの動作を操作できる．シェル変数は以下のように入力することにより設定できる．シェル変数名は，英文字あるいはアンダースコアで始まり，その後ろに 0 個以上の英数字，あるいはアンダースコアの並びで定義する．

```
$ set シェル変数名
```

また，下記のように「＝」を使用することによりシェル変数に値を格納することもできる．

```
$ シェル変数名=値
```

設定したシェル変数の一覧は「set」と入力することで確認できる．一覧の表示では，新規に設定した変数が降順に表示される．また，既存の変数名を入力することにより，その変数の値を確認できる．シェル変数には，ユーザが設定したシェル変数以外にも，あらかじめ用意されている定義済みシェル変数がある．定義済みシェル変数を変更することによって，ユーザのシェル環境を自由に変更することができる．

一度設定したシェル変数は次のように入力することで削除することができる．

```
$ unset シェル変数名
```

シェル変数は，宣言をしたシェルのみで有効であり，使用しているシェルを終了させると，値は消えてしまう．

5.2　環境変数

環境変数とは，そのシェルから起動されたプロセスに受け継がれる変数である．上述したシェル変数は，設定したシェルの中でのみ有効である．そのため，他のシェルやアプリケーションからは利用できない．設定したシェル変数を別に起動したシェル内で使用するには，シェル変数を環境変数にする必要がある．環境変数を定義するコマンドは次の通りである．

```
$ export 環境変数名
```

シェル変数と同様に，ユーザが設定した環境変数以外にも，あらかじめ用意されている定義済み環境変数がある．これらの環境変数の一覧は次のように入力することで確認できる．

```
$ env
```

5.3　エイリアス

エイリアス機能を使うと，コマンドの別名（エイリアス）を登録できる．ユーザは頻繁に用いるコマンドや，長いコマンドを簡単な名前のコマンドで登録することで，効率的な作業が行える．エイリアスを登録するには，alias コマンドを用いる．

```
$ alias 新コマンド=コマンド名
```

エイリアス機能を使って，初めからコマンドとして存在する文字列をエイリアスに指定してしまうと，初めから存在するコマンドを使用できなくなってしまう．そこで，alias コマンドを用いる前に，登録する文字列がコマンドとして存在しないことを確認しておく必要がある．コマンドの存在を確認するには which コマンドを用いて，「which コマンド名」とすることで確認できる．which コマンドを実行して，「Command not found」と表示されたら，その文字列はコマンドとして存在しないので，エイリアスとして使用できる．

ex:alias

```
$ alias lsl="ls -l"
$ lsl
total 1
-rw-r--r-- 1 Shogo None 2 May 9 17:10
sample1.txt
```

この例では，"ls -l"をエイリアス機能を使って"lsl"で実行できるようにしている．

6　シェルスクリプト

シェルは，シェルスクリプトと呼ばれる独自のプログラミング言語を解釈し実行する機能を備えている．これにより，バッチ処理のような一連の仕事を，一つのコマンドでできるようにし，処理の自動化が可能となる．本章では，シェルスクリプトの基本文法を中心に説明する．

6.1 シェルスクリプトの基本

本節では，シェルスクリプトの記述方法，実行方法について説明する．

6.1.1 シェルスクリプトの記述方法

以下にシェルスクリプトの記述方法を例を基に説明する．

ex:sample1.sh

```
$ #!/bin/bash
date
echo "I am $USER"
echo "Hello!"
```

1行目は，スクリプトを実行するのに使用するプログラムを指定している．ここでは，bash シェルを起動している．2行目からは，スクリプトで実行したいコマンドを記述している．例では，日付，ユーザ，メッセージを順に表示するようになっている．

6.1.2 シェルスクリプトの実行方法

作成したシェルスクリプトを実行するには，いくつか方法がある．

```
$ .  スクリプトファイル
```

```
$ source スクリプトファイル
```

```
$ bash  スクリプトファイル
```

いずれのコマンドも，同様にスクリプトを実行できる．

sample1.sh の実行結果

```
$ .  sample1.sh
Fri May 11 23:07:09 2007
I am Shogo
Hello!
```

この例では，date コマンドを用いて日付を表示し，環境変数に格納されたユーザ名を表示している．また，スクリプトをシェルコマンドとして使用したい場合は，alias コマンドを用いて，「alias sample1.sh=". sample1.sh"」のようにエイリアスを定義すればよい．

ex:スクリプトのコマンド化

```
$ alias ex1=".  sample1.sh"
$ ex1
Fri May 11 23:12:42 2007
I am Shogo
Hello!
```

6.1.3 特別なシェル変数

シェル変数には，あらかじめ用意された特別な変数があり，その変数を使用，参照することができる．Table 3 に代表的な変数の一覧を示す．

Table 3　リダイレクト機能

変数	説明
$n	スクリプトに渡された n 番目の引数
$#	与えられた引数の個数
$@	$0 以外の全引数

これら特殊な変数を用いたシェルスクリプトの例を以下に示す．

ex:sample2.sh

```
#!/bin/bash
echo "$0"       #スクリプト名
echo "$1,$2"    #1 番目と 2 番目の引数
echo "$#"       #引数の個数
echo "$@"       #全引数
```

ex:実行結果 sample2.sh

```
$ ./sample2.sh shogo mitsuharu shibutani
sample2.sh
shogo,mitsuharu
3
shogo mitsuharu shibutani
```

この例では，スクリプト実行と同時に引数を渡し，それらを表示している．

6.2 シェルスクリプトの制御構文

シェルスクリプトは，他のプログラミング言語のように制御構文がある．本節ではシェルスクリプトで使用する制御構文について説明する．

6.2.1 条件式

制御構文では，条件判定がよく用いられる．制御構文で用いる条件判定で，文字列の比較や整数値の大小比較を行う場合，以下のように記述する．

```
$ test 条件式
```

```
$ [ 条件式 ]
```

前者は記述方法は，test コマンドを用いた条件判定であり，条件が真の時は 0 を返し，偽の時は 1 を返す．また，後者の記述方法は条件式の前後にスペースを空けなければエラーとなるので注意する．

条件判定には次のような演算子が用いられる．文字列比較演算子を Table 4，ファイル属性演算子を Table 5，論理演算子を Table 6，数値評価演算子を Table 7 にそれぞれ示す．

Table 4　文字列比較演算子

記号	意味
文字列 1 = 文字列 2	2 つの文字列が一致する
文字列 1 != 文字列 2	2 つの文字列が一致しない
-n 文字列	文字列が null でない
-z 文字列	文字列が null である

Table 5　ファイル属性演算子

記号	意味
-d file	指定ファイルがディレクトリである
-e file	指定ファイルが存在する
-f file	指定ファイルが普通のファイルである

Table 6　論理演算子

記号	意味
!条件	条件が偽である
条件 1 -a 条件 2	2 つの条件がともに真である
条件 1 -o 条件 2	2 つの条件のどちらかが真である

Table 7　数値評価演算子

記号	意味
数値 1 -lt 数値 2	数値 1 が数値 2 より小さい
数値 1 -le 数値 2	数値 1 が数値 2 以下
数値 1 -eq 数値 2	数値 1 と数値 2 が等しい
数値 1 -ge 数値 2	数値 1 が数値 2 以上
数値 1 -gt 数値 2	数値 1 が数値 2 より大きい
数値 1 -ne 数値 2	数値 1 と数値 2 が等しくない

test コマンドを用いて,「/home/Shogo/sample1.sh」がディレクトリであるか調べた結果を以下に示す．

ex:ファイル属性演算子

```
$ test -d /home/Shogo/sample1.sh
$ echo $?
1
```

”$?”は直前のコマンドの終了ステータスを意味する．「/home/Shogo/sample1.sh」はファイルであるため，1 を出力する．

6.2.2　if 文

条件分岐である if 文は，if，elif，else を上から順に判定し条件が成立となれば，そのコマンドを実行する．test を用いて真偽の判定も可能である．

if 文の構造

```
if [条件]
then
      コマンド 1
elif [条件]
then
      コマンド 2
else
      コマンド 3
fi
```

ex:if.sh

```
#!/bin/bash
tmp-10
if[ $tmp -lt $1]
then
echo "10 < "  $1
elif[ $tmp -ge $1 ]
then
echo "10 >" $1
fi
```

ex:if.sh 実行結果

```
$ . if.sh 5
10> 5

$ . if.sh 13
10< 13
```

この例では，引数とシェル変数 tmp との大小を判別している．ここで，シェル変数 tmp は 10 であり，引数を 5 とすると「10>5」となり，13 とすると「10<13」と不等号で表示される．

6.2.3 for 文

for 文は下記のように記述することで do と done で挟まれたコマンドを繰り返す．リストには「引数 1 引数 2 引数 3 …」といったように引数が複数用いられ，繰り返し毎に 1 つ 1 つ変数に代入され引数がなくなるまで処理を繰り返す．シェルスクリプトによる繰り返し文は，ファイルを引数として用いることもでき，C 言語などの for 文とは異なり，複数のファイルを処理することによく使われる．

for 文の構造

```
for 変数 in  リスト (引数...)
do
      コマンド
done
```

ex:for.sh

```
#!/bin/bash
for x  in   1 2 3
do
      echo $x
done
```

ex:for.sh 実行結果

```
\$ .for.sh
1
2
3
```

この例では，引数である「1」「2」「3」が順に変数 tmp に格納される．引数がなくなった時点で，繰り返しは終了する．実行結果では，echo コマンドの繰り返しにより，「1，2，3」が順に表示される．また，変数は「$」を前につけることにより，値を参照できる．

6.2.4 while 文

while 文は条件の真偽により処理を繰り返すことに用いられる．条件が真の間 do-done 間を繰り返す．偽ならば処理終了となる．

while 文の構造

```
while [条件式]
do
      コマンド
done
```

ex:while.sh

```
#!/bin/bash
tmp=50
while [ $tmp -gt $1]
do
echo $tmp
tmp=$(expr $tmp - 5)
done
```

ex:while.sh 実行結果

```
$. while.sh 20
50
45
40
35
30
25
```

この例では，シェル変数 tmp と比較して「tmp>引数」となる間，処理を繰り返すというシェルスクリプトである．while 分の中では，シェル変数 tmp の値を表示し，その後で，シェル変数 tmp の値を 5 減算するという操作をしている．

6.3 シェル関数

シェルスクリプトでも関数の作成が可能である．シェル変数に関数を定義することでコマンドのようにも使用することが可能である．シェルスクリプト実行時に引数を記入することで，関数内部にも引数を渡すことが可能である．

シェル関数

```
関数名 (){
      コマンド
```

ex:file_func

```
#!/bin/bash
func(){
echo"$1+$2=$(expr $1 + $2)"
echo"$1-$2=$(expr $1 - $2)"
echo"$1*$2=$(expr $1 \* $2)"
echo"$1/$2=$(expr $1 / $2)"
echo"$1\%$2=$(expr $1 \% $2)"
}
```

ex:file_func 実行結果

```
$source file_func
$ func 8 2
8+2=10
8-2=6
8*2=16
8/2=4
8\%2=0
```

この例では，四則演算を file_func ファイルの func 関数に処理を行わせている．func 関数はシェル変数を 2 つ用いているので，func の後に 2 つ引数をつけている．