第 2 回 UNIX ゼミ

ゼミ担当者　：　村上　耕平，木田　清香，千田　智治
指導院生　：　梶原　広輝，柴田　優，山川　望
開催日　：　2006 年 5 月 17 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．本研究室では最適化の研究に並列計算機を用いることが多く，その際に，並列計算機の利用および管理のための Linux の知識が必要不可欠となる．そこで，本ゼミではそれらを利用できるように，シェルやコマンド，シェルスクリプトなどについて学ぶ．

2　シェル (Shell) とは

2.1　UNIX とシェル

UNIX は，ユーザが「コマンド」を入力し，その結果を得るということを繰り返し操作することで利用できる．コマンドとは，UNIX に何か行わせる際に用いる道具のことであり，コマンドを「シェル」が解釈することで UNIX に命令を伝えることが出来る．シェルは，コマンドをはじめとして，ユーザと UNIX の橋渡しをしてくれるものである．そのため，シェルは UNIX を利用する上で無くてはならないものである．

2.2　シェルの概要

シェルには 2 つの機能がある．1 つはコマンドインタプリタとしての機能であり，もう 1 つはスクリプト言語としての機能である．端末画面からコマンドを入力して結果をもらい，また別のコマンドを入力し，といった一連の操作は，インタプリタとして（対話的に）シェルが動作している状態である．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．シェルの記述法にしたがってスクリプトを書くことで，複数のコマンドを 1 つに組み合わせたり，条件によって処理を分けたりすることが可能である．シェルを使ったプログラムでは，豊富に用意されている UNIX のコマンドを自由に組み合わせることで，複雑な処理を簡単に実現することができる．

2.3　代表的なシェル

シェルには，いくつかあるが，ここでは代表的なものを 3 つ紹介する．

- sh(Bourne shell)
  標準的なシェルで，B シェルとも呼ばれる．これを改良したのが bash(Bourne Again Shell) であり，Linux ではデフォルトのシェルとして bash シェルを使えるようになっている．なお，本ゼミではこの bash シェルを基に進める．

- C シェル (csh)
  C 言語に似たスクリプトが利用できるシェルである．csh を拡張した tcsh もあり，Linux で C シェルとして使われているのは tcsh である．

- Korn シェル (ksh)
  Korn シェル (ksh) は，Bourne シェルを拡張したものである．ksh に tcsh の機能を取り入れた zsh というシェルもある．

2.4　シェルの有用性

シェルの有用性として，次の 2 点が挙げられる．

- ルーティンワークを単純化できる．

- 特別な処理をするコマンドを新たに作成できる．

前者は，できるだけ自分のやることを単純化したり自動化した方が効率的であるが，そのために C 言語などでプログラムをわざわざ書くメリットはないと判断できるものを，シェルスクリプトで記述することで容易に達成できる．

後者の場合は，シェルスクリプトの汎用性が理由に挙げられる．コマンドがシェルスクリプトとして記述されており，一度作ってしまえばたいていの UNIX 上で同じように動作できるということである．もちろんシステムによる違いは出てくるが，そういった違いや，環境の違いによる修正部分も，シェルスクリプトに含めてしまうことができる．

3　コマンド群

本章では，基礎的なコマンドを紹介する．1 つ 1 つのコマンドの動作は非常にシンプルであるが，第 4 章で取り上げるシェルスクリプトでは非常に役に立つコマンドである．

3.1　文字列の表示 (echo)

文字列を表示させる．

```
$ echo 文字列
```

echo コマンドの後にくる文字列をそのまま出力する．

```
$ echo -n 文字列
```

通常は出力文字の後ろに改行が入るが，-n オプションをつけると改行しない．

3.2 ファイルの内容表示とファイル連結 (cat)

ファイルの内容を表示する．ファイルは複数指定し，連続して表示可能である．

```
$ cat ファイル名
```

以下のコマンドにより複数のファイルの内容を連結できる．

```
$ cat ファイル 1 ファイル 2 > 出力ファイル
```

連結するファイルは，複数指定可能でり，上記のコマンドではファイル 1 の内容の後にファイル 2 の内容が続いたものが，出力ファイルとして出力される．

3.3 文字列検索 (grep)

文字列の探索を行う．

```
$ grep 検索文字列 ファイル名
```

指定ファイルの中から検索文字列が含まれる行を出力する．指定するファイルは複数続けることが可能である．

```
$ grep -e 文字列 -e 文字列 ファイル名
```

検索文字列の複数指定可能で，その際には-e オプションを検索文字列の前につける必要がある．また，‘-’(ハイフン) から始まる文字列を検索する際にはオプションと誤認される．このときも，文字列の前に-e オプションを付けることで検索が可能となる．

3.4 文字列置換 (sed)

ファイル内の文字列を 1 行ずつ読みこんで文字列の置換を行う．

```
$ sed s/元文字列/置換後の文字列/ ファイル名
```

ファイル内の文字列に，元文字列と同じ文字の並びがあれば置換後の文字列と置換する．置換後の文字列を空にすることで，元文字列を削除することができる．

3.5 数値演算 (expr)

数値演算を行う．

```
$ expr 数式
```

数式は基本的に他の C 言語のようなプログラミング言語と同じであるが，乗算の「*」は特殊文字として扱われるため,「¥ *」と記述する必要がある．

ex: expr

```
$ expr 106 + 20
126
$ expr 106 - 20
86
$ expr 106 ¥* 20
2120
$ expr 106 / 20
5
$ expr 106 6
```

4 シェルの便利な機能

4.1 シェル変数と環境変数

4.1.1 シェル変数

シェルでは，シェル変数を用いてシェルの内部に値を保持することが出来る．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が終了すると値は消えてしまうので注意が必要である．

```
$ 変数名=値
```

シェル変数に使用できる文字は英数字および「_」で，先頭は英字を使用しなくてはならない．また変数名と値の間の「=」の前後にスペースを入れることはできないので注意する．また，シェル変数の値は “$変数名” で参照することができる．

現在設定されているシェル変数の一覧は set コマンドにより表示させることができる．

```
$ set
```

自分で設定した変数以外にも，あらかじめ値がセットされている変数があることも確認できる．シェル変数を削除するには，unset コマンドを用いる．

```
$ unset シェル変数名
```

4.1.2 環境変数

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．もし，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．環境変数にするには，export コマンドを用いる．

```
$ export 変数名
```

新たにシェル変数とその値を定義して，環境変数とすることも可能である．

```
$ export シェル変数=値
```

現在設定されている環境変数の一覧は env コマンドによって表示できる．シェル変数と同様，あらかじ設定されている環境変数があることが確認できる．

```
$ env
```

また，export コマンドによりシェル変数を環境変数に設定したとしても，ログアウトすると設定は消えてしまう．そのため，毎回使用するような環境変数はホームディレクトリにある『.bashrc』などに記述しておく必要がある．

4.2 エイリアス

UNIX ではコマンドに別名 (エイリアス) を付けることができる．これはエイリアス機能であり，エイリアスを定義すると，長いコマンドを簡単な名前のコマンドで実行することが可能となる．

```
$ alias 新コマンド=コマンド名
```

エイリアスの定義の簡単な例を示す．

ex: alias
```
$ alias cde="cd /etc"
$ pwd
/usr/tmp
$ cde
$ pwd
/etc
```

alias コマンドにより，設定したエイリアスはシェル変数と同様に設定したシェルのみで有効となる．したがって，毎回必要な設定は，ホームディレクトリにある『.bashrc』などに記述する必要がある．

4.3 リダイレクト機能

通常，文字列の入力が必要なコマンドを実行する場合，文字列をキーボードから入力しディスプレイに出力する．この入力文字列がファイルに格納されている文字列を用いる場合，リダイレクト操作により，簡単に入力操作を行うことができる． また，通常ウィンドウの画面上に表示される出力をファイルへの出力とすることも可能である．Table 1 に，リダイレクト機能を用いるための記号を示し，リダイレクト操作の簡単な例を示す．

Table 1　リダイレクト機能

コマンド	機能
<	標準入力をファイルとする
>	標準出力をファイルとする
>>	標準出力をファイルへ追記する
>&	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 ahaha > out1.txt
$ more out1.txt
ahaha
$ echo ihihi >> out1.txt
$ cat out1.txt
ahaha
ihihi
```

また，リダイレクトを使用してファイルに出力したとき，ファイルに出力された内容がわからないので不便である．そのような場合には tee コマンドを用いて，ファイル出力と画面出力を同時に行うことができる．

```
$ コマンド | tee 出力ファイル名
```

ex: tee
```
$ ls | tee out2.txt
dir1 dir2 file1 file2 file3
$ more out2.txt
dir1 dir2 file1 file2 file3
```

4.4 コマンドの連続実行

コマンドをまとめて実行したい場合，以下のような記述によりコマンドの連続実行が可能である．

4.4.1 条件なしの連続実行

```
$ コマンド１ ; コマンド２ ; …
```

以下のようにコマンドをまとめる．

```
$ (コマンド１
>  コマンド２
>    …
>  コマンド N)      もしくは
$ { コマンド１
>  コマンド２
>    …
>  コマンド N}
```

上記のどちらの記述でもほぼ同様の動作が得られるが，小括弧 (「(」,「)」) では，括弧内が別の環境で実行されることとなる．大括弧 (「{」,「}」) では現在の環境での実行となる．コマンドをまとめる利点として，コマンド実行時の出力を一括してファイルに出力する際などに，簡潔に記述できることがあげられる．

ex: コマンドをまとめる

```
$ (echo Hello
> pwd
> cd /tmp
> pwd) > out
$ more out
Hello
/home/tomoharu
/tmp
$ pwd
/home/tomoharu
$ { echo Goobye
> cd /tmp
> }
Goobye
$ pwd
/tmp
```

4.4.2 条件つきの連続実行

```
$ コマンド1 && コマンド2 && …
```

コマンド 1 が正常に実行できればコマンド 2 を実行し，コマンド 2 が正常に実行できればコマンド 3 を実行するといったように，1 つ前のコマンドが正常実行できた場合，次のコマンドが実行される．

```
$ コマンド1 || コマンド2 || …
```

コマンド 1 の実行が失敗した場合，コマンド 2 を実行するといったように，1 つ前のコマンドの実行が失敗したときに，次のコマンドを実行する．

4.5 コマンド置換

シェルでは，コマンドの実行結果を文字列として扱い，その文字列を引数やコマンドとして使用することができる．

```
$(コマンド)      もしくは
‘コマンド‘
```

上記のどちらの記述でも同様の動作が得られるが，バッククォーテーション (‘) はシングルクォーテーション (’) と間違えやすいため，利用には注意が必要である．また，実行結果の文字列を変数に代入することも可能である．簡単な例を以下に示す．

ex: コマンド置換

```
$ more tempfile
ls
$ $(more tempfile)
dir1 dir2 file1 file2 file3
```

4.6 パイプとフィルタ

1 つ前のコマンドの出力が次のコマンドの入力となるとき，パイプ処理により直接データの受け渡しができる．

```
$ コマンド1 > tempfile
$ コマンド2 < tempfile

$ コマンド3 > tempfile1
$ コマンド4 < tempfile1 > tempfile2
$ コマンド5 < tempfile2
```

これをパイプ処理で行うと，以下のような記述により一度の記述で行うことができる．

```
$ コマンド1 | コマンド2
$ コマンド3 | コマンド4 | コマンド5
```

4.7 正規表現

正規表現と呼ばれる特殊な記号を用いることで，1 つのコマンドで複数のファイルやディレクトリを同時に操作することが可能となる．コマンドの引数に正規表現を含むファイル名を指定した場合，コマンドが引数を処理する前に，シェルが正規表現を解釈し，複数の引数に展開した上でコマンドに受け渡す．正規表現には Table 2 のようなものがある．

Table 2 正規表現

記号	意味
*	0 文字以上の任意の文字列
?	任意の 1 文字
[ ]	[ ] 内のどれか 1 文字
{ }	{ } 内の ‘,’ で区切られたどれかの文字列

4.8 その他のプログラミング言語

- AWK

  AWK はテキスト形式で記述されたデータを処理する，プログラミング言語で，インタプリタ型の言語

なのでシェルプログラミングなどと組み合わせて使うことが出来る．

- Perl

  Perl はテキスト形式のデータ処理と，プロセス処理までを可能としたインタプリタ型の言語である．一般に，インターネットなどで見かける CGI プログラムを記述する場合などに用いられている．

5 シェルスクリプト

シェルは，ユーザが入力したコマンドを実行するためにユーザとカーネルの橋渡しをする機能のほかに，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．これは Windows におけるバッチ処理のようなものであり，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．本ゼミでは，シェルスクリプトの基本文法を中心に説明する．

5.1 シェルスクリプトの基本

シェルスクリプトの記述方法および実行方法について説明する．

5.1.1 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の例を示す．

ex: sample1.sh

```
#!/bin/bash
date                    #日付
echo "I am $USER"       #ユーザ
echo "Hello!"           #メッセージ
```

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，メッセージを表示するコマンドが書き並べてある．シェルスクリプトでは，行頭に “#”をつけることでその行をコメントアウトすることができる．

5.1.2 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．上記のようなコマンド+スクリプトファイルという間接的な実行方法は，スクリプトのファイルを作成してすぐに実行可能で便利である．

ex: 実行 sample1.sh

```
$ .  sample1.sh
Wed Jun 1 10:08:36 2006
I am TOMOHARU
Hello!
```

また，今後コマンドのように使用したいスクリプトの場合は次のような手続きを行う．chmod コマンドにより，ユーザに対してファイルの実行権限を与える．実行権限を与えた後は，コマンドと同様，ファイル名を入力するだけで実行される．以下に示すのは，sample1.sh ファイルに実行権限を与え，書かれたコマンドが実行された結果である．

```
$ chmod u+x sample1.sh
$ sample1.sh
Wed Jun 10:08:36 2006
I am TOMOHARU
Hello!
```

5.1.3 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 特別なシェル変数

変数	説明
$n	スクリプトに渡された n 番目の引数 （$0 はスクリプト名 $1，$2…は第 1 引数，第 2 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例を以下に示す．

ex: sample2.sh

```
#!/bin/bash
echo "シェルスクリプト名：$0"
echo "第 1 引数：$1，第 2 引数：$2，第 3 引数：
$3"
echo "引数の数：$#"
echo "全引数：$@"
```

実行結果は次のとおりである．

ex: 実行 sample2.sh

```
$ sample2.sh senda murakami kida
シェルスクリプト名：sample2.sh
第 1 引数：senda，第 2 引数：murakami，第 3 引数：kida
引数の数：3
全引数：senda murakami kida
```

5.2 シェルスクリプトの制御構文

シェルスクリプトにも，他のプログラミング言語のように制御構文がある．まず制御構文について述べる前に，制御構文で用いる条件判定の記述方法について説明する．

5.2.1 条件式の記述方法

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．ここでは test コマンドを用いて，条件判定を行う．条件が真の時は 0 を返し，偽の時は 1 を返す．

```
test 条件式        もしくは
[  条件式  ]
```

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が必要である．文字列比較演算子を Table 4，ファイル属性演算子を Table 5，論理演算演算子を Table 6，数値評価演算子を Table 7 に示す．

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	2 つの数値が等しい
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	2 つの数値が等しくない

test コマンドを用いて/home/TOMOHARU がディレクトリであるかをチェックする．

```
$ test -d /home/TOMOHARU
$ echo $?
0
```

“$?”は直前のコマンドの終了ステータスを意味する．/home/TOMOHARU はディレクトリ（真）であるため，「0」を出力する．

5.2.2 for 文

与えられた引数の数だけ処理の繰り返しを行いたいときに for 文を用いる．for 文の記述方法を以下に示す．for 文は「in」の後に続く引数を変数に代入し，「do」と「done」の間に記述されたコマンドを繰り返し実行する．そして「in」の後の引数が全てなくなれば，処理を終了する．

```
for 変数 in リスト
do
  コマンド
done
```

5.2.3 while 文

while 文は，ある条件の真偽によって処理の繰り返しを行う際に用いる．条件が真である間は，「do」と「done」の間に記述されたコマンドを繰り返し実行する．条件が偽になれば，処理を終了する．

```
while [ 条件文 ]
do
  コマンド
done
```

5.2.4 if 文

特定の条件によって処理を分岐する際に if 文を用いる．if 文の記述方法を以下に示す．複数の条件を用いる “elif”, や条件が不成立の場合に実行する “else”は省略することもできる．

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

5.2.5 case 文

変数の値によって処理が異なるときに case 文を用いる．case 文の記述方法を以下に示す．

```
case 変数 in
  パターン 1) コマンド 1;;
  パターン 2) コマンド 2;;
  パターン 3) コマンド 3;;
          *) コマンド n;; esac
```

パターンには正規表現を用いることができる．上記の例では，どのパターンとも一致しなかった場合はコマンド n を実行する (コマンド n の行は省略可である)．また, パターンを“ |"で区切って論理和（or）をとることもできる．

5.3 シェル関数

シェルスクリプトでは，独自に関数を定義して使用し，関数を引数で渡すことができる．よく利用する複数の処理を関数にまとめることで，プログラム全体が分かりやすくなる．関数は以下のように定義する．

```
関数名 () {
  コマンド
}
```

以下に例を示す．

ex: file_func ファイル

```
#!/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: func の実行

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

上記の例では，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 つの引数をつけている．