ПРОГРАМИРАНЕ НА КОМАНДЕН ЕЗИК В UNIX И LINUX

1. ГЕНЕРИРАНЕ НА СПИСЪК ОТ ИМЕНА НА ФАЙЛОВЕ

Ако в дума, която трябва да е име на файл, се срещат неекранирани символите: “*”, “?” “[]“, то те имат специално значение. Думата се интерпретира от shell като шаблон за име на файл. Тя се разширява до азбучно подреден списък от имената на файловете, които съответстват на шаблона и се намират в текущия или явно указания каталог. Това разширение се нарича Pathname expansion. Ако не се открие нито едно съответствие, то думата остава непроменена. Специалното значението на символите е следното:

`*` Разширява се до произволен брой произволни символи (включително и 0 на брой).

`?` Разширява се до един произволен символ.

`[abc]` Разширява се до един символ измежду изброените в скобите, или в интервала.
`[a-d]`

`[^ab]` Разширява се до един символ, различен от изброените в скобите.

Файл, чието име започва със символа “.” се третира от shell като скрит при някои операции, включително и при интерпретиране на горните метасимволи, т.е. те не се разширяват до “.” в началото на името на файла.

Пример. Нека всички файлове, намиращи се в текущия каталог са:

```
.bashrc  a.o  b.1.2  c  f  f1  fb  file.c
```

тогава в зависимост от шаблона ще бъдат генерирани следните списъци от имена на файлове:

Шаблон	Списък от имена на файлове
`f*`	`f f1 fb file.c`
`*c`	`c file.c`
`f?`	`f1 fb`
`?b*`	`fb`
`[ab]*`	`a.o b.1.2`
`[a-c]*`	`a.o b.1.2 c`
`[^ab]*`	`c f f1 fb file.c`

2. ПРЕНАСОЧВАНЕ НА ВХОД И ИЗХОД

Всяка команда (всъщност всеки процес) има на разположение три стандартно отворени файла, които може да използва: стандартен вход, стандартен изход и стандартен изход за грешки. И трите файла стандартно са свързани с терминала на потребителя. С всеки отворен файл се свързва файлов дескриптор, който е цяло неотрицателно число и посредством него се идентифицира отворения файл в системните примитиви, изпълнявани от процеса. Файловите дескриптори на стандартните файлове са съответно 0, 1 и 2. Преди да изпълни определена команда shell може да свърже стандартните файлове с нещо различно от терминала и това се нарича пренасочване на вход/изход (Redirection). Синтаксисът за пренасочването е следния:

команда < *файл*

Пренасочва стандартния вход на командата от файла.

команда > *файл*

Пренасочва стандартния изход на командата към файла. Ако файлът не съществува, първо се създава, а ако съществува, то старото му съдържание се изтрива.

команда >> *файл*

Пренасочва стандартния изход на командата към файла. Ако файлът не съществува, първо се създава, а ако съществува се отваря за добавяне.

команда 2> файл
команда 2>> файл

Пренасочва стандартния изход за грешки на командата към файла.

Пренасочване може да се прави за всеки един файлов дескриптор. Общият вид на конструкциите за пренасочване на вход/изход е:

[*n*]< *файл*	По премълчаване *n* е 0 и е файлов дескриптор отворен за вход.
[*n*]> *файл,* [*n*]>>*файл*	По премълчаване *n* е 1 и е файлов дескриптор отворен за изход.
[*n*]> *&m*	По премълчаване *n* е 1. Тук *n* и *m* са файлови дескриптори за изход и изходите по двата дескриптора се сливат.

За една команда могат да се пренасочват няколко файла, като shell интерпретира пренасочванията отляво надясно.

Преди пренасочването, в думата *файл* се интерпретират метасимволите: „$”, „~” и тези за имена на файлове (прави се заместване на променливи, аритметично разширение, заместване на „~” и разширение до имена на файлове). Резултатът от разширението става име на файла, но трябва да е една дума, иначе е грешка.

Пример. Пренасочване на стандартен вход и изход на команди.

```
ls d1 > d1list
F=d1list
ls d1 > $F
date > logfile
who >> logfile
write student1 < letter
cc prog.c 2> errorlist
find /home -name core –print > corelist 2> /dev/null
find /home -name core –print > corelist 2>&1
```

Последният пример показва как могат да се пренасочат стандартния изход и изход за грешки към един и същи файл. Има и по-кратка конструкция за това:

команда &>файл.

Съществува друга форма за пренасочване на входа, която позволява командата и входните данни да се намират на едно място. Нарича се Here documents и има следния вид.

команда << низ
входни данни
. . .
низ

Тук *низ* е символен низ, който ограничава входните данни и не трябва да се среща в тях. Ако е указано << *низ*, то нито един метасимвол в данните не се интерпретира, в противен случай се интерпретират: „$”, „`” и символите за имена на файлове.

Пример. Пренасочване на стандартен вход - Here documents.

```
write student <<EOI
Hello from $USER
Happy New Year!
EOI
```

```
write student <<\EOI
Hello from $USER      # No expansion
EOI
```

3. КОНВЕЙЕР

Конвейерът (Pipeline) е конструкция, която включва две или повече команди, разделени чрез символа “|”, а именно:

команда1 | *команда2* [| *команда3*] ...

При изпълнението на конвейер shell пренасочва стандартния изход на *команда1* към стандартния вход на *команда2*, стандартния изход на *команда2* към стандартния вход на *команда3* и т.н. Командите работят като конкурентни процеси, които комуникират чрез един от механизмите за комуникация между процеси, наречен програмен канал (pipe). Команда, която чете данни от стандартния вход, извършва някаква обработка на данните и пише резултата на стандартния изход, се нарича филтър. Командният език включва голям брой команди-филтри,

напр., `cat, cut, grep, head, od, pr, sort, tail, tee, tr, wc` и др., което дава много възможности за конструиране на конвейри.

Пример. Конвейер.

```
ls | wc –l
ls –l | grep "^-" | wc –l
ls –l | grep "^-" | tee filelist | wc –l
who | wc –l
who | grep "student"| wc –l
who | grep "student"| tee userlist | wc –l
```

4. ФОНОВ РЕЖИМ НА ИЗПЪЛНЕНИЕ

Изпълнение на команда във фонов или асинхронен режим (background) означава, че след като shell стартира изпълнението на командата не чака нейното завършване, а продължава работата си, т.е. извежда покана, ако е интерактивен, и чете следваща команда. Това означава, че в такъв режим могат да се изпълняват само външни команди. По премълчаване режимът на изпълнение е привилегирован или синхронен (foreground), т.е. процесът shell се синхронизира със завършването на командата. За да бъде изпълнена една команда във фонов режим се използва друг специален символ “&”:

команда &

Ако командата използва стандартен вход, то трябва той да бъде пренасочен. Същото се препоръчва и за стандартния изход и изход за грешки, тъй като в противен случай резултатът е непредсказуем. Например, ако командата се опита да чете от терминала, то ще й бъде изпратен сигнал, който ще предизвика спирането й (състояние stopped). Изпълнението на спряна командата може да се възстанови само в привилегирован режим (чрез командата `fg`).

Пример. Командата `find`, изпълнена във фонов режим с пренасочване на изходите.

```
find /home –name core –print > corelist 2> /dev/null &
```

5. КОД НА ЗАВЪРШВАНЕ

Всяка команда, изпълнена в привилегирован режим (всъщност всеки процес), при завършването си изработва код на завършване (exit status) и го връща към shell (родителския процес). Прието е код 0 да означава нормално (успешно) завършване на командата, а код различен от 0 (число от 1 до 255) да означава грешка или изключителна ситуация (неуспех). Код на завършване на конвейер е кода на последната команда в него.

6. СПИСЪК ОТ КОМАНДИ

Последователност от команди и конвейри, които са разделени с някой от символите: “;”, “&”, “&&”, “||” и завършва с един от символите “;”, “&”или нов ред ще наричаме списък от команди. В зависимост от символа, разделящ командите (символ за групиране), има следните видове списъци:

- Списък за последователно изпълнение

  команда1 ; *команда2* [; *команда3*] ...

  Символът “;” е еквивалентен на символа за нов ред. Код на завършване на списъка е кода на последната изпълнена команда.

- Списък за асинхронно изпълнение

  команда1 & *команда2* [& *команда3*] ...[&]

  Командите в списъка се изпълняват асинхронно една спрямо друга. Shell стартира изпълнението на *команда1, команда2* и т.н. без да изчаква завършването им, след което ако накрая няма “&” той се синхронизира със завършването на последната команда, в противен случай не чака и нейното завършване, а продължава работата си.

- Списъци за условно изпълнение

команда1 && *команда2*

Изпълнява се *команда1* и ако тя върне код 0 се изпълняза *команда2* (AND list).

команда1 || *команда2*

Изпълнява се *команда1* и ако тя върне код различен от 0 се изпълняза *команда2* (OR list). Код на завършване на списък за условно изпълнение е кода на последната изпълнена команда.

В един списък може да се срещне комбинация от различни символи за групиране. Символите "&&" и "||" са с еднакъв приоритет, който е по-висок приоритета на ";" и "&".

Пример. Списъци от команди.

```
cmp -s f1 f2 && rm f2
grep "^student: " /etc/passwd > /dev/null 2>&1||echo "No user"
```

В списък команди могат да се използват два вида скоби, които се интерпретират от shell по различен начин.

- Скоби

{ *списък_команди*; }

Списък от команди заграден във фигурни скоби се изпълнява от текущия процес shell. Конструкцията се използва за изменение на реда на интерпретиране на символите за групиране или за формиране на обединен изход или вход на няколко команди.

(*списък_команди*)

Списък от команди заграден в кръгли скоби се изпълнява от нов процес shell, който е породен от текущия процес shell (т.е. от subshell). Това означава, че всякакви изменения в обкръжението на shell при изпълнението на *списък_команди* нямат ефект след завършването му (например, промяна на текущия каталог или присвояване на значения на променливи). Друга възможност, която дава тази конструкция, е изпълнение на списък от команди във фонов режим.

Пример. Списъци от команди и скоби.

```
cmp -s f1 f2 && { rm f2; mv f1 file1; }
{ date; who; } > logfile
{ cat d1list; ls d2; } | wc -l
(cd dir; . . . )
(sleep 100; { date; who; } > logfile) &
```

7. ПРОМЕНЛИВИ

Променлива в shell има име, значение и евентуално атрибути. Името е символен низ от букви, цифри и символа за подчертаване "_", като започва с буква или символа за подчертаване. Значението е символен низ, включително и празен низ. Декларирането на променливи не е задължително. Променливата се обявява най-често с оператора за присвояване. С този оператор променливата се включва в обкръжението на текущия процес shell и е там докато не бъде изключена с командата `unset`. Операторът за присвояване има вида:

име_променлива=[*значение*]

Ако не е зададено *значение*, то значението е празен низ. Значението е низ, който може да е ограничен с единични или двойни кавички. Ако в низа няма интервали и метасимволи (като < > | & $ ` и други), то не е необходимо ограничаването му с кавички. Ако в низа има интервали, но няма метасимволи, то е задължително ограничаването му с кавички, независимо кои. Ако в низа има метасимволи, то двата вида кавички имат различно действие. Единичните кавички отменят специалното значение (екранират) на всички метасимволи в низа без своето собствено. Двойните кавички екранират всички метасимволи в низа с изключение на $, ` и \. В *значение* се прави заместване на променлива, заместване на изхода, и аритметични разширения, ако съответните метасимволи не са екранирани.

В по-новите версии на shell (Bash 2.xx) значението може да има вида: *$'низ*'. Тогава в *низ* се заместват всички последователности по ANSI C стандарта: `\n, \t, \v, \nnn`, където `nnn` е осмичен код на символ и др. Цялата конструкция се замества с получения низ, заграден в единични кавички.

Пример. Оператор за присвояване.

```
L=LINUX или L="LINUX" или L='LINUX'
D=/home/student1/proc/d1
OS="Red Hat LINUX" или OS='Red Hat LINUX'
A="a<b" или A='a<b'
X=$'aa\043\t\044' или X=$'aa#\t$'
```

Значението на променлива се използва, когато се срещне следната конструкция:

$*име_променлива* или ${*име_променлива*}

Тогава цялата конструкция, се замества със значението на променливата. Това се нарича заместване на променлива (Variable expansion). Ако променливата не е определена, то се замества с празен низ. Името на променливата се използва без символа „$" пред него само когато се присвоява значение, променят се атрибути или се иска информация за променливата.

Пример. Заместване на променлива.

```
OS="Red Hat $L"
cd $D
ls -l $D/a*
```

Освен тази основна и най-често използвана конструкция за заместване значението на променлива в по-новите версии на shell има по-сложни конструкции за заместване.

${*име_променлива*:=*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то й се присвоява значение *дума* и се замества това значение, иначе се замества значението на *име_променлива*.

${*име_променлива*:-*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то се замества *дума*, иначе се замества значението на *име_променлива*.

${*име_променлива*:+*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то нищо не се замества иначе се замества *дума*.

${*име_променлива*:?*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то се извежда съобщение *дума* на стандартния изход за грешки, иначе се замества значението на *име_променлива*.

Ако в горните конструкции липсва символа „:”, се проверява само дали *име_променлива* е неопределена. Преди да се използва *дума*, в нея се извършва заместване на променлива, заместване на изхода, аритметично разширение и заместване на "~".

${*име_променлива*:*отместване*}

${*име_променлива*:*отместване*:*брой*}

Това са конструкции за заместване на подниз от значението на променлива. Замества се значението на *име_променлива*, започвайки от *отместване* до края или максимално *брой* символи.

${#*име_променлива*}

Замества се с броя символи в значението на *име_променлива*.

С променливите са свързани няколко вътрешни команди: `echo`, `read`, `set`, `unset`, `export`, `readonly`, `declare`.

`echo [`*опции*`] [`*низ* …`]`

Извежда на стандартния изход аргументите си.

`read` *име_променлива* …

Чете ред от стандартния вход и го разделя на думи. Всяка дума присвоява на поредната променлива, указана като аргумент. Ако броят на прочетените думи не съответства на броя на променливите, то или последните променливи имат празно значение или на последната променлива се присвоява останалата част от реда. Тук името на променливата е без символа „$”, защото с `read` се присвоява значение на променлива.

`set`

Тази команда има много различни действия, но в най-простия си вид без аргументи извежда имената и значенията на всички променливи, които са определени в текущия shell.

`unset` *име_променлива* …

Указаните променливи се изключват от обкръжението на текущия shell, т.е. ще се считат за неопределени.

`export [-n][`*име_променлива* …`]`

Указаните променливи се отбелязват за предаване към обкръжението на процеси, които ще бъдат породени от текущия shell (това може да е и subshell). Такива променливи се наричат “експортни” или променливи с атрибут export. Те са подобни на глобалните променливи в езиците за програмиране, само че тук глобалността е еднопосочна. Ако има опция `-n`, то се отменя атрибута export за указаните променливи. Ако няма аргументи извежда имената и значенията на всички експортни променливи.

`readonly [`*име_променлива* …`]`

За указаните променливи се определя атрибут readonly т.е. след това значението им не може да бъде променяно. Ако няма аргументи извежда имената и значенията на всички променливи с атрибут readonly.

`declare [опции] [`*име_променлива* …`]`

Тази команда може да се използва вместо `export` и `readonly`. Чрез нея може да се обяви променлива с определени атрибути и дори значение, или да се измени атрибут на променлива или да се получи информация за променливите от обкръжението на shell. В същност в командите `declare`, `export` и `readonly` вместо *име_променлива* може да е оператор за присвояване. Опциите определят действието на командата.

- `-x` Променливите се обявяват за експортни (атрибут export).
- `-r` Променливите стават достъпни само за четене (атрибут readonly).
- `-i` Променливите се обявяват за целочислени (атрибут integer).

Ако опцията се предшества от знак “+”, то съответния атрибут се отменя. Ако няма аргументи променливи, командата извежда информация за променливите с указаните атрибути.

Пример. Команди `declare, export, readonly, set.`

```
declare                 # set или declare -p
declare -x              # export или export -p
declare -r              # readonly или readonly -p
declare -xr             # all export or readonly variables
A=123; export A         # A=123; declare -x A
declare -x A=123
declare +x A
```

Нормалната работа на shell, а и на цялата операционна система, зависи от ред променливи, които всеки потребител получава вече определени в началото на сесията, т.е. те са включени в обкръжението на процеса login shell. Ще ги наричаме **системни променливи.** Те могат да се използват, а някои да бъдат променяни от потребителя. Пълният списък от системни променливи варира в различните версии на shell, но има някои, които са общи и важни за функционирането му.

Име Значение

`HOME` Пълното име на началния каталог на потребителя.

`PATH` Списък от каталози, в които shell търси външни команди.

`PS1` Първичната покана (prompt) на shell. Възможни значения: $, # , bash$

`PS2` Вторичната покана на shell. Стандартното значение е >.

`IFS` (Internal Field Separator) Символите, които се разглеждат като разделители на думи при word splitting и командата `read`. Стандартно са “<space><tab><newline>”.

`MAIL` Името на файла, който се използва като пощенска кутия на потребителя от командата `mail` и се проверява от shell за новопристигнали съобщения.

`UID` (User id) Вътрешен идентификатор на потребителя. (с атрибут readonly).

`USER` Име на потребителя. (Използват се също `USERNAME`, `LOGNAME`.)

`SHELL` Име на потребителския login shell.

`PWD` Името на текущия в момента каталог.

`OLDPWD` Името на предишния текущ каталог.

Следващите променливи са типични най-вече за bash, но е възможно и поддържането им в други версии на shell.

`OSTYPE` Наименование на ОС.

`HOSTTYPE` Тип на машината.

`HOSTNAME` Наименование на машината.

`BASH` Името на файла с изпълнимия код на bash, стандартно `/bin/bash`.

`BASH_VERSION` Версия на bash.

`BASH_ENV` Име на файл, който се изпълнява при извикване на командна процедура (аналог на инициализиращата процедура за интерактивен не-login shell - `.bashrc`).

При стартиране на процес shell могат автоматично да се изпълнят **инициализиращи командни процедури** (наричат ги profiles).Чрез тях обикновено се настройва обкръжението на процеса shell, например системните променливи. Изпълняваните командни процедури зависят от вида на процеса shell:

- login shell процес
  1. `/etc/profile`
  2. `~/.bash_profile, ~/.bash_login, ~/.profile` (първия намерен)
- интерактивен не-login shell процес
  `~/.bashrc`
- неинтерактивен shell процес (изпълнение на командна процедура)
  `$BASH_ENV`

Ако дума започва с неекраниран символ `"~"`, той е метасимвол и въвежда така нареченото **разширение на символа** `"~"` (Tilde expansion). Вариантите на това разширение са:

"~/" и "~ "

Замества се със значението на променливата HOME или ако HOME не е определена с началния каталог на текущия потребител.

~име_потребител

Замества се с началния каталог на указания потребител.

"~+/" и "~+ "

Замества се със значението на променливата PWD (при "~-/" и "~- " с OLDPWD).

Във всички други случаи не се прави разширение на символа "~", т.е. думата остава непроменена.

Пример. Илюстрират се по-сложни конструкции за заместване на променливи в комбинация с други типове замествания. В коментар е даден изхода от командата echo.

```
# paramsub
# login as moni, current directory is /home/moni/proc
#-------------------------------------------------
unset a  # or a=
i=1
echo "${a:-`whoami`}, a=$a"           # moni, a=
echo "${a:-$USER}, a=$a"              # moni, a=
echo "${a:-~+}, a=$a"                 # /home/moni/proc, a=
echo "${a:-$((i+1))}, a=$a, i=$i"     # 2, a=, i=1
a=
i=1
echo "${a:=`whoami`} a=$a"             # moni, a=moni
echo "${a:=$((i+=1))}, a=$a, i=$i"     # moni, a=moni, i=1
a=
echo "${a:=$((i+=1))}, a=$a, i=$i"     # 2, a=2, i=2
echo  ${a:+"do you want to redefine a?"} # do you want to redefine a?
echo ${a:?"a is unset or null"}        # 2
a=
echo ${a:?"unset or null"}  # paramsub: a : unset or null
                            # output to stderr and exit shell
#-------------------------------------------------
```

8. ЗАМЕСТВАНЕ НА ИЗХОДА

Конструкцията заместване на изхода (Command substitution) означава заместване на команда с това, което тя извежда на стандартния изход. По-старият и новият синтаксис на конструкцията са съответно:

`*команда*` и $(*команда*)

Вместо команда може да е конвейер и дори списък от команди. Възможно е влагане на конструкции заместване на изхода, но използването на стария синтаксис е доста сложно и неудобно. Много често заместването на изхода се използва в оператор за присвояване.

Пример. Заместване на изхода на команда.

```
D1=`pwd` или D1=$(pwd)
now=`date` или now=$(date)
day=`date|cut -f1 -d' '``
num=`who|wc -l` или num=$(who|wc -l)
echo "Number of sessions:`who|wc -l`"
echo `ls \`pwd\`/a*`
echo $(ls $(pwd)/a*)
echo `ls $(pwd)/a*`
echo $(ls `pwd`/a*)
```

9. АРИТМЕТИЧНИ ИЗЧИСЛЕНИЯ

Аритметични изчисления в командния език могат да се извършат по няколко начина. Една възможност е да се използва командата `expr`, която се разпространява с всички UNIX и LINUX системи.

`expr` *`израз`*

Командата изчислява значението на целочисления *`израз`* и го извежда на стандартния изход. *`Израз`* се конструира от константи (цели числа), променливи и аритметичните операции: +, -, *, / (цялата част при деление), % (остатъка при деление). Операциите + и – имат еднакъв приоритет, който е по-нисък от този на *, / и %. Изчислението се извършва от ляво на дясно, като скоби не могат да се използват. В израза се прави заместване на променливи и заместване на изхода преди изчислението.

Пример. Команда `expr`.

```
x=12
expr $x + 1
expr $x "*" 2
x=`expr $x + 1`
expr `who | wc -l` - 1
```

Друг начин за извършване на аритметични изчисления, който се реализира от по-новите версии на shell е така нареченото аритметично разширение (Arithmetic Expansion). Тази конструкция има стар и нов синтаксис, които са съответно:

`$[`*`израз`*`]` и `$((`*`израз`*`))`

Изчислява се целочисления *`израз`* и резултата замества конструкцията. Преди да се изчисли израза се извършва заместване на променливи и заместване на изхода. Конструкциите аритметичното разширение могат да се влагат. Изчисленията се извършват в `long int`. Изразът се конструира от константи, променливи, аритметичните операции и скоби. Операциите и приоритетът им е както в езика C. Значенията на променливите се заместват в израза независимо, дали пред името има или не символ „$”. Константите могат да се записват като десетични, осмични или шестнадесетични цели числа в C стил.

Пример. Аритметично разширение.

```
x=12
echo $(($x+1))   # echo $[$x+1]
echo $(($x*2))
x=$[$x+1]
x=$(($x+1))      # x=$((x+1))
: $((x=x+1))     # : $((x+=1))
echo $((`who | wc -l` - 1))
```

10. КОМАНДНИ ПРОЦЕДУРИ И АРГУМЕНТИ

Командната процедура (команден файл, shell файл, shell script) е текстов файл, съдържащ конструкции на командния език на shell, следователно това е програма на командния език, която се интерпретира от shell. Командата, която завършва изпълнението на командна процедурата е:

`exit [`*`n`*`]`

където *`n`* е кода на завършване, който процедура аналогично на командите, връща. Когато процедурата завърши без команда `exit` (или с `exit,` но без аргумент), то тя връща кода на последната изпълнена команда.

Коментар в командна процедура е всичко от символа “#” до края на реда (освен ако символът е екраниран или ако е в комбинацията “#!”, която е в началото на първия ред на файла). Символът за начало на коментар се екранира, както и другите метасимволи (чрез “, ‘, \)

Пример. Командната процедура изчислява и извежда броя на сесиите в системата. Записана е като файл в текущия каталог с име `nsession`.

```
# nsession
#-------------------------------------------------
date
n=`who | wc -l`
echo "Number of sessions: $n"
#-------------------------------------------------
```

За да бъде извикана за изпълнение една командна процедура така както се вика външна команда е необходимо:

1. Да се дадат права на потребителите да изпълняват файла, например с командата:

```
$ chmod a+x nsession
```

2. Файлът да е в каталог от списъка в променливата `PATH`.

Ако са изпълнени и двете условия, командната процедура се извиква:

```
$ nsession
```

Ако файлът не е в каталог от `PATH`, то при извикване трябва да се задава пълно име на файла – абсолютно или относително:

`$ /home/student/nsession` или ако е в текущия каталог: `./nsession`

Друг начин за изпълнение на командна процедура е да се извика shell и да му се предаде като първи аргумент името на файла с командната процедура:

```
$ bash nsession
```

При този начин файлът се търси в каталозите от `PATH`, ако не е зададен с пълно име, но не е необходимо потребителят да има право за изпълнение. Този метод е удобен при тестване на командни процедури, тогава shell може да се извика с опция `-x` или `-v`. И още един начин за извикване е:

```
$ . nsession
```

Командната процедура се изпълнява от текущия процес shell. Файлът се търси в каталозите от `PATH`, ако не е зададен с пълно име.

В командна процедура могат да се използват **позиционни параметри**. Значения им се присвояват при извикване на процедурата чрез аргументите в командния ред или с команда `set`, но не и с оператор за присвояване. Имената на позиционните параметри са:

```
$0, $1, $2, $3, $4, … ${10}, . . .
```

Нулевият параметър `$0` съдържа името на файла с командната процедура, а останалите съответстват позиционно на действителните аргументи, зададени при извикване на процедурата след името на файла. Ако при извикване не са зададени необходимия брой действителни аргументи, то съответните променливи `${i}` ще имат значение празен низ.

Пример. Командната процедура е с един аргумент, който трябва да е име на потребител, и изчислява и извежда броя на сесиите на потребителя.

```
# userlog user_name
#-------------------------------------------------
n=`who | grep "^$1 " | wc -l`
echo "User $1: $n sessions"
#-------------------------------------------------
```

Съществува още една група от променливи, които се поддържат от shell, могат да се използват от потребителя, но само shell променя значението им. Ще ги наричаме **вътрешни променливи**:

Име	Значение
`$#`	Брой аргументи, предадени на shell без нулевия.
`$*`, `$@`	Всички аргументи, предадени на shell без нулевия, разделени с интервал.
`$?`	Код на завършване на последната изпълнена команда в привилегирован режим.
`$$`	Идентификатор (pid) на процеса shell.
`$!`	Идентификатор (pid) на последния изпълнен фонов процес.

Когато се използват в командна процедура, променливите `$#, $*` и `$@` се отнасят до аргументите, предадени при извикването й. Има разлика между `$*` и `$@` :
`"$*"` е еквивалентно на `"$1`*c*`$2`*c*`$3`*c*`…", `където *c* е първият символ от значението на променливата `IFS` или интервал
`"$@"` е еквивалентно на `"$1" "$2" "$3" …`
Тази разлика се проявява при наличие на аргументи, съдържащи интервали в себе си, и когато върху променливите не се извършва така нареченото действие word splitting (word splitting не се прави при заграждането им в двойни кавички). Следващият пример илюстрира тези различия.

Пример. Командната процедура илюстрира разликата между променливите `$*` и `$@`. В коментар е показан един резултат от изпълнението й.

```
# arglist arg1 arg2 ...
#-------------------------------------------------
index=1
echo "Listing of arguments with \"\$*\":"
for a in "$*"
do
     echo "Arg $index = $a"
      index=$(( index + 1 ))
done
echo
index=1
echo "Listing of arguments with \"\$@\":"
for a in "$@"
do
     echo "Arg $index = $a"
      index=$(( index + 1 ))
done
echo
index=1
echo "Listing of arguments with \$*:"
for a in $*
do
     echo "Arg $index = $a"
     index=$(( index + 1 ))
done
echo
index=0
echo "Listing of arguments with \$@:"
for a in $@
do
     echo "Arg $((index+=1)) = $a"
done
exit 0
#-------------------------------------------------
#  arglist "aa nn" bb cc
#-------------------------------------------------
# Listing of arguments with "$*":
# Arg 1 = aa nn bb cc
#
# Listing of arguments with "$@":
# Arg 1 = aa nn
# Arg 2 = bb
# Arg 3 = cc
#
# Listing of arguments with $*:
# Arg 1 = aa
```

```
# Arg 2 = nn
# Arg 3 = bb
# Arg 4 = cc
#
# Listing of arguments with $@:
# Arg 1 = aa
# Arg 2 = nn
# Arg 3 = bb
# Arg 4 = cc
#--------------------------------------------------
```

Върху значенията на позиционните параметри влияе командата `shift`.

```
shift [n]
```

Измества значенията на аргументите с *n* позиции наляво, т.е. към малките номера, а значенията на първите *n* аргумента се губят. По премълчаване *n* е 1. Променят се съответно и значенията на променливите `$*`,`$@` и `$#`. Например, при изпълнение на:

```
shift
```

се извършва следното изместване `$2->$1, $3->$2,` и т.н. Старото значение в `$1` се губи, а `$0` не участва в изместването.

11. УСЛОВЕН ОПЕРАТОР И КОМАНДИ ЗА ПРОВЕРКА НА УСЛОВИЯ

```
if списък_команди1
then списък_команди2
[elif списък_команди3
then списък_команди4] …
[else списък_команди5]
fi
```

В условните конструкции на командния език *списък_команди1* и *списък_команди3* са произволни списъци от команди, които тук изпълняват и ролята на условие. Когато списъкът от команди, след като се изпълни върне код на завършване 0, условието е истина, а ако върне код различен от 0 е лъжа. Изпълнява се *списък_команди1* и ако върне код 0 се изпълнява *списък_команди2*, иначе се продължава с клона `elif` ако го има или `else` ако го има.

Пример. Условен оператор.

```
if cmp –s f1 f2
then echo "Files f1 and f2 are identical"
else echo "Files f1 and f2 differ"
fi
```

```
if cd dir 2>/dev/null
then echo "Now in dir"
else echo "Can’t change to dir"
fi
```

```
if grep "^student: " /etc/passwd >/dev/null 2>&1; then :
else echo "No user student"
fi
```

```
if who | grep "^student "
then :
else echo "User student is not working"
fi
```

Код на завършване на оператор `if` е кода на последната изпълнена команда в `then` или `else` списъка, или е 0 ако нито едно условие не е истина и няма `else`. Операторите `if` могат да се влагат.

Много често в `if` се използва командата `test` и синонима й [, която проверява различни условия и връща код 0 при истина, и 1 при лъжа.

`test` *условие* или `[` *условие* `]`

Условието, което се проверява включва проверки за файлове, сравнения на низове и сравнения на числови низове, които могат да се комбинират с логическите операции, които в ред на намаляване на приоритета са:

! (логическо NOT), `-a` (логическо AND), `–o` (логическо OR)

- Условия за файлове:

-e *файл*	истина, ако *файл* съществува
-f файл	истина, ако *файл* съществува и е обикновен файл
-d *файл*	истина, ако *файл* съществува и е каталог
-c *файл*	истина, ако *файл* съществува и е символен специален файл
-b *файл*	истина, ако *файл* съществува и е блоков специален файл
-L *файл*	истина, ако *файл* съществува и е символна връзка
-s *файл*	истина, ако *файл* не е празен, размерът му е по-голям 0
-r *файл*	истина, ако *файл* е достъпен за четене за потребителя, изпълняващ test
-w *файл*	истина, ако *файл* е достъпен за писане за потребителя, изпълняващ test
-x *файл*	истина, ако *файл* е достъпен за изпълнение за потребителя, изпълняващ test

файл1 -nt *файл2* истина, ако *файл1* е по-нов от *файл2* според датата на последно изменение
файл1 -ot *файл2* истина, ако *файл1* е по-стар от *файл2* според датата на последно изменение

- Условия за низове:

низ1 = *низ2* истина, ако *низ1* и *низ2* са еднакви
низ1 != *низ2* истина, ако *низ1* и *низ2* са различни
[-n] *низ1* истина, ако *низ1* не е празен
-z *низ1* истина, ако *низ1* е празен

- Условия за сравнение на числа:

n1 –eq *n2* истина, ако *n1* и *n2* са равни

Другите операции за сравнение на числа са : `-ne, -lt , -le, -gt, -ge`.

Преди да се провери условието в него се прави заместване на променливи, заместване на изхода и аритметично разширение.

Пример. Команда `test.`

```
if test –e file; then
     echo "File exists"
fi
```

```
dir=/home/student/proc
if [ -d $dir –a –r $dir ]; then
     echo "You have read permission on directory $dir"
fi
```

В новите версии на bash (от 2.02) е въведена и разширена команда за проверка на условия, която има вида:

```
     [[ условие ]]
```

Разширението се състои в добавянето на скоби, нови логически операции: && (логическо AND) и || (логическо OR) и нови операции за сравнение на низове:
низ1 = = *низ2* еквивалентно на операцията =
низ1 < *низ2* истина, ако *низ1* е преди *низ2* според ASCII наредбата
низ1 > *низ2* истина, ако *низ1* е след *низ2* според ASCII наредбата

Пример. Разширена команда `test.`

```
if [[ -d $dir && –r $dir ]]; then
     echo "You have read permission on directory $dir"
fi
```

Пример. Нов вариант на командната процедура, която изчислява и извежда броя на сесиите на потребител, чието име е зададено като аргумент.

```
# userlog2 user_name
#-------------------------------------------------
if [ $# -eq 0 ]
then echo "usage: $0 user_name" >&2; exit 1
fi
if grep "^$1:" /etc/passwd > /dev/null 2>&1
then
     n=`who | grep "^$1 " | wc -l`
     echo "User $1: $n sessions"
else echo "No user $1" >&2; exit 2
fi
exit 0
#-------------------------------------------------
```

12. ОПЕРАТОРИ ЗА ЦИКЪЛ

В операторите `while` и `until` итерациите на цикъла се управляват от условие. Условието, както и при оператора `if`, е списък от команди. Ако списъкът върне код 0 условието се счита за истина, иначе е лъжа. Тялото на цикъла също е списък от команди между ключовите думи `do` и `done`.

```
while списък_команди1
do списък_команди2
done
```

Цикълът `while` се изпълнява дотогава докато условието е истина. Условието се проверява в началото на итерацията, което означава, че *списък_команди1* се изпълнява поне веднаж и когато върне код различен от 0, управлението се предава на командата след `done`.

```
until списък_команди1
do списък_команди2
done
```

Цикълът `until` се изпълнява дотогава докато условието е лъжа. Условието също се проверява в началото на цикъла, т.е. *списък_команди1* се изпълнява поне веднаж и когато върне код 0, управлението се предава на командата след `done`. Код на завършване на оператор `while` и `until` е кода на последната изпълнена команда в тялото на цикъла *списък_команди2*, или е 0 ако тялото на цикъла не е изпълнено нито веднаж.

Пример. Командната процедура е с един аргумент, който е име на потребител, и чака включването на потребителя в системата и му изпраща директно съобщение.

```
# waitu user_name
#-------------------------------------------------
if [ $# -eq 0 ]
   then echo "usage: waitu user_name" >&2; exit 1
fi
until who | grep "^$1 " >/dev/null
do
   echo "waiting for $1"; sleep 30
done
echo "User $1 is working"
write $1 << !
Hello
!
exit 0
#-------------------------------------------------
```

В оператор `for` итерациите на цикъла се управляват от променлива. Фразата `in` определя значенията, които се присвояват на променливата. За всяко значение се изпълнява тялото на цикъла, което е *списък_команди1*.

```
for променлива [in дума …]
do списък_команди1
done
```

В *дума* се извършва заместване на променливи, заместване на изхода, аритметично разширение и разширение до имена на файлове (т.е. интерпретират се метасимволите $, `, **,** ?, []), в резултат на което се получава списък от елементи. След това на променливата се присвоява всеки един елемент и се изпълнява *списък_команди1*. Ако след като се извърши разширение на метасимволите във фразата `in` не се генерира нито една дума, то тялото на цикъла не се изпълнява нито веднаж. Ако фразата `in` не е зададена по премълчаване се подразбира `in "$@"`, т.е. цикълът се изпълнява по веднаж за всеки един позиционен аргумент.

Код на завършване на оператор `for` е кода на последната изпълнена команда в тялото на цикъла или е 0 ако тялото не се изпълни нито веднаж.

Операторите за цикъл могат да се влагат. Върху изпълнението на цикъл оказват влияние две вътрешни команди, които могат да се използват само в тялото на цикъл.

`break [`*n*`]`

Прекратява изпълнението на съдържащите командата *n* вложени цикъла. Ако *n* е по-голямо от броя на вложените цикли, съдържащи `break`, то се прекратяват всичките. По премълчаване значението на *n* е 1.

`continue [`*n*`]`

Започва нова итерация на *n*-тия цикъл, броейки отвътре навън вложените цикли, съдържащи командата. По премълчаване *n* е 1. Кодът на завършване на `break` и `continue` е 1, ако не се съдържат в цикъл, иначе е 0.

Пример. Командната процедура изпълнява цикъл с 10 итерации по два начина.

```
# Two ways to count up to 10
#-------------------------------------------------
echo "With for"
for i in 1 2 3 4 5 6 7 8 9 10
do
     echo $i
done
echo "With while"
i=1
while [ "$i" -le 10 ]
do
     echo $i
     i=`expr $i + 1`  # i=$(($i+1)) or i=$((i+1))
done
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по-премълчаване е текущия каталог. Извежда пълните имена на всички подкаталози, съдържащи се в каталога. Следват два варианта - `ldir` и `ldir2`.

```
# ldir [ directory ]
# Lists directories in a directory (one level only)
#-------------------------------------------------
if test $# -ne 0
then if test -d "$1"
     then cd $1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
d=`pwd`
for i in *       # $(ls) or `ls`
do
    if test -d "$i"
    then echo $d/$i
    fi
done
exit 0
#-------------------------------------------------

# ldir2 [directory]
# Lists directories in a directory (one level only)
```

```
#-------------------------------------------------
if [ $# -eq 0 ]
then d=`pwd`
else
     if [ -d "$1"]
     then d=$1
     else echo "$1 is not directory">&2; exit 1
     fi
fi
for i in `ls $d`
do
    if [ -d $d/$i ]
    then echo $d/$i
    fi
done
exit 0
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по-премълчаване е текущия каталог. Извежда имената на всички подкаталози, съдържащи се в поддървото на каталога.

```
# listd [directory]
# Lists directories in the subtree of directory (including directory)
# with absolute or relative path name (depends on parameter)
#-------------------------------------------------------------
if [ $# -eq 0 ]
then d=.
else
     if [ -d "$1" ]
     then d=$1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
for i in `find $d -type d -print`
do
    echo $i
done
exit 0
#-------------------------------------------------------------
```

Пример. Друг вариант на решение на горната задача, в който се използва рекурсивно извикване на командната процедура.

```
# listd2 [directory]
# Lists directories in the subtree of directory (including directory)
# with absolute or relative path names (depends on parameter)
#-------------------------------------------------
if test $# -eq 0
then d=.
else if test -d "$1"
     then d=$1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
echo $d
for i in `ls $d`
do
   if test -d $d/$i
   then listd2 $d/$i  # if listd2 in PATH, if not
   # /home/…/listd2 $d/$i
```

```
   fi
done
exit 0
#-------------------------------------------------
```

Пример. Още един вариант на решение, в който се използва рекурсивно извикване на командната процедура. Извежда винаги пълните имена на каталози, включително и на самия аргумент, независимо как е зададен.

```
# listd3 [directory]
# Lists directories in the subtree of directory (including directory)
# wiht absolute path names
#-------------------------------------------------
if test $# -eq 1
then if test -d "$1"
     then cd $1
     else echo "$1 not directory" >&2; exit 1
     fi
fi
echo `pwd`
for i in `ls`
do
   if test -d $i
   then listd3 $i  # if listd3 in PATH, if not
   # then /home/…/listd3 $i
   fi
done
exit 0
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по премълчаване е текущия каталог. Изчислява и извежда имената на обикновените и достъпни за изпълнение файлове в каталога и накрая броя им.

```
# lfile [directory]
# Lists and counts regular/execute files in a directory (one level only)
#-------------------------------------------------
if [ $# -ne 0 ]
then if [ -d "$1" ]
     then cd $1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
count=0
for i in `ls`
do
    if [ -f $i -a -x $i ]
    then echo $i
       count=$((count + 1))
    fi
done
echo "Regular&execute: $count"
#-------------------------------------------------
```

Пример. Командната процедура може да се извиква с произволен брой аргументи, които са имена на потребители. За всеки аргумент изчислява и извежда броя на сесиите на съответния потребител или съобщение ако не съществува такъв потребител.

```
# userlogs user_name ...
#-------------------------------------------------
if [ $# -eq 0 ]
```

```
then echo "usage: userlogs name_name ..." >&2; exit 1
fi
for u
do
      if grep "^$u:" /etc/passwd >/dev/null 2>&1
      then
            n=`who | grep "^$u " | wc -l`
            echo "User $u: $n sessions"
      else
            echo "No user $u" >&2
      fi
done; exit 0
#-------------------------------------------------
```

Пример. Друг вариант на решение, който използва командата `shift`.

```
# userlogs2 user_name ...
#-------------------------------------------------
if [ $# -eq 0 ]
then echo "usage: $0 name_name ..." >&2; exit 1
fi
while [ -n "$1" ]
do
      if grep "^$1:" /etc/passwd >/dev/null 2>&1
      then
            n=`who | grep "^$1 " | wc -l`
            echo "User $1: $n sessions"
      else
            echo "No user $1" >&2
      fi
      shift
done
exit 0
#-------------------------------------------------
```

13. ОПЕРАТОР `case`

```
      case  дума  in
      шаблон1 [ | шаблон2 ] …) списък_команди ;;
      . . .
      esac
```

В *дума* се извършва заместване на променливи, заместване на изхода и аритметично разширение (т.е. интерпретират се метасимволите: $, `). След това се търси съответствие на получената дума с шаблоните по реда на задаването им. Изпълнява се списъка команди при първото намерено съответствие и изпълнението на оператора завършва. В шаблоните могат да се използват метасимволите: „*”, „?”, „[]”, които се разширяват по същите правила, както при имена на файлове, но тук се разширяват до произволни думи. Оператор `case` връща кода на завършване на последната изпълнена команда или 0 ако не е намерено нито едно съответствие.

Пример. Фрагмент от командна процедура, който чете отговор на потребителя и го интерпретира като „Да”, „Не” или „Изход”.

```
read answer
case "$answer"  in
[Qq]*|[Ee]*) echo "Answer is Quit"; exit 1;;
[Yy]*| "") echo "Answer is YES";;
[Nn]*) echo "Answer is NO";;
*) echo "Invalid answer";;
esac
```

14. ФУНКЦИИ

Функция с име *име_функция* и тяло *списък_команди* се определя по следния начин:

```
[function]  име_функция () { списък_команди; }
```

Определението на функцията трябва да предшества извикването ѝ. Функция се извиква така както и обикновена команда, чрез името си. Тогава се изпълнява *списък_команди* в контекста на текущия процес shell, т.е. не се създава нов процес shell, както при изпълнение на командна процедура. Това означава, че всички променливи и техните значения, определени в текущия shell са достъпни и при изпълнение на функцията. Единствено позиционните параметри `$1`, `$2`, и т.н. и променливите `$#`, `$*`, `$@` се изменят по време на изпълнението на функцията от аргументите, зададени при извикването ѝ.

Във тялото на функцията може да се определят локални променливи чрез командата `local`. Опциите са същите, както на командата `declare`.

```
local [опции] променлива[=значение]
```

При завършване функцията връща код на завършване, който е кода на последната изпълнена команда в *списък_команди*, ако това не е команда `return`.

```
return [n]
```

Командата `return` явно завършва изпълнението на функцията и връща изпълнението към мястото на викането ѝ. Аргументът *n* (число от 0 до 255) е код на завършване и ако не е указан се използва пак кода на последната изпълнена преди `return` команда. След завършване изпълнението на функцията се възстановяват значенията на временно изменените позиционни параметри и вътрешни променливи (`$1`, `$2`, и т.н., и `$#`, `$*`, `$@`) такива, каквито са били преди извикването. Променливата `$?` ще съдържа кода на завършване, върнат от функцията.

Пример. Командната процедура може да се извиква с произволен брой аргументи (включително и 0), които са имена на потребители. За всеки аргумент изчислява и извежда броя на сесиите на съответния потребител или съобщение ако не съществува такъв потребител. Ако е извикана без аргументи изчислява и извежда общия брой сесии в системата.

```
# userls [user_name . . .]
#-------------------------------------------------
function scnt()
{
if [ $# -eq 0 ]
then
     n=`who | wc -l`
else
     if grep "^$1:" /etc/passwd >/dev/null 2>&1
     then n=`who | grep "^$1 " | wc -l`
     else return 1
     fi
fi
return 0
}
if [ $# -eq 0 ]
then
     scnt
     echo "Number of sessions: $n"
else
     for u
     do
          if scnt $u
          then echo "User $u: $n sessions"
          else echo "No user $u" >&2
          fi
     done
fi
#-------------------------------------------------
```

15. ЕКРАНИРАНЕ

Под термина екраниране се разбира отменяне на специалното значение на символи или думи. Съществуват три начина за това.

- Чрез символа \ (escape character)

Ако не е екраниран символът \ запазва нормалното значение на символа след него, освен ако това не е символ за нов ред. Комбинацията \<newline> се интерпретира като игнориране на символа за нов ред, т.е. като продължение на реда.

```
$ echo \x              # x
$ echo \\x             # \x
$ echo \\\x            # \x
$ echo xxx\            # xxxzzz
> zzz
```

- Чрез двойни кавички - "*низ*"

Запазва се нормалното значение на всички символи в низа, с изключение на $, `, \. Чрез символа \ в низа се екранират само $, `, \, " и <newline>.

```
dd=123
$ echo "\x"                        # \x
$ echo "\\x"                       # \x
$ echo "\\\x"                      # \\x
$ echo "aa\xbb\"cc$dd"             # aa\xbb"cc123
$ echo "aa\xbb\"cc\$dd"            # aa\xbb"cc$dd
```

- Чрез единични кавички - '*низ*'

Запазва се нормалното значение на всички символи в низа.

```
$ echo '\x'            # \x
$ echo '\\x'           # \\x
$ echo '\\\x'          # \\\x
```

16. ЗАМЕСТВАНИЯ И ИЗПЪЛНЕНИЕ НА КОМАНДА

Преди да предаде управлението на команда shell изпълнява определена последователност от действия:

1. Анализира синтаксиса на командата в съответствие със собствените си синтактически правила.
2. Извършва заместания, а именно:
   - Заместване на "~"
   - Заместване на име на променлива със значението
   - Заместване на изхода на команда
   - Аритметично разширение
   - Разширение до имена на файлове
3. Извършва пренасочвания на вход и изход
4. Изпълнява командата, като й предава аргументите след направените заместания.
   При това, ако името на командата не е пълно име на файл, то проверява дали има:
   - Функция с указаното име
   - Вътрешна команда с указаното име
   - Търси файл в каталозите на променливата `PATH` с указаното име.

Изпълнява първата намерена команда. Ако името на командата е пълно име на файл - относително или абсолютно, тогава направо се опитва да изпълни програмата във файла. Ако не намери файла или намерения файл не е изпълним (потребителят няма прово `x`), извежда съобщение за грешка и завършва изпълнението на командата с код съответно 127 или 126.