

Linux Professional Institute

LPIC-1

جلسه دوازدهم: اسکریپت‌نویسی در bash

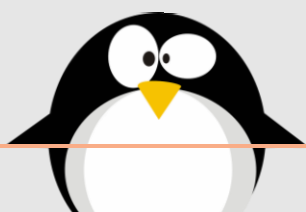
در این جلسه:

ویدئو اول:

- صحبت کلی در مورد اسکریپتینگ
- معرفی زبان برنامه‌نویسی Perl
- آشنایی با ساختار یک اسکریپت Perl
- آشنایی با ساختار و تعریف متغیرها در Perl
- آشنایی با Escape Sequence ها
- صحبت در مورد ماژول strict و warning در Perl

ویدئو دوم:

- صحبت در مورد چگونگی مقایسه‌ی متغیرها در Perl
- صحبت در مورد گزاره‌ی if، حلقه‌ی while، for و foreach در Perl
- صحبت در مورد چگونگی دریافت ورودی از کاربر و عملگر chomp در Perl
- صحبت در مورد چگونگی اجرای دستورات لینوکس در اسکریپت Perl
- صحبت در مورد ساختمان‌های داده و چگونگی کار کردن با آنها در Perl
- صحبت در مورد مفهوم undef در Perl
- صحبت در مورد subroutine ها در Perl
- صحبت در مورد Hash ها در Perl
- صحبت کلی در مورد ساختار ایمیل



نکته:

در این جلسه، به توضیح خیلی سطحی و ابتدایی زبان برنامه‌نویسی Perl پرداخته می‌شود. با توجه به این که توضیحات استاد در این جلسه بسیار سطحی می‌باشد و نیازی به جزوه ندارد، ما در جزوه‌ی این جلسه، به توضیح چگونگی نوشتن اسکریپت‌های bash می‌پردازیم.



فهرست مطالب

۱	مقدمه.....
۱	متغیرهای شِل.....
۱	متغیرهای محیطی گلوبال.....
۲	متغیرهای محیطی لو کال.....
۳	تعریف متغیرهای لو کال.....
۴	تعریف متغیرهای گلوبال.....
۴	پیدا کردن متغیرهای محیطی سیستمی.....
۵	فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شِل لاگین.....
۵	فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شِل تعاملی.....
۵	فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شِل غیر تعاملی.....
۶	استفاده از Aliasها.....
۷	شِل اسکریپتینگ (Shell Scripting).....
۸	اجرای چندین دستور در یک خط.....
۸	Redirect کردن خروجی.....
۹	پایپ کردن اطلاعات.....
۱۰	نوشتن شِل اسکریپت.....
۱۰	Shebang (!#).....
۱۰	قرار دادن کامنت و دستور در شِل اسکریپت‌ها.....
۱۱	اجرای شِل اسکریپت.....
۱۱	شِل اسکریپتینگ پیشرفته.....
۱۲	نمایش پیام روی ترمینال.....
۱۲	استفاده از متغیرها در اسکریپت.....
۱۳	استفاده از متغیرهای محیطی گلوبال.....
۱۳	تعریف متغیرهای لو کال.....
۱۴	آرگمان‌های کامندلاین.....
۱۵	دریافت ورودی از کاربر.....
۱۵	خواندن ورودی با استفاده از read.....
۱۷	تعریف محدودیت زمانی برای ارائه‌ی ورودی.....
۱۸	محدود کردن تعداد کاراکترهای ورودی.....
۱۸	دریافت ورودی بدون نشان دادن کاراکترهای دریافتی روی ترمینال.....

۱۹.....	Exit Status ها
۲۰.....	نوشتن برنامه‌های اسکریپتی
۲۰.....	جایگزینی یک دستور
۲۱.....	انجام عملیات ریاضی
۲۲.....	گزاره‌های منطقی
۲۲.....	گزاره‌ی <i>if</i>
۲۴.....	گزاره‌ی <i>case</i>
۲۶.....	حلقه‌ها
۲۶.....	حلقه‌ی <i>for</i>
۲۹.....	حلقه‌ی <i>while</i>
۳۰.....	فانکشن‌ها
۳۳.....	اجرای اسکریپت‌ها در پشت صحنه
۳۳.....	فرستادن اسکریپت به پشت صحنه
۳۵.....	فرستادن چندین اسکریپت به پشت صحنه
۳۵.....	اجرای اسکریپت‌ها بدون وابستگی به یک کنسول
۳۶.....	ارسال سیگنال
۳۷.....	اینترپت کردن (Interrupt) پراسس
۳۷.....	استاپ کردن (Stop) یک پراسس
۳۷.....	کنترل جاب‌ها
۳۸.....	مشاهده‌ی جاب‌ها
۴۰.....	از سر گیری (زنده کردن) جاب‌های استاپ شده
۴۱.....	اجرای اتوماتیک اسکریپت‌ها در زمان‌های مشخص
۴۱.....	زمان‌بندی اسکریپت با استفاده از دستور <i>at</i>
۴۱.....	استفاده از دستور <i>at</i>
۴۲.....	صف‌های <i>at</i>
۴۲.....	دسترسی به خروجی جاب‌ها
۴۳.....	مشاهده‌ی جاب‌های موجود در صف <i>at</i>
۴۳.....	حذف یک جاب از صف <i>at</i>
۴۳.....	اجرای اسکریپت‌ها در بازه‌های زمانی مشخص

مقدمه

جلسه قبل، با systemd-journald آشنا شدیم و دانش خود را در مورد سیستم‌های لاگینگ در لینوکس بهبود بخشیدیم. سپس در مورد مدیریت و تنظیم ساعت دقیق در لینوکس صحبت کردیم و در نهایت با ابزارهای متفاوت جهت زمان‌بندی عملیات متفاوت در لینوکس آشنا شدیم. در این جلسه با شیل که تا کنون خیلی از آن استفاده کرده‌ایم بیشتر آشنا می‌شویم و در مورد شیل اسکریپت‌ها، دلیل نیاز به آنها، چگونگی ایجاد و مدیریت آنها به طور مفصل صحبت خواهیم کرد.

متغیرهای شیل

قبل از صحبت در مورد اسکریپت‌ها و چگونگی ایجاد آنها، بهتر است در مورد چگونگی ذخیره‌ی اطلاعات توسط شیل صحبت کنیم. همانطور که می‌دانید در اکثر توزیع‌های لینوکسی، شیل پیش‌فرض ما bash نام دارد. bash از متغیرهای محیطی، یا Environment Variableها برای ذخیره‌ی اطلاعات استفاده می‌کند. متغیرهای محیطی، اطلاعات را درون RAM ذخیره می‌کنند و بدین شکل، هر برنامه یا اسکریپتی که توسط شیل اجرا شود، می‌تواند به اطلاعات ذخیره شده در این متغیرها، دسترسی داشته باشد. متغیرهای محلی، مکان بسیار مناسبی برای ذخیره‌ی داده‌هایی نظیر اطلاعات حساب کاربری، سیستم، شیل و هر چیز دیگر می‌باشند. به طور کلی، دو نوع متغیر محیطی در شیل وجود دارد:

- متغیرهای گلوبال (Global Environment Variables)
- متغیرهای لوکال (Local Environment Variables)

در این بخش، می‌خواهیم با این دو نوع متغیر آشنا شده و چگونگی استفاده از آنها را یاد بگیریم.

نکته: ما می‌توانیم در یک سیستم، چندین شیل ایجاد کنیم. برای این کار، کافی است درون شیل کنونی، دستور bash را وارد کنیم تا یک شیل جدید با یک PID جدید برایمان ایجاد شود. این شیل جدید، تفاوت‌هایی با شیلی که قبلاً در آن بودیم خواهد داشت که یکی از این تفاوت‌ها، مربوط به در دسترس بودن متغیرها می‌باشد.

متغیرهای محیطی گلوبال

متغیرهای گلوبال، متغیرهایی هستند که هم توسط شیل و هم توسط کلیه‌ی پراسس‌هایی که توسط شیل ایجاد می‌شوند قابل مشاهده و استفاده می‌باشند. لینوکس به صورت اتوماتیک هنگام استارت شدن یک شیل، تعدادی متغیر گلوبال ایجاد کرده و آنها را مقداردهی می‌کند. به طور کلی، متغیرهای گلوبال سیستمی، با حروف بزرگ مشخص می‌شوند.

برای مشاهده‌ی متغیرهای گلوبال محیطی، می‌توانیم از دستور printenv استفاده کنیم:

```
[root@localhost ~]# printenv
```

```
XDG_SESSION_ID=4
HOSTNAME=localhost.localdomain
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SELINUX_USE_CURRENT_RANGE=
SSH_TTY=/dev/pts/0
QT_GRAPHICSSYSTEM_CHECKED=1
USER=root
MAIL=/var/spool/mail/root
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
PWD=/root
[...]
LANG=en_US.UTF-8
LOGNAME=root
LESSOPEN=||/usr/bin/lesspipe.sh %sXDG_RUNTIME_DIR=/run/user/0
_=/usr/bin/printenv
```

اکثر این متغیرهای گلوبالی که به صورت پیش فرض در سیستم وجود دارند، طی فرآیند روشن شدن شدن سیستم، تعریف و مقداردهی می شوند.

برای مشاهده مقدار اختصاص یافته به یک متغیر محیطی، می توانیم از دستور `echo` و نام متغیر مورد نظر استفاده کنیم. در شِل، برای استفاده از یک متغیر (یعنی مشاهده مقدار آن و یا استفاده از آن)، باید یک علامت دلار (\$) قبل از نام متغیر قرار دهیم. بیا مقدار ذخیره شده در متغیر محیطی `HOME` را با هم مشاهده کنیم:

```
[root@localhost ~]# echo $HOME
/root
```

همانطور که می بینید، برای مشاهده مقدار ذخیره شده در متغیر `HOME`، ابتدا یک علامت \$ قبل از نام متغیر قرار دادیم و سپس با دستور `echo` مقدار آن را مشاهده کردیم.

متغیرهای محیطی لوکال

متغیرهای محیطی لوکال، متغیرهایی هستند که فقط توسط شِلی که آنها را تعریف کرده قابل مشاهده و دسترسی می باشند. یعنی اگر یک شِل جدید ایجاد کنیم (مثلا با وارد کردن دستور `bash` در شِل)، متغیر لوکالی که تعریف کرده بودیم قابل دسترسی نخواهد بود. متأسفانه هیچ دستور خاصی برای مشاهده متغیرهای محیطی لوکال وجود ندارد؛ اما ما می توانیم با استفاده از دستور `set`، کلیدی متغیرهای تعریف شده در شِل، اعم از متغیرهای گلوبال و همچنین لوکال را مشاهده کنیم. برای مثال:

```
[root@localhost ~]# set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:histappend:ho
stcomplete:interactive_comments:login_shell:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
[...]
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
PIPESTATUS=([0]="0")
PPID=1591
PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%%.*}"'
"${PWD/#$HOME/~}"
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/root
QT_GRAPHICSSYSTEM_CHECKED=1
SELINUX_LEVEL_REQUESTED=
UID=0
USER=root
XDG_RUNTIME_DIR=/run/user/0
XDG_SESSION_ID=4
_=/root
colors=/root/.dircolors
```

همانطور که می بینید، در خروجی این دستور، برخی از متغیرهای گلوبالی که در خروجی دستور `printenv` دیدیم نیز قابل مشاهده می باشند. سایر متغیرهای موجود در خروجی، متغیرهای لوکال می باشند.

تعریف متغیرهای لوکال

همانطور که گفتیم، متغیرهای لوکال فقط در شِلی که تعریف شده‌اند قابل دسترسی می‌باشند. متغیرهای شِلی، می‌توانند یک رشته از حروف یا یک عدد صحیح را درون خود ذخیره کنند. برای تعریف یک متغیر لوکال در شِلی، به صورت زیر عمل می‌کنیم:

```
[root@localhost ~]# my_name=behnam
[root@localhost ~]# my_age=99
```

همانطور که می‌بینید، ما ابتدا نام متغیر مورد نظر را نوشته، سپس یک علامت = قرار داده و در مقابل آن، مقداری که می‌خواستیم به این متغیر اختصاص دهیم را مشخص کردیم. نکته‌ای که باید به آن توجه کنید این است که هنگام مقداردهی به یک متغیر، نباید هیچ فاصله‌ای بین نام متغیر، علامت = و مقدار اختصاص داده شده به متغیر وجود داشته باشد، وگرنه شِلی به ما پیغام خطا می‌دهد.

بیاید از صحت اختصاص مقادیر مشخص شده به این متغیرها اطمینان حاصل کنیم:

```
[root@localhost ~]# echo $my_name
behnam
[root@localhost ~]# echo $my_age
99
```

حال بیاید یک متغیر جدید به نام full_name ایجاد کرده و نام کامل خود را در آن ذخیره کنیم:

```
[root@localhost ~]# full_name=the albatross
bash: albatross: command not found
```

همانطور که می‌بینید، شِلی این مقدار را از ما قبول نکرد و به ما ارور داد. دلیل این امر، وجود فاصله بین the و albatross بود. وجود این فاصله باعث می‌شود که bash کلمه‌ی موجود پس از فاصله را به عنوان یک دستور در نظر گیرد. برای این که یک رشته از حروف که بینشان فاصله وجود دارد را به یک متغیر بدهیم، باید آن رشته را بین دو علامت ' قرار دهیم:

```
[root@localhost ~]# full_name='behnam sajjadi'
[root@localhost ~]# echo $full_name
behnam sajjadi
```

توجه کنید که کلیه‌ی متغیرهایی که تعریف کردیم، از حروف کوچک تشکیل شده بودند. شدیداً پیشنهاد می‌شود که هنگام تعریف متغیرهای لوکال، از حروف کوچک استفاده کنیم، چرا که این امر به ما کمک می‌کند که بتوانیم متغیرهای لوکال را از متغیرهای گلوبال تمیز دهیم.

لازم است بار دیگر یادآوری کنیم که متغیرهای لوکال، فقط در شِلی کنونی قابل استفاده و مشاهده هستند. یعنی:

```
[root@localhost ~]# my_var=123
[root@localhost ~]# echo $my_var
123
[root@localhost ~]# bash
[root@localhost ~]# echo $my_var
```

```
[root@localhost ~]# exit
exit
[root@localhost ~]# echo $my_var
123
```

همانطور که می‌بینید، ما ابتدا یک متغیر به نام my_var ایجاد کردیم و مقدار ۱۲۳ را درون آن ذخیره کردیم. سپس با استفاده از دستور echo، صحت اختصاص این مقدار به متغیر را بررسی کردیم. پس از آن با اجرای دستور bash، یک شِل جدید ایجاد کردیم و سپس سعی کردیم مقدار اختصاص یافته به متغیر my_var را مشاهده کنیم؛ اما همانطور که می‌بینید در خروجی جوابی دریافت نکردیم. دلیل این امر، این است که متغیر my_var یک متغیر محلی بوده و در این شِل جدید، مقداری به آن اختصاص داده نشده است. سپس با نوشتن دستور exit، از این شِل خارج شده و به شِل قبلی باز گشتیم. در این شِل، متغیر my_var هنوز مقدار ۱۲۳ را درون خود ذخیره کرده است.

تعریف متغیرهای گلوبال

متغیرهای گلوبالی که در یک شِل تعریف می‌شوند، توسط همه‌ی پراسس‌های فرزند آن شِل (شِل‌های دیگر ایجاد شده توسط آن شِل) قابل دسترسی می‌باشند. برای ایجاد و تعریف متغیرهای گلوبال، ابتدا یک متغیر لوکال ایجاد می‌کنیم و سپس با استفاده از دستور export، آن متغیر لوکال را تبدیل به یک متغیر گلوبال می‌کنیم. برای مثال:

```
[root@localhost ~]# echo $my_var
123
[root@localhost ~]# export my_var
[root@localhost ~]# bash
[root@localhost ~]# echo $my_var
123
```

همانطور که می‌بینید، ما ابتدا مقدار ذخیره شده در متغیر my_var را مشاهده کردیم. سپس با استفاده از دستور export، متغیر my_var را تبدیل به یک متغیر گلوبال کردیم. پس از آن با وارد کردن دستور bash، یک شِل جدید ایجاد کردیم و در شِل جدید، مقدار ذخیره شده در متغیر my_var را مشاهده کردیم. همانطور که می‌بینید، این متغیر در شِل جدید نیز قابل دسترسی می‌باشد. توجه کنید که هنگام استفاده از دستور export برای تبدیل یک متغیر به متغیر گلوبال، نیازی نیست که قبل از نام متغیر، یک علامت دلار (\$) قرار دهیم.

پیدا کردن متغیرهای محیطی سیستمی

لینوکس از متغیرهای محیطی برای شناسایی خود در برنامه‌ها و اسکریپت‌های متفاوت استفاده می‌کند. ما با بررسی مقدار اختصاص یافته به این متغیرها، می‌توانیم اطلاعاتی در مورد سیستمی که اسکریپت روی آن اجرا می‌شود به دست آوریم. اما اول باید بدانیم که این متغیرها چگونه مقداردهی می‌شوند. هنگامی که داخل یک سیستم لینوکسی لاگین می‌کنیم و یک شِل bash جدید استارت می‌زنیم، bash به صورت اتوماتیک داخل چندین فایل، به جستجوی دستورهای متفاوت می‌پردازد. به این فایل‌ها، فایل‌های Startup می‌گویند. این که bash کدام فایل‌های استارت‌آپ را پردازش کند، کاملاً بستگی به چگونگی استارت زدن یک شِل bash خواهد داشت. به طور کلی ما می‌توانیم به ۳ روش bash را استارت بزنیم:

- استارت زدن bash به عنوان شِل پیش‌فرض کاربر هنگام لاگین
- استارت زدن bash به عنوان یک شِل تعاملی (Interactive) که هنگام لاگین ایجاد نشده
- استارت زدن bash به عنوان یک شِل غیرتعاملی (non-Interactive) برای اجرای یک اسکریپت

در ادامه‌ی این بخش، به بررسی فایل‌های استارت‌آپ خوانده شده توسط bash، با توجه به چگونگی استارت شل می‌پردازیم.

فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شل لاگین

زمانی که درون یک سیستم لینوکسی لاگین می‌کنیم، bash به عنوان یک شل لاگین استارت می‌شود. شل لاگین به دنبال چهار فایل Startup متفاوت برای پردازش دستورها می‌گردد. این فایل‌ها، به ترتیب زیر پردازش می‌شوند:

- /etc/profile
- \$HOME/.bash_profile
- \$HOME/.bash_login
- \$HOME/.profile

فایل /etc/profile، فایل استارت‌آپ پیش‌فرض و اصلی شل bash می‌باشد. هر گاه که داخل یک سیستم لینوکس لاگین می‌کنیم، bash دستورات موجود در این فایل را اجرا می‌کند. توزیع‌های متفاوت لینوکسی، دستورات متفاوتی را درون این فایل قرار می‌دهند.

سایر فایل‌ها، برای این هستند که هر کاربر، بتواند متغیرهای محیطی مخصوص به محیط خود را ایجاد کرده و شل را برای خود، شخصی‌سازی کند. نکته‌ی قابل توجه این است که اکثر سیستم‌های لینوکسی، فقط از یکی از سه فایل زیر استفاده می‌کنند:

- \$HOME/.bash_profile
- \$HOME/.bash_login
- \$HOME/.profile

\$HOME به موقعیت دایرکتوری Home هر کاربر اشاره می‌کند. کاربران می‌توانند با تغییر این فایل‌ها، متغیرهای محلی که می‌خواهند هنگام استارت شدن bash ایجاد شود را به bash معرفی کنند.

نکته: توجه کنید که این فایل‌ها، با دلیل وجود نقطه در ابتدای اسمشان، فایل‌های مخفی یا Hidden می‌باشند و اجرای دستور ls، بدون هیچ آپشنی، این فایل‌ها را به ما نشان نمی‌دهد.

فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شل تعاملی

اگر bash را بدون لاگین کردن درون سیستم استارت بزنیم (یعنی مثلاً با تایپ کردن bash در یک شل)، ما چیزی که به آن یک شل تعاملی می‌گویند را استارت زده‌ایم. رفتار این شل کمی با شل لاگین متفاوت می‌باشد، اما ما هنوز می‌توانیم دستورات خود را در آن وارد کنیم.

اگر bash را به عنوان یک شل تعاملی استارت بزنیم، بش فایل /etc/profile را نمی‌خواند و به جای آن، فایل \$HOME/.bashrc را می‌خواند. فایل bashrc، دو کار انجام می‌دهد؛ اولاً به دنبال فایل /etc/bash.bashrc می‌گردد، که فایلی است که به ما اجازه می‌دهد اسکریپت‌ها و متغیرهای محلی را برای همه‌ی کاربرانی که یک شل تعاملی ایجاد می‌کنند تعریف کنیم، و دوماً به عنوان مکانی برای قرار دادن Alias‌ها و همچنین اسکریپت‌های شخصی کاربر عمل می‌کند.

فایل‌های استارت‌آپ هنگام اجرای bash به عنوان شل غیرتعاملی

شل غیرتعاملی، شلی است که سیستم برای اجرای یک اسکریپت، استارت می‌زند. به عبارت دیگر، کلیدی اسکریپت‌ها درون یک شل غیرتعاملی اجرا می‌شوند. ما نمی‌توانیم داخل این شل دستوری وارد کنیم و به

عبارت دیگر، هیچ تعاملی با این شیل نداریم. البته می‌توانیم کاری کنیم که این شیل هنگام استارت شدن، دستورهای استارت‌آپ خاصی را اجرا کند.

این دستورهای استارت‌آپ، باید داخل یک فایل قرار گیرند. ما این فایل را با استفاده از متغیر محلی BASH_ENV به bash معرفی می‌کنیم. زمانی که bash به عنوان شیل غیرتعاملی اجرا می‌شود، مقدار اختصاص داده شده به متغیر محیطی BASH_ENV را چک کرده و به دنبال فایل استارت‌آپ ذکر شده در این متغیر می‌گردد و اگر این فایل وجود داشته باشد، دستورات موجود در آن فایل را اجرا می‌کند. در بسیاری از توزیع‌های لینوکس، این متغیر محلی به صورت پیش‌فرض مقداری ندارد.

استفاده از Aliasها

Aliasها به ما اجازه می‌دهند که یک دستور (به همراه آپشن‌ها و پارامترها) را با نام دیگری (یعنی یک نام مستعار) اجرا کنیم. این امر به ما کمک می‌کند که بتوانیم دستورهای طولانی را بسیار کوتاه‌تر کرده و میزان تایپ خود را کم کنیم. با این که Aliasها متغیرهای محیطی نیستند، اما بسیار شبیه به آنها عمل می‌کنند. اکثر توزیع‌های لینوکس، به صورت پیش‌فرض یک سری Alias برای خود دارند. برای مشاهده‌ی Aliasهای موجود در سیستم، از دستور alias به همراه آپشن -p استفاده می‌کنیم:

```
[root@localhost ~]# alias -p
alias cp='cp -i'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

همانطور که می‌بینید، خروجی این دستور به ما می‌گوید که چه دستوری، جایگزین چه دستوری شده است. برای مثال، در خط اول، می‌بینیم که دستور cp، جایگزین دستور cp -i شده است. این بدین معنی است که ما هر بار داخل شیل دستور cp را وارد می‌کنیم، در واقع دستور cp -i اجرا می‌شود.

ما می‌توانیم با استفاده از دستور alias، به سادگی Aliasهایی برای خودمان ایجاد کنیم:

```
[root@localhost ~]# alias list="ls -l"
```

همانطور که می‌بینید، برای معرفی یک Alias جدید، کافی است دستور alias را وارد کرده، سپس نام مستعار مورد نظر خود را وارد کرده، سپس یک علامت = قرار داده و در مقابل آن، دستوری که می‌خواهیم در ازای وارد کردن نام مستعار اجرا شود را وارد کنیم.

در اینجا، وارد کردن دستور list در شیل، در واقع دستور ls -l را اجرا می‌کند:

```
[root@localhost ~]# list
db_backup.sh
file.txt
ftp.sh
```

بدین ترتیب، با تعریف یک نام مستعار، می‌توانیم یک دستور را فقط با نام مستعارش صدا بزنیم.

البته Alias ها مثل متغیرهای محیطی لوکال عمل می کنند و فقط در شِلی که در آن تعریف شده اند کار می کنند.
برای مثال:

```
[root@localhost ~]# bash
[root@localhost ~]# list
bash: list: command not found
```

به نظر شما ما چگونه می توانیم کاری کنیم که یک Alias در کلیه ی شِلی ها وجود داشته باشد؟ همانطور که از قبل می دانیم، bash همیشه هنگام استارت یک شِلی تعاملی جدید، فایل `$HOME/.bashrc` را می خواند، بنابراین کافی است دستور مربوط به ایجاد alias را داخل این فایل قرار داده تا در کلیه ی شِلی های استارت شده توسط خودمان، آن Alias تعریف شده باشد.
بیا این کار را با هم انجام دهیم:

```
[root@localhost ~]# vim .bashrc
# .bashrc

# User specific aliases and functions
```

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias list='ls -l'
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
```

همانطور که می بینید، از قبل چندین Alias در این فایل تعریف شده بود. ما Alias مربوط به list را به شکلی که می بینید به فایل اضافه کردیم. پس از ذخیره ی این فایل، می توانیم وارد هر شِلی تعاملی شده و به این Alias دسترسی داشته باشیم (البته شِلی کنونی به این Alias دسترسی نخواهد داشت، چون شِلی فقط هنگام استارت شدن فایل `.bashrc` را می خواند، پس باید یک شِلی جدید ایجاد کنیم):

```
[root@localhost ~]# bash
[root@localhost ~]# list
db_backup.sh
file.txt
ftp.sh
[root@localhost ~]# bash
[root@localhost ~]# list
db_backup.sh
file.txt
ftp.sh
```

همانطور که می بینید، اکنون Alias ایجاد شده در همه ی شِلی ها قابل دسترسی می باشد.

شِلی اسکریپتینگ (Shell Scripting)

شِلی اسکریپت ها، برنامه های کوچکی هستند که می توانند عملیات متفاوت در لینوکس را به صورت اتوماتیک انجام دهند. به عمل نوشتن این اسکریپت ها، شِلی اسکریپتینگ می گویند. شِلی اسکریپت ها با پردازش اطلاعات و ایجاد گزارش هایی که در حالت معمولی نیاز به وارد کردن هزاران دستور دارند، بسیار در زمان صرفه جویی کرده و به سیستم انعطاف پذیری بالایی می دهند.

در این بخش با مقدمات شیل اسکریپتینگ و چگونگی نوشتن آنها آشنا می‌شویم.

اجرای چندین دستور در یک خط

ما تا به اینجا، یک دستور را در یک خط شیل وارد می‌کردیم و نتیجه‌ی آن را می‌دیدیم. لینوکس به ما اجازه می‌دهد که چند دستور متفاوت را در یک خط شیل وارد کرده و نتیجه‌ی آنها را دریافت کنیم. برای این کار، کافی است بین دستورهای مورد نظر از علامت نقطه‌ویرگول (;) استفاده کنیم. برای مثال:

```
[root@localhost ~]# uname ; who
Linux
root pts/0 2021-02-20 11:56 (192.168.1.10)
```

پس از وارد کردن این دستور، شیل ابتدا دستور اول را اجرا کرده و نتیجه‌ی آنرا نشان می‌دهد، سپس به سراغ دستور دوم رفته، آن را اجرا کرده و نتیجه‌ی آن را پس از نتیجه‌ی دستور اول در خروجی نشان می‌دهد. شاید این قابلیت چیز بسیار ساده‌ای به نظر بیاید، اما این قابلیت، چیزی است که شیل اسکریپتینگ را ممکن می‌سازد.

Redirect کردن خروجی

یکی دیگر از قابلیت‌های شیل اسکریپت‌ها، ذخیره کردن خروجی دستورها در یک فایل می‌باشد. در اکثر مواقع، پس از اجرای یک دستور، می‌خواهیم خروجی آن را برای مشاهده و آنالیز ذخیره کنیم. ما می‌توانیم این کار را با استفاده از Redirect کردن خروجی انجام دهیم.

ما در جلسه‌های قبلی در مورد Redirectorها صحبت کردیم، اما بار دیگر نیز به توضیح آن می‌پردازیم. Redirect کردن خروجی، به ما امکان می‌دهد که خروجی یک دستور که معمولاً روی مانیتور سیستم نشان داده می‌شود را داخل یک فایل ذخیره کنیم. پرواضح است که ذخیره‌ی خروجی یک دستور (مخصوصاً وقتی که خودمان پای سیستم نباشیم) بسیار ما را در عیب‌یابی یا تایید صحت اجرای صحیح یک اسکریپت، یاری خواهد کرد.

برای Redirect کردن خروجی یک دستور، پس از نوشتن نام دستور مورد نظر، از علامت > استفاده کرده و پس از آن، نام فایلی که می‌خواهیم خروجی دستور درون آن ذخیره شود را وارد می‌کنیم. برای مثال:

```
[root@localhost ~]# date > time.txt
[root@localhost ~]# cat time.txt
Sun Feb 21 10:11:35 +0330 2021
```

همانطور که می‌بینید، ما خروجی دستور date را داخل فایل time.txt ریدایرکت کردیم. اگر دقت کنید، می‌بینید که در صورت ریدایرکت کردن خروجی یک دستور درون یک فایل، هیچ چیزی روی صفحه‌ی ما نشان داده نمی‌شود. پس هنگام ریدایرکت کردن، خروجی دستور به صورت کامل داخل یک فایل قرار می‌گیرد.

استفاده از علامت > برای ریدایرکت کردن خروجی یک دستور، همیشه یک فایل جدید ایجاد می‌کند. یعنی اگر حتی فایل مشخص شده پس از علامت > از قبل وجود داشته باشد، آن فایل پاک شده و یک فایل جدید ایجاد می‌شود. اگر بخواهیم خروجی دستور به انتهای یک فایل چسبیده شود، می‌توانیم از علامت >> استفاده کنیم. این علامت باعث می‌شود که خروجی یک دستور، به انتهای محتویات فایل مشخص شده چسبیده شود. نمونه‌ای از این امر را در صفحه‌ی بعد، می‌بینیم.

```
[root@localhost ~]# who >> time.txt
[root@localhost ~]# cat time.txt
Sun Feb 21 10:11:35 +0330 2021
root      pts/0      2021-02-21 10:10 (192.168.1.102)
```

همانطور که می‌بینید، اکنون فایل time.txt، علاوه بر خروجی دستور date، خروجی دستور who را نیز درون خود دارد.

ریدایرکت کردن خروجی یکی از مهم‌ترین موارد در اجرای شیل اسکریپت‌ها می‌باشد، چرا که با کمک آن می‌توانیم اطلاعات زیادی را در مورد چگونگی اجرای اسکریپت‌های خود به دست آوریم.

نکته: اگر از قبل به خاطر داشته باشید، گفته بودیم که علامت > و >> فقط خروجی دستور را در صورت اجرای صحیح ریدایرکت می‌کنند. این خروجی در واقع روی STDOUT نوشته می‌شود. به عبارت دیگر، این علامت در صورت وجود هر گونه خطا در اجرای دستور، عمل ریدایرکشن را انجام نمی‌دهد، چرا که خروجی دستوری که به خطا برخورد کرده است، روی STDERR نوشته می‌شود. برای ریدایرکت کردن STDERR، باید از علامت >2 یا >>2 استفاده کنیم. این باعث می‌شود که ما بتوانیم خطاهای ایجاد شده توسط یک دستور را نیز ذخیره کنیم.

پایپ کردن اطلاعات

پایپ کردن به ما امکان می‌دهد که خروجی یک دستور را به ورودی یک دستور دیگر بدهیم. در واقع دستور دوم، از خروجی دستور اول به عنوان ورودی خود استفاده می‌کند. این قابلیت بسیار قدرتمند می‌باشد و ما را در نوشتن شیل اسکریپت‌ها یاری می‌دهد.

برای پایپ کردن خروجی یک دستور، از علامت | استفاده می‌کنیم. برای مثال:

```
[root@localhost ~]# ls -l
hello
just
stuff
testing
[root@localhost ~]# ls -l | sort -r
testing
stuff
just
hello
```

همانطور که می‌بینید، پس از وارد کردن این دستور، خروجی دستور ls به عنوان ورودی به دستور sort داده شد. بدین ترتیب، ما خروجی دستور ls را روی صفحه نمی‌بینیم، بلکه فقط خروجی دستور sort را روی صفحه مشاهده می‌کنیم. لازم به ذکر است که ما هیچ محدودیتی از نظر تعداد دستورهای که می‌توانیم پایپ کنیم نداریم.

نکته: علامت‌های > و >> | کاراکترهایی هستند که به آنها متاکاراکتر می‌گویند. متاکاراکترها، کاراکترهایی هستند که برای شیل لینوکس، معنای خاصی دارند. اگر بخواهیم از یکی از متاکاراکترها در در اسم یک فایل ... استفاده کنیم، باید آنها را با استفاده از علامت \ مشخص کرده یا آن کاراکتر را داخل دو علامت ' یا " قرار دهیم. به این کار، Escape کردن می‌گویند.

نوشتن شیل اسکریپت

لینوکس به ما اجازه می‌دهد که چندین دستور متفاوت را داخل یک فایل قرار داده و آن فایل را به عنوان یک دستور از ترمینال، اجرا کنیم. به این فایل‌ها، شیل اسکریپت می‌گویند. شیل اسکریپت‌ها فایل‌های متنی معمولی می‌باشند، پس برای ایجاد آنها، کافی است از یک ادیتور، نظیر vim، emacs و... استفاده کنیم. دقت کنید که استفاده از پردازشگرهای متن، مثل LibreOffice Writer و... برای این کار مناسب نیست و نوشتن اسکریپت درون آنها، باعث می‌شود که اسکریپت به درستی اجرا نشود.

Shebang (#!)

برای این که شیل اسکریپتی که ایجاد می‌کنیم به درستی اجرا شود، باید فایل شیل اسکریپت از یک فرمت خاص پیروی کند. اولین خط موجود در فایل شیل اسکریپت باید شلی که برای اجرای این اسکریپت نیاز داریم (مثلا zsh، bash و...) را مشخص کند. این خط، به صورت زیر نوشته می‌شود:

```
#!/bin/bash
```

همانطور که می‌بینید، این خط در نگاه اول کمی عجیب به نظر می‌آید. در لینوکس، به علامت #! Shebang می‌گویند. این علامت به سیستم عامل اطلاع می‌دهد که از چه شلی برای اجرای اسکریپت استفاده کند. سیستم‌های لینوکس از چندین شیل متفاوت پشتیبانی می‌کنند (bash، zsh، csh و...)، اما معروف‌ترین و پرکاربردترین شیل، bash می‌باشد. ما می‌توانیم شیل اسکریپت‌های نوشته شده برای سایر شیل‌ها را نیز درون سیستم اجرا کنیم، به شرطی که آن شیل در سیستم نصب باشد.

قرار دادن کامنت و دستور در شیل اسکریپت‌ها

پس از مشخص کردن نوع شیل با استفاده از علامت #! می‌توانیم دستورهایی که می‌خواهیم توسط اسکریپت اجرا شود را مشخص کنیم. در اینجا، ما می‌توانیم هر دستور را درون یک خط جدید بنویسیم. در صورت قرار دادن هر دستور در یک خط جدید، نیازی به استفاده از علامت ; در انتهای هر دستور نداریم. بیایید یک شیل اسکریپت را با یکدیگر ایجاد کنیم:

```
[root@localhost ~]# cat script1.sh
#!/bin/bash
# This script shows the date and who is logged in
date
who
```

همانطور که می‌بینید، در خط اول این اسکریپت، ما خط Shebang را قرار دادیم که به سیستم می‌گوید که این اسکریپت، برای شیل bash نوشته شده است. خط دوم این اسکریپت، یکی دیگر از قابلیت شیل اسکریپت‌ها، یعنی قابلیت قرار دادن کامنت‌ها را نشان می‌دهد. خط‌هایی که با کاراکتر # شروع می‌شوند، کامنت نام دارند و به ما این امکان را می‌دهند که به زبان انسانی، توضیحاتی در مورد عملکرد اسکریپت درون فایل قرار دهیم. شیل خط‌های کامنت را نمی‌خواند و از آنها رد می‌شود. ما پس از قرار دادن Shebang، می‌توانیم در هر کجای اسکریپت یک کامنت قرار دهیم. خط‌های ۳ و ۴ موجود در فایل اسکریپت، به شیل می‌گویند که ابتدا دستور date را اجرا کرده و سپس دستور who را اجرا کند.

اجرای شیل اسکریپت

شاید فکر کنید برای اجرای شیل اسکریپت، کافی است نام آن را مثل یک دستور وارد کرده و دکمه‌ی Enter را بزنیم. بیایید این کار را امتحان کنیم:

```
[root@localhost ~]# script1.sh
-bash: script1.sh: command not found
```

همانطور که می‌بینید با وارد کردن نام فایل اسکریپت به عنوان یک دستور، پیغامی مبنی بر عدم موفقیت شیل در پیدا کردن این دستور دریافت می‌کنیم. دلیل این امر این است که هنگام وارد کردن یک دستور، bash در یک سری دایرکتوری خاص، به دنبال آن دستور گشته و از آنجایی که اسکریپت ما در آن دایرکتوری‌های خاص قرار ندارد، نمی‌تواند آن را پیدا کند. این دایرکتوری‌های خاص، توسط یک متغیر محلی خاص به نام PATH تعریف می‌شوند.

از آنجایی که دایرکتوری Home ما داخل PATH قرار ندارد، نمی‌توانیم اسکریپت را به صورت مستقیم اجرا کنیم. بنابراین، باید از یک روش دیگر برای اجرای اسکریپت خود استفاده کنیم. ساده‌ترین روش، استفاده از آدرس relative اسکریپت می‌باشد. برای این کار، کافی است قبل از نام اسکریپت، علامت / را قرار دهیم. همانطور که از قبل می‌دانید، به معنای دایرکتوری که هم‌اکنون درون آن هستیم می‌باشد و علامت / به ما این امکان را می‌دهد که نام یک فایل درون دایرکتوری کنونی را مشخص کنیم. پس برای اجرای اسکریپت، باید نام اسکریپت را به صورت زیر وارد کنیم:

```
[root@localhost ~]# ./script1.sh
-bash: ./script1.sh: Permission denied
```

همانطور که می‌بینید باز هم اسکریپت ما اجرا نشد. این بار خطای دریافت شده به ما می‌گوید که مشکلی در مجوزها وجود دارد. بیایید نگاهی به مجوزهای فایل script1.sh بیاندازیم:

```
[root@localhost ~]# ls -l script1.sh
-rw-r--r--. 1 root root 72 Feb 21 11:00 script1.sh
```

همانطور که می‌بینید، فایل script1.sh مجوز اجرا ندارد. لینوکس به صورت پیش‌فرض به هیچ فایلی مجوز اجرا نمی‌دهد، اما ما می‌توانیم با استفاده از دستور chmod، مجوز اجرا را به این فایل بدهیم. پس:

```
[root@localhost ~]# chmod u+x script1.sh
[root@localhost ~]# ls -l script1.sh
-rwxr--r--. 1 root root 72 Feb 21 11:00 script1.sh
```

همانطور که می‌بینید، با استفاده از دستور chmod، مجوز اجرا (x) را به مالک فایل script1.sh اضافه کردیم. حال که این فایل مجوز اجرا دارد، می‌توانیم آن را اجرا کنیم. پس:

```
[root@localhost ~]# ./script1.sh
Mon Feb 22 11:28:57 +0330 2021
root pts/0 2021-02-22 11:26 (192.168.1.102)
```

همانطور که می‌بینید، این بار اسکریپت ما به صورت کامل اجرا شد.

شیل اسکریپتینگ پیشرفته

در بخش قبل به بررسی مباحث اولیه در ایجاد شیل اسکریپت‌ها پرداختیم و دیدیم که می‌توانیم هر تعداد دستور را داخل یک فایل اسکریپت قرار داده و آن فایل را اجرا کنیم. در این بخش، به بررسی مباحث پیشرفته‌تر در شیل

اسکرپتینگ می‌پردازیم. این مباحث به ما کمک می‌کنند که شل اسکرپت‌هایی بنویسیم که مانند برنامه‌های واقعی کار می‌کنند.

نمایش پیام روی ترمینال

همانطور که می‌دانید، ما خیلی از اوقات چندین دستور را پشت‌سر هم درون یک فایل اسکرپت قرار می‌دهیم. آنالیز خروجی یک اسکرپت که چندین دستور را پشت سر هم اجرا می‌کند می‌تواند کار گیج‌کننده و دشواری باشد. ما می‌توانیم بین دستورهای موجود در فایل اسکرپت، پیام‌هایی را در خروجی نشان دهیم که ما را در درک خروجی یاری می‌دهند.

دستور echo به ما اجازه می‌دهد که پیام‌هایی را روی ترمینال نشان دهیم. استفاده از این دستور به صورت تکی روی ترمینال، فقط پیامی که به ورودی آن داده‌ایم را در خروجی به ما نشان می‌دهد. یعنی:

```
[root@localhost ~]# echo this is boring
this is boring
```

پس استفاده از این دستور داخل شل اسکرپت، باعث می‌شود که پیام دلخواه ما هنگام اجرای اسکرپت روی ترمینال نمایش داده شود. بیایید اسکرپتی که در بخش قبل نوشتیم را با اضافه کردن مواردی، بهبود بخشیم:

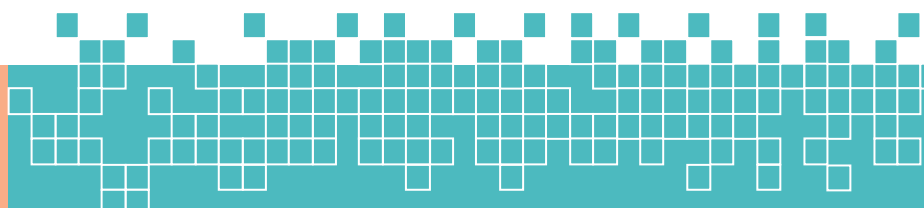
```
[root@localhost ~]# cat script2.sh
#!/bin/bash
# This script shows the date and who is logged in
```

```
echo Currently, the date and time is:
date
echo
echo "Let's view the people logged into the system:"
who
```

همانطور که می‌بینید، ما سه دستور echo به اسکرپت script1.sh اضافه کردیم و آن را در یک فایل جدید به نام script2.sh ذخیره کردیم. ما در اولین دستور echo، پیام خود را بین دو علامت " قرار ندادیم، اما در سومین دستور echo، پیام خود را بین دو علامت " قرار دادیم. دلیل این کار، این است که در متنی که به سومین دستور echo دادیم، علامت ' در متن ما وجود دارد (در کلمه‌ی Let's). علامت ' یک متاکاراکتر می‌باشد و اگر متن خود را بین " قرار نمی‌دادیم، شل موفق به اجرای این دستور نمیشد و به ما خطا می‌داد. اگر دقت کنید، می‌بینید که هیچ نوشته‌ی جلوی دومین دستور echo وجود ندارد. این باعث می‌شود که شل، یک خط خالی ایجاد کند و به عبارتی، خروجی نمایش داده شده توسط دستور قبلی و بعدی را از یکدیگر جدا کرده و خواندن آن را ساده‌تر کند.

استفاده از متغیرها در اسکرپت

یکی از مهم‌ترین موارد در برنامه‌نویسی، قابلیت ایجاد متغیرهایی برای ذخیره‌ی موقت یک سری از اطلاعات می‌باشد. ما در بخش‌های قبلی با متغیرهای محیطی گلوبال و لوکال و چگونگی ایجاد آنها آشنا شدیم. ما می‌توانیم از این متغیرها در اسکرپت‌های خود نیز استفاده کنیم. در این بخش، به بررسی چگونگی استفاده از متغیرها در اسکرپت می‌پردازیم.



استفاده از متغیرهای محیطی گلوبال

همانطور که قبلاً نیز گفتیم، از متغیرهای محیطی گلوبال برای نگهداری اطلاعاتی خاصی در مورد سیستم استفاده می‌شود. این اطلاعات، مواردی نظیر نام سیستم، نام کاربر کنونی، موقعیت دایرکتوری Home کاربر کنونی و... می‌باشند. ما می‌توانیم این متغیرهای گلوبال تعریف شده در سیستم را در اسکریپت‌های خود به کار ببریم.

برای دسترسی به این متغیرها، کافی است یک علامت دلار (\$) قرار داده و سپس نام متغیر مورد نظر را بنویسیم. برای مثال:

```
[root@localhost ~]# cat script3.sh
```

```
#!/bin/bash
```

```
# Using global variables
```

```
echo User info for user $USER
```

```
echo UID: $UID
```

```
echo Home directory: $HOME
```

همانطور که می‌بینید، ما از متغیرهای محیطی گلوبال \$USER، که معمولاً نام یوزرنیم کاربر را درون خود نگهداری می‌کند، \$UID که UID یوزرنیم کاربر را درون خود نگهداری می‌کند و همچنین \$HOME که موقعیت دایرکتوری Home یوزر کنونی را درون خود نگهداری می‌کند در اسکریپتمان استفاده کردیم. بیا این اسکریپت را اجرا کرده و نگاهی به خروجی آن بیاندازیم. فراموش نکنید که ما باید به اسکریپت‌های خود مجوز اجرا بدهیم:

```
[root@localhost ~]# chmod u+x script3.sh
```

```
[root@localhost ~]# cat script3.sh
```

```
User info for user root
```

```
UID: 0
```

```
Home directory: /root
```

همانطور که می‌بینید، این اسکریپت اطلاعاتی در مورد یوزری که این اسکریپت را اجرا کرده، یعنی در اینجا یوزر روت، به ما می‌دهد.

تعریف متغیرهای لوکال

برای ذخیره‌ی اطلاعات به صورت موقت درون شِل اسکریپت، از متغیرهای لوکال استفاده می‌کنیم. برای تعریف یک متغیر لوکال و اختصاص یک مقدار به آن، کافی است نام مورد نظر برای متغیر را نوشته، یک علامت = قرار داده و سپس مقداری که می‌خواهیم به متغیر اختصاص دهیم را مشخص کنیم. یعنی چیزی شبیه:

```
pi=3.14
```

```
banner="Welcome to Web Server 1"
```

```
ip_address="192.168.1.1"
```

بر خلاف زبان‌هایی مثل C، Java و...، نیازی نیست که نوع یک متغیر را مشخص کنیم و bash به صورت اتوماتیک نوع هر متغیر را تشخیص می‌دهد.

حال بیا یک شِل اسکریپت جدید با متغیرهای لوکال بنویسیم:

```
[root@localhost ~]# cat script4.sh
```

```
#!/bin/bash
```

```
# Defining variables
```



```
days=5
user="Behnam"
echo "It's been $days days since $user logged in."
```

همانطور که می‌بینید، ما ابتدا دو متغیر تعریف کردیم و از آنها در دستور echo استفاده کردیم. دقت کنید که برای اختصاص یک مقدار به متغیر، نباید هیچ فاصله‌ای بین نام متغیر، علامت = و مقدار مورد نظر وجود داشته باشد. علاوه بر این، به خاطر داشته باشید که برای استفاده از یک متغیر، حتما باید در ابتدای نام آن یک علامت \$ قرار دهیم.

بیا این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script4.sh
[root@localhost ~]# ./script4.sh
It's been 5 days since Behnam logged in.
```

توجه داشته باشید که پس از اجرای شیل اسکریپت، متغیرهای تعریف شده درون شیل اسکریپت توسط ترمینال کنونی قابل دسترسی نمی‌باشند. یعنی اگر در اینجا دستور echo \$days را وارد کنیم، نتیجه‌ای دریافت نخواهیم کرد. اگر بخواهیم این متغیرها پس از اجرای شیل اسکریپت نیز قابل دسترسی باشند، باید آنها را تبدیل به متغیرهای گلوبال کنیم که این کار با دستور export قابل انجام می‌باشد. ما قبلاً در مورد این امر صحبت کردیم و دیگر به توضیح آن نمی‌پردازیم.

آرگمان‌های کامندلاین

یکی از قابلیت‌های بسیار مناسب شیل اسکریپت‌ها، امکان ارائه‌ی اطلاعات به یک شیل اسکریپت هنگام اجرای آن اسکریپت می‌باشد. این امر به ما کمک می‌کند در هر بار اجرا، اطلاعات جدیدی را به اسکریپت بدهیم. یکی از روش‌های ارائه‌ی اطلاعات به یک شیل اسکریپت، استفاده از آرگمان‌های کامندلاین می‌باشد. این آرگمان‌ها، اطلاعاتی هستند که هنگام وارد کردن نام یک اسکریپت در کامندلاین (ترمینال)، رو به روی نام اسکریپت قرار می‌دهیم. یعنی چیزی شبیه:

```
command argument1 argument2 ...
```

همانطور که می‌بینید، آرگمان‌های کامندلاین پس از نام دستور یا اسکریپت نوشته می‌شوند و بین آرگمان‌ها و دستور یا اسکریپت مورد نظر، یک فاصله‌ی خالی قرار می‌گیرد.

ما این آرگمان‌ها را با استفاده از متغیرهای موقعیتی خاص موجود در شیل، دریافت می‌کنیم. برای مثال، از \$1 برای به دست آوردن اولین آرگمان استفاده می‌کنیم، از \$2 برای به دست آوردن دومین آرگمان استفاده می‌کنیم و به همین ترتیب تا کلمه‌ی آرگمان‌ها به دست آورده شوند.

بیا یک اسکریپت که از آرگمان‌های دریافتی در کامندلاین استفاده می‌کند بنویسیم:

```
[root@localhost ~]# cat script5.sh
#!/bin/bash
# Learning about command line arguments

echo "$1 is $2 years old!"
[root@localhost ~]# chmod u+x script5.sh
[root@localhost ~]# ./script5.sh Behnam 99
Behnam is 99 years old!
[root@localhost ~]# ./script5.sh Amir 10
Amir is 10 years old!
```



همانطور که می‌بینید، اسکریپت script5.sh از دو آرگمان کامندلاین استفاده می‌کند. در این اسکریپت، متغیر \$1 نام شخص و متغیر \$2 سن شخص را درون خود ذخیره می‌کند. هنگام اجرای script5.sh، باید حتما مقداری که می‌خواهیم به این دو متغیر اختصاص داده شود را وارد کنیم. اگر یکی یا هیچ کدام از این دو متغیر را وارد نکنیم، شِل به ما پیغام خطایی نمی‌دهد، اما چیزی که مد نظر ماست در خروجی به ما نشان داده نمی‌شود:

```
[root@localhost ~]# ./script5.sh
is years old!
```

همانطور که می‌بینید، با این که آرگمانی به اسکریپت ندادیم، باز هم بدون هیچ مشکلی اجرا شد. چک کردن این که آرگمانی دریافت کردیم یا نکردیم با استفاده از گزاره‌های پیشرفته‌تر قابل انجام می‌باشد.

دریافت ورودی از کاربر

آرگمان‌های کامندلاین به ما کمک می‌کنند آپشن‌ها و پارامترهایی را هنگام اجرای اسکریپت از کاربر دریافت کنیم، اما بعضا لازم است اسکریپت ما قابلیت تعامل با کاربران را داشته باشد. پیش می‌آید که بخواهیم حین اجرای اسکریپت، سوالی از کاربر پرسیده و پاسخ دریافتی از او را درون اسکریپت به کار ببریم. دستور read ما را در انجام این کار یاری می‌دهد.

خواندن ورودی با استفاده از read

دستور read، می‌تواند از STDIN سیستم (کیبورد) یا هر File Descriptor دیگر، ورودی دریافت کند. پس از دریافت ورودی، دستور read اطلاعات دریافتی را درون یک متغیر قرار می‌دهد. بیایید یک اسکریپت که از این دستور استفاده می‌کند بنویسیم:

```
[root@localhost ~]# cat script6.sh
#!/bin/bash
# Testing the read command

echo -n "Enter your name: "
read name
echo "Hello $name, we have been waiting for you!"
[root@localhost ~]# chmod u+x script6.sh
[root@localhost ~]# ./script6.sh
Enter your name: Behnam
Hello Behnam, we have been waiting for you!
```

شاید اولین سوالی که از خود می‌پرسید، این باشد که آپشن -n در دستور echo چه کاری را انجام می‌دهد. در حالت معمولی، ارائه‌ی یک متن به دستور echo و اجرای آن، باعث می‌شود که پس از نشان دادن متن روی ترمینال، یک خط جدید ایجاد کند. ارائه‌ی آپشن -n، باعث می‌شود که echo این خط جدید را پس از نمایش متن ارائه شده در ورودی، ایجاد نکند. بدین صورت، اسکریپت ما حالت یک فُرم را به خود می‌گیرد. حتما در سیستم خود، دستور echo را با و بدون آپشن -n اجرا کنید تا عملکرد این آپشن را به خوبی درک کنید.

پس از دستور echo، ما از دستور read برای گرفتن ورودی از کاربر استفاده کردیم. همانطور که می‌بینید، پس از نوشتن دستور read، ما متغیری به نام name را جلوی این دستور قرار دادیم. این باعث می‌شود که دستور read ورودی دریافتی را درون متغیری به نام name ذخیره کند.



نکته‌ی جالب در مورد دستور read این است که با استفاده از آپشن -p، می‌توانیم یک پرامپت ایجاد کرده و بدین ترتیب نیازی به استفاده از دستور echo نخواهیم داشت. برای مثال:

```
[root@localhost ~]# cat script7.sh
#!/bin/bash
# Testing the read -p option

read -p "Enter your full name: " name
echo "Glad to meet you, $name!"
[root@localhost ~]# chmod u+x script7.sh
[root@localhost ~]# ./script7.sh
Enter your full name: Behnam Albatross
Glad to meet you, Behnam Albatross!
```

همانطور که می‌بینید، استفاده از آپشن -p دستور read باعث شد که نیازی به استفاده از echo نداشته باشیم و خود دستور read، سوال مورد نظر ما را از کاربر پرسید. همانطور که می‌بینید، دقیقاً پس از مشخص کردن سوالی که می‌خواهیم از کاربر بپرسیم، نام متغیری که می‌خواهیم ورودی کاربر درون آن ذخیره شود را مشخص کردیم.

اگر کمی به اسکرپت بالا توجه کنید، می‌بینید که هنگام وارد کردن نام کامل خود، دستور read هم نام و هم نام خانوادگی ما را داخل یک متغیر به نام name قرار داد. دستور read به ما این امکان را می‌دهد که موارد وارد شده را در بیش از یک متغیر نیز بریزیم. این دستور می‌تواند مقادیر وارد شده در پرامپت را با توجه به فاصله‌ی خالی بین آنها، داخل چندین متغیر متفاوت بریزد. اگر تعداد متغیرهای مشخص شده کمتر از تعداد موارد وارد شده باشد، آخرین متغیر مشخص شده، سایر موارد وارد شده را درون خود جا می‌دهد. اگر موارد وارد شده کمتر از متغیرهای مشخص شده باشد، به متغیرهای باقی مانده هیچ مقداری اختصاص نمی‌یابد.

بیا این امر را در یک اسکرپت امتحان کنیم:

```
[root@localhost ~]# cat script8.sh
#!/bin/bash
# Testing multiple variable entry

read -p "Tell us your full name: " first last
echo "You entered $first as your first name"
echo "and $last as your last name."
[root@localhost ~]# chmod u+x script8.sh
[root@localhost ~]# ./script8.sh
Tell us your full name: Behnam Albatross
You entered Behnam as your first name
and Albatross as your last name.
```

همانطور که می‌بینید، این بار پس از مشخص کردن پرامپت دستور read، نام دو متغیر را قرار دادیم و سپس با استفاده از دستور echo، این دو متغیر را روی صفحه نشان دادیم. بیا ایند هنگام اجرای این دستور، فقط نام خود را وارد کنیم:

```
[root@localhost ~]# ./script8.sh
Tell us your full name: Behnam
You entered Behnam as your first name
and as your last name
```



همانطور که می‌بینید، در اینجا اسکریپت هیچ مقداری را به عنوان نام خانوادگی نشان نداد و به عبارت دیگر، مقداری را به متغیر `last` اختصاص نداد.

بیا این بار علاوه بر نام و نام خانوادگی، پیشوند و پسوند اسم را نیز وارد کنیم:

```
[root@localhost ~]# ./script8.sh
Tell us your full name: Behnam Madlad Albatross
You entered Behnam as your first name
and Madlad Sajjadi as your last name.
```

همانطور که می‌بینید، در صورتی که موارد وارد شده در پرامپت بیشتر از تعداد متغیرهای تعریف شده باشد، `bash` کلیه مقادیر باقی مانده را درون آخرین متغیر می‌ریزد. یعنی الان متغیر `last`، رشته‌ی `Madlad Sajjadi` را درون خود ذخیره کرده است.

ما حتی می‌توانیم هنگام استفاده از دستور `read`، هیچ متغیری به آن اختصاص ندهیم. در آن حالت، دستور `read` مقدار وارد شده توسط کاربر را درون متغیر محیطی ویژه‌ی `REPLY` قرار می‌دهد. برای مثال:

```
[root@localhost ~]# cat script9.sh
#!/bin/bash
# Testing the REPLY variable

read -p "Name your favorite animal: "
echo "You're weird. You'd like a $REPLY?"
[root@localhost ~]# chmod u+x script9.sh
[root@localhost ~]# ./script9.sh
Name your favorite animal: frog
You're weird. You'd like a frog?
```

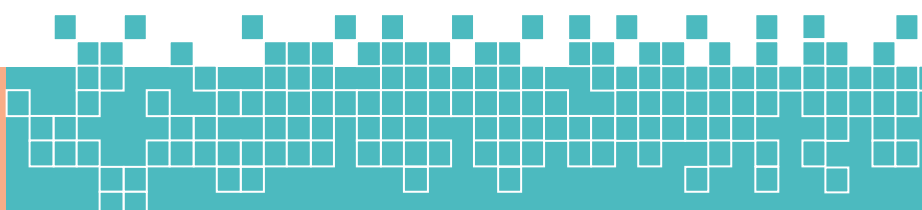
همانطور که می‌بینید، با این که پس از نوشتن دستور `read` هیچ متغیری مشخص نکردیم، این دستور مقدار وارد شده را درون متغیر `REPLY` قرار داد و ما توانستیم از مقدار اختصاص داده شده به این متغیر استفاده کنیم.

تعریف محدودیت زمانی برای ارائه‌ی ورودی

استفاده از دستور `read` یک خطر نیز به همراه خود دارد؛ اسکریپت ما در صورت عدم دریافت ورودی، تا ابد منتظر کاربر برای وارد کردن اطلاعات می‌ماند. ممکن است بخواهیم کاری کنیم که در صورت عدم دریافت ورودی از کاربر در یک بازه‌ی زمانی، اسکریپت بیخیال دریافت ورودی از کاربر شده و به سراغ اجرای سایر خطوط موجود در اسکریپت رود. آپشن `-t` دستور `read` این قابلیت را به ما می‌دهد که یک تایمر برای دستور `read` تعریف کنیم. با استفاده از این آپشن، می‌توانیم مدت زمان انتظار دستور `read` برای دریافت ورودی از کاربر را در واحد ثانیه مشخص کنیم. اگر وارد کردن ورودی بیش از زمان مشخص شده طول بکشد، این دستور به سراغ اجرای سایر دستورهای موجود در اسکریپت می‌رود. برای مثال:

```
[root@localhost ~]# cat script10.sh
#!/bin/bash
# Testing the -t option of read

read -t 5 -p "Do you like scripting? " answer
echo "$answer! Interesting..."
```



همانطور که می‌بینید، ما آپشن `-t` را به دستور `read` داده و پس از آن، عدد ۵ را قرار دادیم. این به `read` می‌گوید که فقط ۵ ثانیه منتظر دریافت ورودی بماند. اگر پس از گذشت ۵ ثانیه ورودی دریافت نشود، سیستم به سراغ انجام خط‌های بعدی رفته تا به انتهای اسکریپت برسد. بیایید این امر را امتحان کنیم:

```
[root@localhost ~]# chmod u+x script10.sh
[root@localhost ~]# ./script10.sh
Do you like scripting? ! Interesting...
```

ما در اینجا پس از اجرای اسکریپت `script10.sh` و مشاهده‌ی پرامپت دستور `read`، هیچ ورودی به اسکریپت ندادیم. پس از گذر ۵ ثانیه، اسکریپت به سراغ اجرای خط‌های بعدی رفت و در نهایت از آن خارج شد. البته اسکریپت ما در این حالت زیاد به درد بخور نیست، چون معمولاً در اسکریپت‌ها می‌خواهیم کاری کنیم که در صورت عدم دریافت ورودی، اسکریپت یک کار دیگر انجام دهد یا یک پیام خطا به کاربر نشان دهد. در بخش‌های بعدی با گزاره‌هایی ما را در انجام این کار یاری می‌دهند آشنا می‌شویم.

محدود کردن تعداد کاراکترهای ورودی

علاوه بر تعریف محدودیت زمانی، می‌توانیم تعداد کاراکترهای دریافتی از کاربر در ورودی را نیز محدود کنیم. یعنی می‌توانیم کاری کنیم که دستور `read` تعداد کاراکترهای دریافتی از کاربر را شمرده و در صورتی که از حدی بالاتر رفت، به صورت اتوماتیک مقدار دریافتی را به یک متغیر اختصاص داده و به سراغ اجرای سایر بخش‌های اسکریپت برود. ما این کار را با استفاده از آپشن `-n` انجام می‌دهیم، به طوری که `ژ` نشان دهنده‌ی حداکثر تعداد کاراکترهای قابل قبول می‌باشد. بیایید این امر را در یک اسکریپت با هم ببینیم:

```
[root@localhost ~]# cat script11.sh
#!/bin/bash
# Testing character limits

read -n3 -p "Enter three of your favorite characters: " chars
echo
echo "Ah, $chars. Interesting choice!"
```

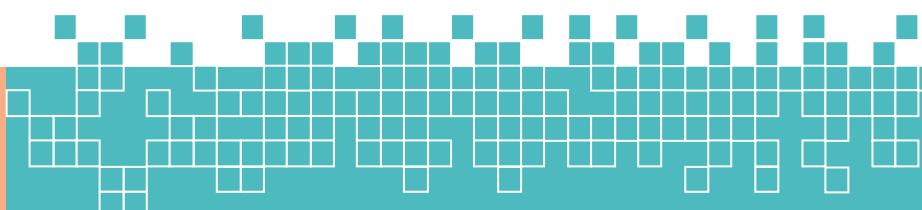
همانطور که می‌بینید، ما با استفاده از آپشن `-n3`، به دستور `read` می‌گوییم که به محض وارد شدن ۳ کاراکتر در ورودی، مقادیر وارد شده را به متغیر `chars` تحویل داده و سپس به سراغ اجرای سایر خطوط اسکریپت رود. بیایید این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script11.sh
[root@localhost ~]# ./script11.sh
Enter three of your favorite characters: BEH
Ah, BEH. Interesting choice!
```

همانطور که می‌بینید، به محض وارد کردن سومین کاراکتر (در اینجا، کاراکتر `H`)، دیگر اجازه‌ی وارد کردن ورودی به ما داده نشد و اسکریپت به سراغ اجرای سایر خطوط رفت. ما حتی نیازی به فشردن دکمه‌ی `Enter` نداشتیم.

دریافت ورودی بدون نشان دادن کاراکترهای دریافتی روی ترمینال

بعضاً ممکن است بخواهیم در اسکریپت خود، اطلاعات حساسی مثل پسورد را از کاربر دریافت کنیم. در چنین حالتی، ما نمی‌خواهیم مواردی که کاربر در پرامپت وارد می‌کند روی ترمینال نمایش داده شود. برای این کار، ما از آپشن `-s` دستور `read` استفاده می‌کنیم.



استفاده از این آپشن، باعث می‌شود که ورودی کاربر روی صفحه‌ی مانیتور نشان داده نشود. البته اگر بخواهیم خیلی دقیق بگوییم، ورودی کاربر روی صفحه نشان داده می‌شود، اما رنگ کاراکترها، هم رنگ صفحه‌ی ترمینال می‌شود و در نتیجه قابل مشاهده نخواهد بود. بیایید این امر را در یک اسکریپت مشاهده کنیم:

```
[root@localhost ~]# cat script12.sh
#!/bin/bash
# Hiding the input from users

read -s -p "Please enter your password: " pass
echo
echo "lol, you use $pass as your password? Weak..."
[root@localhost ~]# chmod u+x script12.sh
[root@localhost ~]# ./script12.sh
Please enter your password:
lol, you use letmein as your password? Weak...
```

همانطور که می‌بینید، ورودی تایپ شده توسط ما روی صفحه‌ی نمایش نشان داده نشد، اما مقدار آن به متغیر pass اختصاص داده شد و ما توانستیم مثل قبل، از آن استفاده کنیم.

Exit Statusها

زمانی که اجرای یک شیل اسکریپت به پایان می‌رسد، یک Exit Status به شیلی که آن را استارت زده می‌فرستد. این Exit Status به ما می‌گوید که آیا شیل اسکریپت به صورت صحیح اجرا شده یا نه. لینوکس یک متغیر محیطی ویژه به نام \$? به ما می‌دهد. این متغیر، Exit Status آخرین دستور اجرا شده در سیستم را درون خود نگهداری می‌کند. برای مشاهده‌ی Exit Status یک دستور، باید بلافاصله پس از اجرای دستور، مقدار ذخیره شده در متغیر \$? را مشاهده کنیم. برای مثال:

```
[root@localhost ~]# ps
  PID TTY          TIME CMD
 7823 pts/0    00:00:00 bash
 9251 pts/0    00:00:00 ps
[root@localhost ~]# echo $?
0
```

همانطور که می‌بینید، ما بلافاصله پس از اجرای دستور ps، مقدار ذخیره شده در متغیر \$? را بررسی کردیم. مقدار ۰ برای Exit Status، به معنای اجرا شدن صحیح و کامل یک دستور می‌باشد. اگر دستور در حین اجرا به یک خطایی بخورد، Exit Status، دارای یک عدد مثبت خواهد شد. ما می‌توانیم با استفاده از دستور exit، مقدار Exit Status شیل اسکریپت‌های خود را تغییر دهیم. بیایید این امر را در یک اسکریپت امتحان کنیم:

```
[root@localhost ~]# cat script13.sh
#!/bin/bash
# Testing the exit status concept

echo "Aloha!!"
exit 69
[root@localhost ~]# chmod u+x script13.sh
[root@localhost ~]# ./script13.sh
Aloha!!
[root@localhost ~]# echo $?
69
```

همانطور که می‌بینید، ما در اسکریپت خود از دستور `exit 69` استفاده کردیم و این باعث شد که `Exit Status` اسکریپت ما برابر با ۶۹ شود. توجه داشته باشید که ما هیچ وقت نباید `Exit Status` اجرای صحیح برنامه را تغییر بدهیم و `Exit Status` اجرای صحیح بهتر است همیشه برابر با ۰ باشد؛ زمانی که اسکریپت‌های پیشرفته‌تر بنویسیم، می‌توانیم برای خطاهای متفاوت در اسکریپت، `Exit Status`‌های مجزا ایجاد کنیم. این امر ما را در دیباگ کردن مشکلات موجود در اسکریپت، یاری می‌دهد.

نوشتن برنامه‌های اسکریپتی

تا به اینجا یاد گرفتیم که چگونه می‌توانیم دستورهای معمولی موجود در سیستم را داخل یک شِیل اسکریپت قرار داده و بدین شکل، مجموعه‌ای از دستورها را با استفاده از یک اسکریپت، اجرا کنیم. اما شِیل اسکریپت‌ها قابلیت‌های پیشرفته‌تری نیز دارند. شِیل `bash` دستورهای بسیار شبیه به دستورهای یک زبان برنامه‌نویسی در اختیار ما قرار می‌دهد که باعث می‌شود بتوانیم اسکریپت‌های خود را تبدیل به یک برنامه‌ی کامل کنیم. از قابلیت‌های پیشرفته‌ی شِیل، می‌توان به قابلیت ذخیره‌ی خروجی یک دستور، قابلیت انجام عملیات ریاضی، قابلیت چک کردن مقدار یک متغیر و وضعیت یک فایل، حلقه‌ها و... اشاره کرد. در این بخش، به آشنایی با برخی از قابلیت‌های پیشرفته‌ی شِیل که بسیار شبیه به قابلیت‌های یک زبان برنامه‌نویسی می‌باشند می‌پردازیم.

جایگزینی یک دستور

می‌توان گفت که قابلیت ذخیره و پردازش داده، یکی از به‌دردبخورترین قابلیت‌های شِیل اسکریپت‌ها می‌باشد. ما تا به اینجا در مورد چگونگی ذخیره‌ی خروجی یک دستور و همچنین چگونگی پایپ کردن خروجی یک دستور در ورودی دستور دیگر صحبت کرده‌ایم. اما روش بهتری برای ذخیره‌ی و استفاده از خروجی یک دستور وجود دارد.

جایگزینی دستور یا `Command Substitution` به ما امکان می‌دهد که خروجی یک دستور را داخل یک متغیر محلی درون شِیل اسکریپت بریزیم. پس از این که خروجی دستور داخل یک متغیر ریخته شد، می‌توانیم از ابزارهای `grep`، `sort` و... برای پردازش و آنالیز آن استفاده کنیم.

برای این که خروجی یک دستور را داخل یک متغیر بریزیم، می‌توانیم به یکی از دو روش زیر عمل کنیم:

- قرار دادن علامت بک‌تیک (```) دور دستور، یعنی مثلاً ``date``.
- قرار دادن دستور بین علامت `$()`، یعنی مثلاً `$(date)`.

هر دو روش دقیقاً مانند هم عمل می‌کنند و خروجی دستور را درون یک متغیر می‌ریزند. از آنجایی که بسیاری علامت بک‌تیک را با علامت آپاستروف اشتباه می‌گیرند، معمولاً از روش دوم برای جایگزینی دستور استفاده می‌کنیم.

بیایید عملکرد هر دو روش را تست کنیم:

```
[root@localhost ~]# today=`date`
[root@localhost ~]# echo $today
Sat Feb 27 12:50:11 +0330 2021
```

همانطور که می‌بینید، قرار دادن علامت بک‌تیک دور دستور `date` و اختصاص آن به یک متغیر، باعث شد که خروجی این دستور درون متغیر `today` قرار گیرد. لازم است بار دیگر ذکر کنیم که علامت بک‌تیک با علامت آپاستروف (`'`) تفاوت دارد.

در روش دوم:

```
[root@localhost ~]# my_name=$(whoami)
[root@localhost ~]# echo $my_name
root
```

همانطور که می‌بینید، قرار دادن دستور whoami داخل پرانتز علامت‌های $()$ باعث شد که خروجی این دستور داخل متغیر my_name قرار گیرد.

انجام عملیات ریاضی

خیلی از اوقات نیاز است که کارهایی فراتر از پردازش رشته‌های متنی انجام دهیم. متأسفانه شِیل bash از نظر انجام عملیات ریاضی، زیاد قدرتمند نیست و شِیل‌های جدیدتر مثل zsh، قابلیت‌های بهتری برای انجام عملیات ریاضی دارند. با این حال، باید چگونگی انجام عملیات ساده‌ی ریاضی در bash را یاد بگیریم. برای این که بتوانیم در اسکریپت خود عملیات ریاضی انجام دهیم، باید عملیات مورد نظر را داخل پرانتز علامت‌های $(())$ قرار دهیم. برای مثال:

```
[root@localhost ~]# result=$(( 2 * 2 ))
[root@localhost ~]# echo $result
4
```

محدودیت بزرگ انجام عملیات ریاضی از طریق این روش، عدم امکان انجام عملیات روی اعداد اعشاری می‌باشد. ما باید این امر را هنگام انجام عملیاتی نظیر تقسیم و... در نظر بگیریم، چرا که شِیل فقط می‌تواند اعداد صحیح را نشان داده و روی آن عملیات انجام دهد.

اگر بخواهیم با اعداد اعشاری کار کنیم، باید به سراغ برنامه‌های جانبی برویم. برنامه‌ی bc، یک ماشین‌حساب لینوکسی است که می‌تواند با اعداد اعشاری کار کند. برای مثال:

```
[root@localhost ~]# bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4 * 0.5
2.0
3.14 * (2 + 5)
21.98
quit
```

همانطور که می‌بینید، استفاده از bc کار دشواری نیست، اما هنگام انجام عملیات تقسیم، برنامه‌ی bc کمی ما را اذیت می‌کند. برای مثال:

```
[root@localhost ~]# bc -q
1 / 2
0
quit
```

همانطور که می‌بینید، این برنامه نتیجه تقسیم عدد یک بر دو را صفر نشان می‌دهد. دلیل این امر، این است که ما باید به bc بگوییم که نتیجه‌ی عملیات را با چند رقم اعشار در خروجی به ما نشان دهد. ما می‌توانیم این امر را با استفاده از یک متغیر داخلی به نام scale تنظیم کنیم. برای مثال:

```
[root@localhost ~]# bc -q
scale=2
1 / 2
.50
quit
```

همانطور که می‌بینید با ارائه‌ی عدد ۲ به متغیر `scale`، به برنامه‌ی `bc` می‌گوییم که نتایج را با دو رقم اعشار به ما نشان دهد.

نکته: اجرای `bc` با آپشن `-q`، باعث می‌شود که پیغام خوش‌آمدگویی `bc` (در مورد لایسنس برنامه و...) به ما نشان داده نشود.

استفاده از برنامه‌ی `bc` درون اسکریپت، کمی پیچیده می‌باشد. ما باید از جایگزینی دستور استفاده کنیم و بدین طریق، خروجی دستور `bc` را درون یک متغیر بریزیم. اما فرمتی که باید برای این کار استفاده کنیم، به صورت زیر می‌باشد:

```
result=$(echo "options; expression" | bc)
```

به طوری که `options`، مشخص کننده‌ی متغیرهای مربوط به برنامه‌ی `bc`، نظیر متغیر `scale` می‌باشد و `expression`، مشخص کننده‌ی عملیات ریاضی می‌باشد که می‌خواهیم `bc` آن را انجام دهد. برای مثال:

```
[root@localhost ~]# result=$(echo "scale=2; 1 / 2" | bc)
[root@localhost ~]# echo $result
.50
```

همانطور که می‌بینید، با این که چنین کاری نمای بسیاری عجیبی دارد، اما نتیجه‌ی مورد نظر ما را می‌دهد و برای انجام محاسبات کوچک در پروژه‌های کوچک، جواب می‌دهد. اگر نیاز به انجام محاسبات بزرگ‌تر داشته باشیم، باید به سراغ یک شی دیگر، به نام `Zsh` برویم. این شی قابلیت انجام محاسبات پیشرفته را دارد و این کار را بسیار ساده می‌کند. ما در اینجا به توضیح چگونگی اسکریپت‌نویسی برای `Zsh` نمی‌پردازیم.

گزاره‌های منطقی

تا به اینجا کلیه‌ی اسکریپت‌هایی که نوشتیم به صورت خطی عمل کرده‌اند، یعنی یک دستور پس از دستور دیگر اجرا شده تا به انتهای اسکریپت برسیم. اما همه‌ی برنامه‌ها خطی نیستند و خیلی از اوقات می‌خواهیم برنامه‌ی ما در شرایط متفاوت، عملیات متفاوتی انجام دهد. مثلاً ممکن است بخواهیم اسکریپت در صورت وجود یک فایل، یک کاری انجام دهد و در صورت عدم وجود، یک کار دیگر انجام دهد. ما می‌توانیم این کار را با استفاده از گزاره‌های منطقی انجام دهیم.

گزاره‌های منطقی به ما اجازه می‌دهند که بتوانیم یک شرط خاص را تست کنیم و با توجه به نتیجه‌ی آن، دستورهای متفاوتی را اجرا کنیم. نتیجه‌ی تست شرط‌ها، `True` یا `False` می‌باشد. روش‌های متفاوتی برای پیاده‌سازی گزاره‌های منطقی وجود دارد که در این بخش با آنها آشنا می‌شویم.

گزاره‌ی if

ساده‌ترین گزاره‌ی منطقی، گزاره‌ی شرطی `if` می‌باشد. فرمت گزاره‌ی شرطی `if` به صورت زیر می‌باشد:

```
if [ condition ]
then
    commands
fi
```

کروشه‌هایی که دور `condition` در دستور `if` قرار گرفته‌اند، به عنوان مخفی برای دستور `test` عمل می‌کنند. دستور `test`، یک شرط را سنجیده و در صورت برقرار بودن شرط، مقدار منطقی بول `True` را در خروجی به ما می‌دهد و در صورت برقرار نبودن شرط، مقدار منطقی بول `False` را در خروجی به ما می‌دهد.



در دستور `if`، اگر شرط برقرار باشد، دستورهای (`commands`) نوشته شده پس از `then` اجرا می‌شوند. اگر شرط `condition` برقرار نباشد، شیل از روی دستورهای (`commands`) نوشته شده پس از `then` عبور کرده و آنها را اجرا نمی‌کند.

برای نوشتن `condition` درون براکت، فرمت‌های بسیار زیادی وجود دارد. در شیل، مقادیر عددی، رشته‌ها و حتی فایل‌ها و دایرکتوری‌ها، فرمت‌های تست شرط مخصوص به خود را دارند. در جدول ۱، فرمت‌های ممکن برای تست شروط متفاوت را مشاهده می‌کنیم.

جدول ۱ - فرمت‌های قابل استفاده برای تست شروط متفاوت

عملکرد	نوع	تست
مساوی بودن عدد $n1$ با عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -eq n2</code>
بزرگ‌تر یا مساوی بودن عدد $n1$ از عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -ge n2</code>
بزرگتر بودن عدد $n1$ از عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -gt n2</code>
کوچکتر یا مساوی بودن عدد $n1$ از عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -le n2</code>
کوچکتر بودن عدد $n1$ از عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -lt n2</code>
مساوی نبودن عدد $n1$ با عدد $n2$ را بررسی می‌کند.	عدد	<code>n1 -ne n2</code>
مساوی بودن رشته‌ی $str1$ با رشته‌ی $str2$ را بررسی می‌کند.	رشته	<code>str1 = str2</code>
مساوی نبودن رشته‌ی $str1$ با رشته‌ی $str2$ را بررسی می‌کند.	رشته	<code>str1 != str2</code>
کوچکتر بودن رشته‌ی $str1$ از رشته‌ی $str2$ را بررسی می‌کند.	رشته	<code>str1 < str2</code>
بزرگتر بودن رشته‌ی $str1$ از رشته‌ی $str2$ را بررسی می‌کند.	رشته	<code>str1 > str2</code>
بزرگتر بودن طول رشته‌ی $str1$ از صفر را بررسی می‌کند.	رشته	<code>-n str1</code>
برابر بودن طول رشته‌ی $str1$ با صفر را بررسی می‌کند.	رشته	<code>-z str1</code>
وجود $file$ و از نوع دایرکتوری بودن آن را بررسی می‌کند.	فایل	<code>-d file</code>
وجود $file$ را بررسی می‌کند.	فایل	<code>-e file</code>
وجود $file$ و از نوع فایل بودن آن را بررسی می‌کند.	فایل	<code>-f file</code>
وجود $file$ و عدم خالی بودن آن را بررسی می‌کند.	فایل	<code>-s file</code>
وجود $file$ و قابل خواندن بودن آن را بررسی می‌کند.	فایل	<code>-r file</code>
وجود $file$ و قابل نوشتن بودن آن را بررسی می‌کند.	فایل	<code>-w file</code>
وجود $file$ و قابل اجرا بودن آن را بررسی می‌کند.	فایل	<code>-x file</code>
وجود $file$ و مالکیت آن توسط کاربر کنونی را بررسی می‌کند.	فایل	<code>-0 file</code>
وجود فایل و همچنین عضویت $file$ در گروه پیش‌فرض کاربر کنونی را بررسی می‌کند.	فایل	<code>-G file</code>
جدیدترین بودن $file1$ نسبت به $file2$ را بررسی می‌کند.	فایل	<code>file1 -nt file2</code>
قدیمی‌تر بودن $file1$ نسبت به $file2$ را بررسی می‌کند.	فایل	<code>file1 -ot file2</code>

بیا باید از دستور `if` درون یک اسکریپت استفاده کنیم. فرض کنید می‌خواهیم اسکریپتی بنویسیم که به ازای دریافت دو عدد به عنوان آرگمان، به ما بگوید که آیا این دو عدد با هم برابر هستند یا عدد اول بزرگتر یا کوچکتر از عدد دومی است. پس:

```
[root@localhost ~]# cat script14.sh
#!/bin/bash
# Testing the if statement

if [ $1 -eq $2 ]
then
    echo "First number is equal to second number."
fi

if [ $1 -gt $2 ]
then
    echo "First number is greater than second number."
fi

if [ $1 -lt $2 ]
then
    echo "First number is smaller than second number."
fi
```

این اسکریپت نیاز به توضیح خاصی ندارد و با توجه به توضیحاتی که دادیم، خوتان باید بفهمید که این اسکریپت چگونه کار می‌کند. بیا باید این اسکریپت را اجرا کنیم و عملکرد آن را ببینیم:

```
[root@localhost ~]# chmod u+x script14.sh
[root@localhost ~]# ./script14.sh 69 420
First number is smaller than second number
[root@localhost ~]# ./script14.sh 420 420
First number is equal to second number
[root@localhost ~]# ./script14.sh 420 69
First number is greater than second number
```

همانطور که می‌بینید، اسکریپت به درستی برابر بودن، کوچکتر بودن یا بزرگتر بودن اعدادی که به عنوان آرگمان به اسکریپت می‌دهیم را تشخیص می‌دهد.

نکته: ما می‌توانیم در بررسی شروط `if`، از عبارتهای بولی `AND` و `OR` نیز استفاده کنیم. برای `AND` از `&&` و برای `OR` از `||` استفاده می‌کنیم. برای مثال:

```
[root@localhost ~]# if [ 1 > 2 ] && [ 1 > 0 ]
> then
> echo "wow"
> fi
wow
```

در اینجا اگر ۱ از ۲ بزرگتر باشد و همچنین ۱ از صفر بزرگتر باشد، نتیجه‌ی تست ما `True` خواهد شد و در نتیجه، پیغام `wow` روی صفحه نشان داده می‌شود.

گزاره‌ی case

اگر به اسکریپت قبلی نگاه کنید، می‌بینید که ما ۳ عبارت `if` نوشتیم تا مقدار اختصاص یافته به متغیر `$1` را بررسی کنیم. حال فرض کنید می‌خواستیم مقدار اختصاص یافته به متغیر `$1` را برای ۱۰ مقدار متفاوت بررسی کنیم و به ازای هر مقدار، یک دستور متفاوت اجرا کنیم. در چنین حالتی باید ۱۰ دستور `if` می‌نوشتیم.



پرواضح است که ۱۰ بار نوشتن دستور `if` اصلا منطقی و زیبا نیست. به همین دلیل، شیل گزاره‌ی `case` را در اختیار ما قرار می‌دهد. بدین ترتیب، به جای نوشتن دستور `if` برای هر مقدار یا وضعیت، می‌توانیم از یک گزاره‌ی `case` استفاده کنیم.

گزاره‌ی `case` به ما اجازه می‌دهد که مقادیر متفاوت اختصاص داده شده به یک متغیر را بررسی کرده و به ازای هر مقدار، یک دستور را اجرا کنیم. به طور کلی، گزاره‌ی `case` فرمتی شبیه زیر دارد:

```
case variable in:
pattern1) commands1;;
pattern2 | pattern3) commands2;;
*) default commands;;
esac
```

گزاره‌ی `case`، مقدار اختصاص یافته به متغیر `variable` را بررسی می‌کند و اگر مقدار برابر با `pattern1` باشد، دستورات `commands1` را اجرا می‌کند. ما می‌توانیم بیش از یک `pattern` مشخص کنیم و همچنین می‌توانیم در یک خط، بیش از چندین `pattern` قرار داده، به شرطی که هر `pattern` را با علامت `|` از `pattern` قبلی جدا کنیم. این کار باعث می‌شود که منطق `OR` بین دو `pattern` مشخص شده در یک خط پیاده‌سازی شود.

هنگام استفاده از گزاره‌ی `case`، ما باید حالتی که مقدار اختصاص یافته به `variable` برابر با هیچ کدام از `pattern` ها نیست را نیز در نظر بگیریم. ما این حالت را با استفاده از علامت `*` مشخص کرده و دستورهای که می‌خواهیم در این حالت اجرا شوند را (`default commands`) وارد می‌کنیم.

حال بیایید از گزاره‌ی `case` در یک اسکریپت استفاده کنیم. فرض کنید می‌خواهیم نام یک فرد را در ورودی دریافت کنیم و با توجه به نام فرد، پیام متفاوتی روی صفحه نشان دهیم:

```
[root@localhost ~]# cat script15.sh
#!/bin/bash
# Testing the case statement

read -p "Enter your name: " name

case $name in
Behnam)
    echo "Welcome, $name!"
    echo "You're awesome, man!";;
Ali | Abbas)
    echo "Hi, $name!"
    echo "You're late...";;
Kaveh)
    echo "Ok, $name, you're really late!";;
*)
    echo "$name, you are NOT welcome here.";;
esac
```

همانطور که می‌بینید، برای تست مقدار ذخیره شده در متغیر `name`، نیازی به قرار دادن رشته‌ها بین دو علامت `"` نیست؛ یعنی مثلا ما دور کلمه‌ی `Behnam`، علامت `"` را قرار ندادیم. اگر به دومین الگوی ذکر شده در `case` نگاه کنید، می‌بینید که ما از دو الگو در یک خط استفاده کرده‌ایم و آنها را با علامت `|` از هم جدا کرده‌ایم. این به معنای این می‌باشد که اگر مقدار ذخیره شده در متغیر `name` برابر با `Ali` یا برابر با `Abbas` بود، باید دستورهای ذکر شده برای این `case` اجرا شوند.

حال بیایید اسکریپت را اجرا کرده و عملکرد آن را بررسی کنیم:

```
[root@localhost ~]# chmod u+x script15.sh
[root@localhost ~]# ./script15.sh
Enter your name: Behnam
Welcome, Behnam!
You're awesome, man!
[root@localhost ~]# ./script15.sh
Enter your name: Ali
Hi, Ali!
You're late...
[root@localhost ~]# ./script15.sh
Enter your name: Abbas
Hi, Abbas!
You're late...
[root@localhost ~]# ./script15.sh
Enter your name: Kaveh
Ok, Kaveh, you're really late!
[root@localhost ~]# ./script15.sh
Enter your name: Joe Mama
Joe Mama, you are NOT welcome here.
```

همانطور که می بینید، خروجی دریافتی از اسکریپت با توجه به مقدار وارد شده در ورودی اسکریپت، دچار تغییر می شود.

حلقه ها

هنگام نوشتن اسکریپت، خیلی از اوقات لازم است که یک دستور را چندین و چند بار تکرار کنیم. مثلاً ممکن است بخواهیم یک دستور را روی تک تک فایل های موجود در دایرکتوری اجرا کنیم. ما می توانیم این کار را با استفاده از حلقه ها انجام دهیم. `bash` چندین دستور ساده ی حلقه در اختیار ما قرار می دهد که در این بخش با آنها آشنا می شویم.

حلقه ی `for`

حلقه ی `for`، تک تک اعضای موجود در یک سری (یا مجموعه ای از اطلاعات) را پیمایش و به ما اجازه می دهد که عملیاتی را روی هر عضو آن سری انجام دهیم. یک سری، می تواند مجموعه ای از اعداد، مجموعه ای از فایل ها و دایرکتوری ها و یا حتی خطوط موجود درون یک فایل باشد. فرمت نوشتن حلقه ی `for` به صورت زیر می باشد:

```
for variable in series ; do
    commands
done
```

در اینجا، `variable` به عنوان یک متغیر جانگهدار عمل کرده و در هر پیمایش، مقدار موجود در یک عضو موجود در `series` را درون خود قرار می دهد. دستورهای ذکر شده در بدنه ی حلقه (`commands`)، می توانند از `variable` دقیقاً مانند هر متغیر دیگر موجود در اسکریپت، استفاده کنند. شاید درک عملکرد `for` کمی برایتان دشوار باشد. بیایید با یک مثال عملکرد این دستور را بهتر درک کنیم. فرض کنید می خواهیم یک اسکریپت بنویسیم که اعداد ۱ تا ۵ را روی ترمینال به ما نشان می دهد. برای این کار:

```
[root@localhost ~]# cat script16.sh
#!/bin/bash
```



```
# Getting to know the for loop
```

```
for num in $(seq 1 5); do
    echo $num
done
```

بهتر است ابتدا با دستور seq آشنا شویم. دستور seq با دریافت یک نقطه‌ی شروع و یک نقطه‌ی پایان به عنوان آرگمان، کلمه‌ی اعداد بین آن دو نقطه را در خروجی به ما تحویل می‌دهد. برای مثال، `seq 1 5` کلمه‌ی اعداد موجود بین ۱ تا ۵، اعم از خود ۱ و خود ۵ را در خروجی به ما نشان می‌دهد.

حال بیایید در مورد عملکرد اسکریپت صحبت کنیم. دستور `for` در این اسکریپت، در هر مرحله عبور از دستور `seq`، یکی از مقادیر موجود در خروجی این دستور را به متغیر `num` می‌دهد. یعنی در اولین عبور، متغیر `num` مقدار ۱ (اولین خروجی دستور `seq`) را خواهد داشت. حال ما می‌توانیم با این مقدار اختصاص یافته به متغیر `num`، هر کاری را انجام دهیم و هر دستوری را اجرا کنیم. در دومین عبور، متغیر `num` مقدار ۲ را خواهد داشت و به همین ترتیب تا متغیر `num`، عدد ۵ را داشته باشد. پس از رسیدن به عدد ۵، خروجی `seq` به پایان می‌رسد و دستور `for` نیز به دلیل عدم وجود مقداری دیگر در خروجی `seq`، به کار خود پایان می‌دهد.

حال بیایید این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script16.sh
[root@localhost ~]# ./script16.sh
1
2
3
4
5
```

همانطور که می‌بینید، اسکریپت دقیقاً طبق انتظار ما عمل کرد.

حال بیایید یک اسکریپت دیگر بنویسیم. فرض کنید می‌خواهیم یک اسکریپت بنویسیم که به کلمه‌ی محتویات یک دایرکتوری نگاه کرده و نام آنها را در خروجی به ما نشان دهد:

```
[root@localhost ~]# cat script17.sh
#!/bin/bash
# Testing for loops

for file in $(ls); do
    echo "There is a file named $file here."
done
```

همانطور که می‌بینید، در اینجا `file` به عنوان متغیری عمل می‌کند که در هر پیمایش، یکی از مقادیر موجود در خروجی دستور `ls` را به خود اختصاص می‌دهد. ما برای هر مقدار اختصاص یافته به متغیر `file`، پیامی را در خروجی `echo` خواهیم کرد. بیایید اسکریپت را اجرا کرده و عملکرد آن را ببینیم:

```
[root@localhost ~]# chmod u+x script17.sh
[root@localhost ~]# ./script17.sh
There is a file named script10.sh here.
There is a file named script11.sh here.
There is a file named script12.sh here.
There is a file named script13.sh here.
There is a file named script14.sh here.
There is a file named script15.sh here.
There is a file named script16.sh here.
There is a file named script1.sh here.
```

```

There is a file named script2.sh here.
There is a file named script3.sh here.
There is a file named script4.sh here.
There is a file named script5.sh here.
There is a file named script6.sh here.
There is a file named script7.sh here.
There is a file named script8.sh here.
There is a file named script9.sh here.

```

همانطور که می‌بینید، این اسکریپت تک‌تک موارد موجود در خروجی دستور `ls` را به متغیر `file` اختصاص داد و ما هر دفعه، مقدار اختصاص یافته (در اینجا نام فایل) به متغیر `file` را در خروجی نشان دادیم. حال بیایید یک اسکریپت کمی پیشرفته‌تر بنویسیم. فرض کنید می‌خواهیم اسکریپتی بنویسیم که به کلیه محتویات یک دایرکتوری نگاه کرده و به ما می‌گوید که کدام از آنها فایل و کدام از آنها دایرکتوری هستند. پس:

```
[root@localhost ~]# cat script18.sh
```

```
#!/bin/bash
```

```
# Understanding for loops
```

```

for entity in $( ls ); do
    if [ -f $entity ]
    then
        echo "$entity is a file!"
    fi

    if [ -d $entity ]
    then
        echo "$entity is a directory!"
    fi
done

```

همانطور که می‌بینید، ما دقیقاً مثل اسکریپت قبل، تک‌تک موارد موجود در خروجی دستور `ls` را به متغیر `entity` اختصاص می‌دهیم و سپس با استفاده از دستور `if` و تست‌هایی که قبلاً یاد گرفتیم، فایل یا دایرکتوری بودن `entity` را بررسی می‌کنیم. حال بیایید این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script18.sh
```

```
[root@localhost ~]# ./script18.sh
```

```

ladmad is a directory!
landmine is a directory!
madlad is a directory!
script10.sh is a file!
script11.sh is a file!
script12.sh is a file!
script13.sh is a file!
script14.sh is a file!
script15.sh is a file!
script16.sh is a file!
script17.sh is a file!
script18.sh is a file!
script1.sh is a file!
script2.sh is a file!
script3.sh is a file!
script4.sh is a file!
script5.sh is a file!
script6.sh is a file!
script7.sh is a file!

```



```
script8.sh is a file!
script9.sh is a file!
```

همانطور که می بینید، اسکریپت دقیقاً مورد انتظار ما عمل می کند.

حلقه‌ی while

یکی دیگر از حلقه‌های بسیار کاربردی، حلقه‌ی while می باشد. بیایید قبل از توضیح آن، با فرمت نوشتار حلقه‌ی while آشنا شویم:

```
while [ condition ] ; do
    commands
done
```

حلقه‌ی while، تا زمانی که شرط *condition* برقرار باشد (True باشد)، تکرار خواهد شد و به محض این که شرط *condition* دیگر برقرار نباشد (False باشد)، حلقه دیگر تکرار نخواهد شد. *condition* در حلقه‌ی while دقیقاً از فرمت شرط‌ها در دستور if استفاده می کند، پس ما می توانیم شرط‌هایی که روی اعداد، رشته‌ها و فایل‌ها کار می کنند را بررسی کنیم.

بیایید یک اسکریپت بنویسیم که فاکتوریل عددی که به عنوان آرگمان به اسکریپت داده می شود را محاسبه کند. اگر فراموش کرده اید، فاکتوریل عدد n که آن را با $n!$ نشان می دهیم، به صورت زیر محاسبه می شود:

$$n! = n \times (n - 1) \times (n - 2) \dots \times 1$$

برای مثال:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

نوشتن این اسکریپت نباید زیاد دشوار باشد. پس:

```
[root@localhost ~]# cat script19.sh
#!/bin/bash
# Testing while loops

number=$1
factorial=1

while [ $number -gt 0 ] ; do
    factorial=$(( $factorial * $number ))
    number=$(( $number - 1 ))
done
echo "Factorial of $1 is: $factorial"
```

این اسکریپت، اولین آرگمانی که به آن داده می شود را در متغیر *number* ذخیره می کند و همچنین یک متغیر به نام *factorial* ایجاد کرده و مقدار ۱ را به آن اختصاص می دهد. سپس اسکریپت ما وارد یک حلقه‌ی while می شود. این حلقه تا زمانی که مقدار اختصاص یافته به متغیر *number* بزرگتر از صفر باشد، تکرار خواهد شد. در بدنه‌ی حلقه‌ی while، حاصل ضرب مقدار متغیر *factorial* در مقدار متغیر *number* به دست آمده و در متغیر *factorial* ذخیره می شود و سپس از متغیر *number* یک واحد کسر می شود. همانطور که می بینید، این حلقه در هر بار عبور، یک واحد از مقدار اختصاص یافته به متغیر *number* کم می کند. این حلقه انقدر این کار را تکرار می کند تا مقدار اختصاص یافته به *number* برابر با صفر شود و در نتیجه، شرط حلقه‌ی while برابر با False شده و حلقه برسد. هنگامی که این اتفاق بیافتد، متغیر

factorial، مقدار فاکتوریل عددی که کاربر به عنوان آرگمان به اسکریپت داده را درون خود خواهد داشت، پس ما آن را در خروجی echo می‌کنیم.

بیا این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script19.sh
[root@localhost ~]# ./script19.sh 5
Factorial of 5 is: 120
[root@localhost ~]# ./script19.sh 3
Factorial of 3 is: 6
```

همانطور که می‌بینید، این اسکریپت فاکتوریل هر عددی که بخواهیم را به ما می‌دهد!

فانکشن‌ها

با پیچیده‌تر شدن شیل اسکریپت‌ها، بعضاً لازم است چندین بار از قطعه‌کدی که یک عملیات خاص را انجام می‌دهد استفاده کنیم. این عملیات می‌توانند کارهای بسیار ساده‌ای مثل نمایش پیام روی ترمینال یا دریافت اطلاعاتی از کاربر باشند و یا می‌توانند کارهای پیچیده‌ای مثل انجام یک سری محاسبات باشند که لازم است آن را چندین و چند بار در اسکریپت خود تکرار کنیم.

اگر بخواهیم برای هر بار انجام این عملیات، کل کدی که آن عملیات را انجام می‌دهد را از اول بنویسیم یا آن را در اسکریپت کپی کنیم، نه تنها کارمان بسیار سخت می‌شود، بلکه خواندن اسکریپت، دیباگ کردن و... بسیار دشوار خواهد شد.

فانکشن‌ها، به ما امکان می‌دهند که یک نام به قطعه‌ای از کد اختصاص دهیم و سپس از آن قطعه کد در هر کجای اسکریپت خود استفاده کنیم. در این حالت، هر بار که بخواهیم از آن قطعه کد استفاده کنیم، کافی است نام اختصاص یافته به آن را بنویسیم تا آن قطعه کد اجرا شود (به این کار، صدا زدن فانکشن می‌گوییم). در این بخش می‌خواهیم در مورد چگونگی ایجاد و استفاده از فانکشن‌ها صحبت کنیم.

تعریف فانکشن در شیل اسکریپت، به دو روش امکان‌پذیر است. در روش اول، ما از واژه‌ی function استفاده کرده، سپس نام مورد نظر برای فانکشن را نوشته و در نهایت قطعه کد مورد نظر خود را داخل یک آکولاد می‌نویسیم. یعنی:

```
function name {
    commands
}
```

به طوری که *name* نشان دهنده‌ی نام منحصر به فردی می‌باشد که به این فانکشن اختصاص می‌دهیم. هر فانکشن در یک اسکریپت، باید یک نام منحصر به فرد داشته باشد. *commands* نشان دهنده‌ی یک یا چندین دستور می‌باشد که یک کار خاصی را انجام می‌دهند. زمانی که یک فانکشن را صدا می‌زنیم، شیل تک‌تک دستورهای نوشته شده را طبق ترتیب نوشته شده، اجرا می‌کند.

روش دوم تعریف فانکشن، به صورت زیر می‌باشد:

```
name() {
    commands
}
```

پراکنش‌های موجود پس از *name*، به *bash* می‌گویند که ما داریم یک فانکشن تعریف می‌کنیم. فانکشن‌هایی که به این روش تعریف می‌شوند نیز باید نام منحصر به فرد داشته باشند.

برای استفاده از یک فانکشن در اسکریپت خود، کافی است نام فانکشن را دقیقاً مانند یک دستور داخل اسکریپت خود بنویسیم. بیایید نمونه‌ای از تعریف و استفاده از فانکشن را مشاهده کنیم:

```
[root@localhost ~]# cat script20.sh
#!/bin/bash
# Trying out functions

function func1 {
    echo "Hello from func1!"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( count + 1))
done

echo "The while loop just finished"
func1
echo "And we have reached the end of our script!"
```

بیایید این اسکریپت را اجرا کنیم:

```
[root@localhost ~]# chmod u+x script20.sh
[root@localhost ~]# ./script20.sh
Hello from func1!
Hello from func1!
Hello from func1!
Hello from func1!
Hello from func1!
The while loop just finished
Hello from func1!
And we have reached the end of our script!
```

همانطور که می‌بینید اسکریپت دقیقاً طبق انتظار ما اجرا می‌شود. درک عملکرد کد بالا نباید زیاد برایتان دشوار باشد. همانطور که می‌بینید، هر بار که از نام فانکشن `func1` در اسکریپت استفاده کردیم، دستورهای موجود در بدنه‌ی این فانکشن اجرا شدند. به عبارت دیگر، `bash` هر بار که نام یک فانکشن را می‌بینید، به سراغ آن فانکشن رفته، دستورهای موجود در بدنه‌ی آن فانکشن را اجرا می‌کند و سپس به سراغ اجرای دستورهای بعدی نوشته شده در اسکریپت می‌رود.

فانکشن‌ها می‌توانند برای خود یک `Exit Status` داشته باشند. در فانکشن‌ها، `Exit Status`‌ها با دستور `return` مشخص می‌شوند. دستور `return` به ما امکان می‌دهد که یک عدد صحیح را به عنوان `Exit Status` فانکشن تعریف کنیم. بیایید از `return` در یک فانکشن استفاده کنیم:

```
[root@localhost ~]# cat script21.sh
#!/bin/bash
# Trying out the return command

double(){
    read -p "Enter a number: " num
    echo "Lets double $num up!"
    return $(( num * 2 ))
}
double
```



```
echo "The doubled up value is $?!"
```

```
[root@localhost ~]# chmod u+x script21.sh
```

```
[root@localhost ~]# ./script21.sh
```

```
Enter a number: 69
```

```
Lets double 69 up!
```

```
The doubled up value is 138!
```

همانطور که می‌بینید، ما در این اسکریپت یک فانکشن به نام double ایجاد کردیم. این فانکشن از کاربر یک عدد در ورودی دریافت می‌کند و آن را در متغیر num ذخیره می‌کند. سپس عدد ذخیره شده در متغیر num را در ۲ ضرب کرده و نتیجه‌ی آن را به عنوان Exit Status به ما باز می‌گرداند. این بدین معنی است که هر بار این فانکشن را اجرا کنیم، عدد بازگشت داده شده توسط این فانکشن، داخل متغیر محیطی \$? ذخیره خواهد شد.

همانطور که می‌توانیم خروجی یک دستور را درون یک متغیر بریزیم، ما می‌توانیم خروجی فانکشن‌هایی که یک مقداری را return می‌کنند را درون یک متغیر بریزیم. یعنی مثلاً ما می‌توانیم خروجی فانکشن double را داخل یک متغیر بریزیم. به صورت زیر:

```
result=$(double)
```

تمرین: یک اسکریپت بنویسید که اطلاعات مربوط به رمز عبور هر کاربر موجود در سیستم را در خروجی نشان می‌دهد.

برای نوشتن این اسکریپت، به یوزرنیم همه‌ی کاربران موجود در سیستم و اطلاعات رمز عبور آنها نیاز داریم. ما می‌توانیم یوزرنیم کاربران سیستم را از فایل /etc/passwd به دست آوریم و برای به دست آوردن اطلاعات پسورد، می‌توانیم از دستور chage استفاده کنیم. بیایید نگاهی به فایل /etc/passwd بیاندازیم:

```
[root@localhost ~]# cat /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/sbin/nologin
```

```
[...]
```

```
ntp:x:38:38::/etc/ntp:/sbin/nologin
```

```
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

همانطور که می‌بینید، این فایل از یوزرنیم کاربران و همچنین کلی اطلاعات دیگر تشکیل شده است. اما ما فقط به یوزرنیم کاربران نیاز داریم. به عبارت دیگر، ما فقط فیلد اول موجود در هر خط این فایل را نیاز داریم. برای دسترسی به اطلاعات فیلد اول موجود در هر خط، می‌توانیم از دستور cut استفاده کنیم. دستور cut، می‌تواند بخش‌هایی از هر خط موجود در یک فایل را حذف کند. ما از این دستور به صورت زیر استفاده می‌کنیم:

```
[root@localhost ~]# cut -d : -f 1 /etc/passwd
```

```
root
```

```
bin
```

```
[...]
```

```
ntp
```

```
chrony
```

همانطور که می‌بینید، ما ابتدا دستور cut را با یک سری آپشن وارد کرده و سپس نام فایل مورد نظر را وارد کردیم. ما با استفاده آپشن -d، به cut گفتیم که فیلدهای موجود در فایل /etc/passwd با چه علامتی از هم جدا شده‌اند. در اینجا، فیلدها با استفاده از علامت دونقطه (:) از هم جدا شده‌اند، به همین دلیل از این علامت پس از مشخص کردن آپشن -d استفاده کردیم. سپس با استفاده از آپشن -f، به cut گفتیم که کدام فیلد

موجود در فایل، اطلاعات مورد نظر ما را درون خود دارد. با توجه به این که یوزرنیم کاربران در فیلد اول این فایل وجود دارد، ما پس از مشخص کردن این آپشن، از عدد ۱ استفاده کردیم.

حال که به یوزرنیم کلیه کاربران موجود در سیستم دسترسی پیدا کردیم، کافی است این یوزرنیم‌ها را به ورودی دستور chage بدهیم. ما از آپشن -l این دستور استفاده خواهیم کرد، چرا که این آپشن، اطلاعات مربوط به رمز یک کاربر را در خرجی به ما نشان می‌دهد.

حال که تقریباً می‌دانیم به چه صورت باید به اطلاعات مورد نظر دسترسی پیدا کنیم، بیایید اسکریپت را بنویسیم:

```
[root@localhost ~]# cat pass_info.sh
#!/bin/bash
# Show user's password info
user_list=$( cut -d : -f 1 /etc/passwd )

for user in $user_list ; do
    echo "Here's the password info for $user:"
    chage -l $user
    echo "-----"
done
```

با توجه به مواردی که تا به اینجا یاد گرفته‌ایم، درک چگونگی عملکرد این اسکریپت باید بسیار ساده باشد، پس ما به توضیح آن نمی‌پردازیم. کافی است به این اسکریپت مجوز اجرا داده و آن را به عنوان کاربر روت اجرا کنیم تا اطلاعات پسورد همه کاربران به ما نشان داده شود.

اجرای اسکریپت‌ها در پشت صحنه

خیلی از اوقات، اجرای یک اسکریپت زمان بسیار زیادی می‌گیرد و ممکن است ما نخواهیم که ترمینال ما در طول زمان اجرای اسکریپت درگیر باشد و در نتیجه امکان ورود هیچ دستوری توسط ما وجود نداشته باشد. ما در جلسه‌های قبلی در مورد چگونگی اجرای یک برنامه در پشت صحنه صحبت کرده بودیم. در این بخش، باری دیگر در مورد چگونگی اجرای برنامه‌ها (و اسکریپت‌ها) در پشت صحنه صحبت خواهیم کرد.

فرستادن اسکریپت به پشت صحنه

فرض کنید یک اسکریپت ساده به صورت زیر داریم:

```
[root@localhost ~]# cat script22.sh
#!/bin/bash
# Just testing some stuff

for num in $(seq 1 100)
do
    echo "Loop #$num"
    sleep 2
done
[root@localhost ~]# chmod u+x script22.sh
[root@localhost ~]# ./script22.sh
Loop #1
Loop #2
[...]
Loop #100
```

همانطور که می‌بینید، این اسکریپت هر دو ثانیه یک عدد را در خروجی نشان می‌دهد تا این که به عدد ۱۰۰ برسد. اگر دقت کنید می‌بینید که در طول اجرای این اسکریپت، ما نمی‌توانیم هیچ دستور دیگری را در سیستم

اجرا کنیم و عملاً تا زمان به پایان رسیدن این اسکریپت، سیستم ما بلااستفاده می‌باشد. در چنین حالتی، اجرای اسکریپت در پشت صحنه بسیار کاربردی می‌باشد.

اجرای یک شیل اسکریپت در پشت صحنه بسیار ساده می‌باشد؛ کافی است پس از نوشتن نام اسکریپت مورد نظر، یک علامت امپرسند (&) قرار دهیم. یعنی:

```
[root@localhost ~]# ./script22.sh &
[1] 7724
Loop #1
[root@localhost ~]# ls -l
total 88
-rwxr--r--. 1 root root 72 Feb 22 11:45 script1.sh
-rwxr--r--. 1 root root 167 Feb 22 11:52 script2.sh
[...]
-rwxr--r--. 1 root root 99 Mar 3 10:15 script22.sh
[root@localhost ~]# Loop #2
```

همانطور که می‌بینید، با قرار دادن علامت امپرسند، bash اسکریپت را از شیل جدا کرده و آن را به عنوان یک پراسس در پشت صحنه اجرا می‌کند. اگر دقت کنید، می‌بینید که اولین چیزی که شیل پس از اجرای اسکریپت به ما نشان داد، عبارت زیر بود:

```
[1] 7724
```

عدد موجود در کروهه، Job Number نام دارد که شماره‌ای می‌باشد که شیل به پراسس‌های موجود در پشت صحنه اختصاص می‌دهد. شیل به هر پراسس اجرا شده در یک ترمینال، یک شماره جاب منحصر به فرد اختصاص می‌دهد. عدد بیرون از کروهه، شماره‌ی پراسسی (PID) است که لینوکس به اسکریپت ما اختصاص داده است.

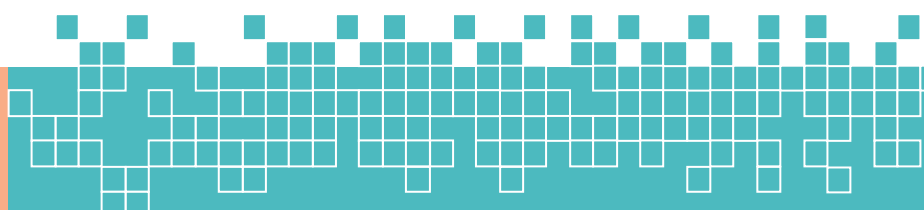
پس تا اینجا فهمیدیم که هر برنامه‌ای که داخل یک ترمینال اجرا شود، یک شماره‌ی جاب منحصر به فرد و هر برنامه‌ای که در سیستم لینوکس اجرا شود، یک شماره‌ی پراسس منحصر به فرد خواهد داشت. همانطور که می‌بینید، پس از اجرای اسکریپت در پشت صحنه و مشاهده‌ی شماره‌ی جاب و شماره‌ی پراسس آن، می‌توانیم با فشار دادن دکمه‌ی Enter، بار دیگر پرامپت شیل را دریافت کرده و مانند قبل با سیستم کار کنیم.

اما اینجا یک مشکل وجود دارد؛ با این که اسکریپت ما در پشت صحنه در حال اجرا است، اما اسکریپت باز هم از صفحه‌ی ترمینال ما برای نمایش خروجی خود استفاده می‌کند. اگر دقت کنید می‌بینید که هر چند لحظه یک بار، اسکریپت ما خروجی خود را روی صفحه‌ی ما نشان می‌دهد و عملاً استفاده از شیل را بسیار دشوار می‌کند. برای رفع این مشکل، می‌توانیم با استفاده از ریدایرکتورها خروجی اسکریپت را در یک فایل بریزیم (یعنی `./script22.sh > out.txt &`).

به محض اتمام اسکریپت اجرایی در پشت صحنه، شیل پیام زیر را به ما نشان می‌دهد:

```
[1]+ Done ./script22.sh
```

این پیام، شماره‌ی جاب و همچنین وضعیت آن (Done) به اضافه‌ی نام دستوری که این جاب را استارت زده (`./script22.sh`) را به ما نشان می‌دهد.



فرستادن چندین اسکریپت به پشت صحنه

ما می‌توانیم هر تعداد اسکریپت را به پشت صحنه بفرستیم. برای مثال:

```
[root@localhost ~]# ./script22.sh &
[1] 8828
Loop #1
[root@localhost ~]# ./script22.sh &
[2] 8831
Loop #1
[root@localhost ~]# ./script22.sh &
[3] 8834
Loop #1
[root@localhost ~]# ./script22.sh &
[4] 8838
Loop #1
Loop #2
Loop #2
Loop #2
[...]
```

همانطور که می‌بینید، با فرستادن هر اسکریپت به پشت صحنه، شیل به آن اسکریپت یک شماره‌ی جاب اختصاص می‌دهد و لینوکس نیز به هر جاب، یک PID اختصاص می‌دهد. ما می‌توانیم صحت در حال اجرا بودن این اسکریپت‌ها را با مشاهده‌ی خروجی دستور `ps au` بررسی کنیم:

```
[root@localhost ~]# ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      7645  0.0  0.2 115580 2228 pts/1    Ss   10:20   0:00 -bash
root      8828  0.0  0.1 113188 1404 pts/1    S   12:33   0:00 /bin/bash ./script22.sh
root      8831  0.0  0.1 113188 1404 pts/1    S   12:33   0:00 /bin/bash ./script22.sh
root      8834  0.0  0.1 113188 1400 pts/1    S   12:33   0:00 /bin/bash ./script22.sh
root      8838  0.0  0.1 113188 1404 pts/1    S   12:33   0:00 /bin/bash ./script22.sh
root      8868  0.0  0.0 107956 356 pts/1    S   12:33   0:00 sleep 2
root      8869  0.0  0.0 107956 356 pts/1    S   12:33   0:00 sleep 2
root      8870  0.0  0.0 107956 352 pts/1    S   12:33   0:00 sleep 2
root      8871  0.0  0.0 107956 356 pts/1    S   12:33   0:00 sleep 2
root      8873  0.0  0.1 155372 1856 pts/1    R+   12:33   0:00 ps au
```

همانطور که می‌بینید تک تک پراسس‌هایی در پشت صحنه اجرا کردیم، در خروجی دستور `ps` نشان داده شده‌اند. نکته‌ای که باید در اینجا به آن توجه کنیم، موارد نوشته شده در ستون TTY می‌باشد. همانطور که می‌بینید تک‌تک اسکریپت‌هایی که در پشت صحنه اجرا کردیم، به Session ترمینال pts/1 وابسته هستند. این بدین معنی است که به محضی که این Session قطع شود یا از این ترمینال خارج شویم، کلیه‌ی پراسس‌های موجود در پشت صحنه نیز در هر کجای کار که باشند، قطع خواهد شد. این امر می‌تواند برای ما مشکل ساز باشد. در بخش بعد در مورد چگونگی رفع این مشکل صحبت خواهیم کرد.

اجرای اسکریپت‌ها بدون وابستگی به یک کنسول

گاهی اوقات لازم است یک اسکریپت را از یک ترمینال اجرا کنیم و کاری کنیم که حتی پس از خارج شدن ما از آن ترمینال، اسکریپت در پشت صحنه در حال انجام کار خود باشد. ما می‌توانیم این کار را با استفاده از دستور `nohup` انجام دهیم. ما قبلاً در مورد دستور `nohup` و چگونگی استفاده از آن صحبت کرده بودیم، اما در این بخش نیز به توضیح آن می‌پردازیم.

هنگامی که از یک ترمینال خارج می‌شویم (مثلاً وقتی ارتباط SSH خود را با سرور قطع می‌کنیم)، لینوکس یک

سیگنال SIGHUP به کلیه‌ی دستورهای که توسط ترمینال ما ایجاد شده بوده می‌فرستد. این سیگنال باعث می‌شود که آن پراسس‌ها در هر کجای کار که هستند، خاتمه یابند. دستور nohup، باعث می‌شود که پراسس‌هایی که توسط ترمینال ما اجرا شده‌اند، به سیگنال SIGHUP پاسخی نداده و در نتیجه، حتی پس از خارج شدن ما از ترمینال به کار خود ادامه دهند.

لازم به ذکر است که استفاده از دستور nohup باعث اجرای دستورهای ما در پشت صحنه نمی‌شود. ما می‌توانیم دستور nohup را با علامت امپرسند ترکیب کرده تا یک پراسس را در بک‌گراند (بدون ریسک خاتمه یافتن آن هنگام خروج از ترمینال) اجرا کنیم. یعنی:

```
[root@localhost ~]# nohup ./script22.sh &
```

```
[1] 9221
```

```
[root@localhost ~]# nohup: ignoring input and appending output to 'nohup.out'
```

همانطور که می‌بینید، دقیقاً مثل قبل، شیل یک شماره‌ی جاب به این پراسس اختصاص می‌دهد و لینوکس نیز یک شماره‌ی پراسس به آن اختصاص می‌دهد. تنها تفاوت در این است که nohup باعث می‌شود اسکریپت ما به SIGHUP پاسخی ندهد و به علاوه، nohup خروجی اسکریپت را به جای نمایش روی صفحه‌ی ترمینال، داخل فایلی به نام nohup.out در دایرکتوری کنونی می‌ریزد.

فایل nohup.out، کلیه‌ی مواردی که توسط برنامه روی ترمینال نشان داده می‌شده (STDOUT) را داخل خود ذخیره می‌کند. پس از پایان پراسس، ما می‌توانیم محتویات این فایل را مشاهده کرده تا از چگونگی عملکرد پراسس باخبر شویم.

```
[root@localhost ~]# cat nohup.out
```

```
Loop #1
```

```
[...]
```

```
Loop #99
```

```
Loop #100
```

همانطور که می‌بینید، دقیقاً همان مواردی که قبلاً روی ترمینال نشان داده میشد، الان داخل فایل nohup.out قرار گرفته است.

نکته: توجه کنید که اگر یک دستور دیگر را در در همین دایرکتوری با استفاده از nohup اجرا کنیم، خروجی آن دستور نیز به محتویات فایل nohup.out چسبیده می‌شود. پس هنگام اجرای چندین دستور با استفاده از nohup در یک دایرکتوری، باید حواسمان به فایل خروجی باشد. ما می‌توانیم با ریدایرکت کردن خروجی nohup، کاری کنیم که nohup به جای نوشتن خروجی داخل فایل nohup.out، آن را در فایل مورد نظر خود ما ریدایرکت کند.

ارسال سیگنال

ما می‌توانیم به کلیه پراسس‌های موجود در سیستم، سیگنال‌های کنترلی متفاوتی ارسال کنیم. از میان این سیگنال‌ها، دو سیگنال را می‌توانیم با استفاده از کلیدهای کیبورد ارسال کنیم. این دو سیگنال، می‌توانند یک پراسسی که روی ترمینال ما در حال اجرا می‌باشد را Interrupt کرده یا آن را Stop کند.



اینترپت کردن (Interrupt) پراسس

فشردن دکمه‌های CTRL+C، یک سیگنال SIGINT به پراسسی که در حال حاضر روی ترمینال ما در حال اجرا می‌باشد ارسال می‌کند و آن پراسس را کاملاً متوقف می‌کند. بیایید با استفاده از دستور sleep این امر را امتحان کنیم:

```
[root@localhost ~]# sleep 100
```

```
^C
```

```
[root@localhost ~]#
```

همانطور که می‌بینید، به محض زدن دکمه‌ی CTRL+C، بدون دریافت هیچ پیامی، از پراسس sleep خارج شدیم.

استاپ کردن (Stop) یک پراسس

لینوکس به ما امکان می‌دهد که یک پراسس در حال اجرا را استاپ کنیم. این کار در برخی از اوقات می‌تواند خطرناک باشد (مثلاً وقتی که اسکریپت یا برنامه‌ی استاپ شده در حال نوشتن یا خواندن یک فایل باشد)، اما در اکثر اوقات، مشکلی برای ما ایجاد نمی‌کند.

ما می‌توانیم با فشردن دکمه‌های CTRL+Z، یک سیگنال SIGSTP به پراسسی که در حال حاضر روی ترمینال ما در حال اجرا می‌باشد ارسال کنیم. پراسس‌های استاپ شده در RAM سیستم باقی می‌مانند و ما می‌توانیم بار دیگر آنها را زنده کرده و از آنها بخواهیم که به کار خود را از جایی که استاپ شده‌اند ادامه دهند. ما قبلاً در مورد چگونگی زنده کردن یک پراسس استاپ شده صحبت کرده‌ایم، اما در بخش بعد، باز هم این امر را توضیح می‌دهیم.

بیایید استاپ کردن یک پراسس را امتحان کنیم:

```
[root@localhost ~]# sleep 100
```

```
^Z
```

```
[1]+  Stopped                  sleep 100
```

همانطور که می‌بینید، هنگام استفاده از CTRL+Z برای استاپ کردن یک پراسس، شیل یک پیام در خروجی به ما نشان می‌دهد. اگر ما یک سری پراسس استاپ شده داشته باشیم و بخواهیم از ترمینال کنونی خارج شویم، bash به ما هشداری در مورد وجود پراسس‌های استاپ شده می‌دهد و اجازه‌ی خروج را به ما نمی‌دهد:

```
[root@localhost ~]# exit
```

```
logout
```

```
There are stopped jobs.
```

```
[root@localhost ~]#
```

در این حالت، یا باید پراسس استاپ شده را کاملاً متوقف کنیم یا باید بار دیگر دستور exit را بنویسیم تا از ترمینال خارج شویم.

کنترل جاب‌ها

در بخش قبل، با استفاده از کلیدهای CTRL+Z، یک پراسس در حال اجرا روی ترمینال خود را استاپ کردیم. پس از استاپ کردن یک پراسس (یا یک Job)، لینوکس به ما این امکان را می‌دهد که آن جاب را از سر بگیریم یا آن را به صورت کامل متوقف کنیم. از سر گرفتن یک جاب، با ارسال سیگنال SIGCONT به آن پراسس صورت می‌پذیرد.

عمل استارت زدن، استاپ کردن یا متوقف کردن یک پراسس، Job Control نام دارد. جاب کنترل به ما امکان می‌دهد که چگونگی اجرای پراسس‌ها در ترمینال خود را مدیریت کنیم. در این بخش، با دستورهای موجود برای کنترل جاب‌ها آشنا می‌شویم.

مشاهده‌ی جاب‌ها

کلیدی‌ترین دستور برای مدیریت جاب‌ها، دستور jobs می‌باشد. این دستور جاب‌های کنونی زیر نظر شل کنونی را به ما نشان می‌دهد. بیایید یک اسکریپت نوشته و با اجرا و متوقف کردن آن، چگونگی مشاهده‌ی جاب‌ها را بررسی کنیم:

```
[root@localhost ~]# cat script23.sh
```

```
#!/bin/bash
```

```
# I may not have a job but my OS does
```

```
echo "Testing... Testing... PID: $$"
```

```
count=1
```

```
while [ $count -le 10 ] ; do
```

```
    echo "Loop #$count"
```

```
    sleep 10
```

```
    count=$(( $count + 1 ))
```

```
done
```

همانطور که می‌بینید، اسکریپت بالا بسیار ساده می‌باشد و نیاز به توضیح خاصی ندارد. تنها نکته‌ی جدید در این اسکریپت، استفاده‌ی ما از متغیر محلی \$ (که آن را به صورت \$\$ فراخواندیم) می‌باشد. این متغیر، PID که لینوکس به این اسکریپت می‌دهد را درون خود ذخیره می‌کند. پس با این حساب، اسکریپت ما اول PID اختصاص داده به خود را روی صفحه‌ی ترمینال نشان می‌دهد و سپس وارد یک حلقه شده و در هر پیمایش، شماره‌ی پیمایش کنونی را روی صفحه نشان داده، سپس ده ثانیه صبر می‌کند و بعد به سراغ پیمایش بعدی می‌رود و این کار را تا جایی ادامه می‌دهد که متغیر count مقداری کمتر یا برابر با ۱۰ داشته باشد.

ما می‌خواهیم این اسکریپت را ۲ دفعه اجرا کنیم. یک بار از این اسکریپت با استفاده از دکمه‌ی Ctrl+Z بیرون می‌آییم و بار دیگر، با استفاده از علامت امپرسند (&)، اسکریپت را به بک‌گراند می‌فرستیم (و خروجی آن را درون یک فایل Redirect می‌کنیم تا صفحه‌ی ترمینال را شلوغ نکند). پس:

```
[root@localhost ~]# chmod u+x script23.sh
```

```
[root@localhost ~]# ./script23.sh
```

```
Testing... Testing... PID: 20288
```

```
Loop #1
```

```
^Z
```

```
[1]+  Stopped                  ./script23.sh
```

```
[root@localhost ~]# ./script23.sh > script23.out &
```

```
[2] 20290
```

حال بیایید با وارد کردن دستور jobs، جاب‌های در حال اجرا در ترمینال کنونی را مشاهده کنیم:

```
[root@localhost ~]# jobs
```

```
[1]+  Stopped                  ./script23.sh
```

```
[2]-  Running                  ./script23.sh > script23.out &
```

همانطور که می‌بینید، دستور jobs، هم جاب‌های متوقف شده و هم جاب‌های در حال اجرا را به ما نشان داده و علاوه بر آن، شماره‌ی جاب و همچنین دستوری که آن جاب را ایجاد کرده را نیز به ما نشان می‌دهد.

دستور jobs چندین آپشن دارد که به شرح زیر می‌باشند:

جدول ۲- آپشن‌های دستور jobs

آپشن	عملکرد
-l	PID هر جاب را در کنار جاب نامبر به ما نشان می‌دهد.
-n	فقط جاب‌هایی که وضعیتشان از آخرین وضعیت گزارش داده شده روی شل دچار تغییر شده است را نشان می‌دهد.
-p	فقط PID جاب‌ها را نشان می‌دهد.
-r	فقط جاب‌های در حال اجرا (Running) را نشان می‌دهد.
-s	فقط جاب‌های استاپ شده را نشان می‌دهد.

اگر به خروجی jobs نگاه کنید، می‌بینید که پس از شماره‌ی جاب، یک علامت + و پس از شماره‌ی جاب دیگر، یک علامت - قرار گرفته است. جایی که دارای علامت + می‌باشد، جاب پیش‌فرض یا default job نام دارد. دستورهای کنترل جاب در صورت مشخص نبودن شماره‌ی جایی که باید روی آن عملیاتی انجام دهند، عملیات خود را روی جاب پیش‌فرض انجام می‌دهند. جایی که دارای علامت - می‌باشد، جایی است که پس از اتمام جاب دیفالت کنونی، تبدیل به جاب دیفالت خواهد شد. ما در هر زمان، فقط می‌توانیم یک جاب دارای علامت + و یک جاب دارای علامت - داشته باشیم.

بیایید این امر را با هم بررسی کنیم:

```
[root@localhost ~]# ./script23.sh
Testing... Testing... PID: 20581
Loop #1
^Z
[1]+  Stopped                  ./script23.sh
[root@localhost ~]# ./script23.sh
Testing... Testing... PID: 20583
Loop #1
^Z
[2]+  Stopped                  ./script23.sh
[root@localhost ~]# ./script23.sh
Testing... Testing... PID: 20585
Loop #1
^Z
[3]+  Stopped                  ./script23.sh
[root@localhost ~]# jobs
[1]  20581 Stopped              ./script23.sh
[2]- 20583 Stopped              ./script23.sh
[3]+ 20585 Stopped              ./script23.sh
```

همانطور که می‌بینید، ما script23.sh را سه بار اجرا کردیم و هر دفعه، آن را استاپ کردیم. اگر به خروجی jobs توجه کنید، می‌بینید که از بین ۳ جاب کنونی ایجاد شده توسط این ترمینال، فقط یکی از آنها علامت + و فقط یکی از آنها علامت - را دارد. اگر ما دستور % kill را بدون مشخص کردن شماره‌ی جاب وارد کنیم، این دستور به صورت اتوماتیک جاب پیش‌فرض، یعنی جاب سوم (دارای PID برابر با ۲۰۵۸۵) را متوقف می‌کند:

```
[root@localhost ~]# kill %
[3]+  Stopped                  ./script23.sh
[3]+  Terminated              ./script23.sh
```



```
[root@localhost ~]# jobs -l
[1] - 20581 Stopped                ./script23.sh
[2]+ 20583 Stopped                ./script23.sh
```

همانطور که می‌بینید، به محض متوقف شدن جاب پیش‌فرض، جاب دوم (دارای PID برابر با ۲۰۵۸۳) تبدیل به جاب پیش‌فرض شد و جاب اول، تبدیل به جابی شد که پس از اتمام جاب دوم، تبدیل به جاب پیش‌فرض می‌شود.

از سرگیری (زنده کردن) جاب‌های استاپ شده

همانطور که گفتیم، ما می‌توانیم جاب‌های استاپ شده را بار دیگر زنده کنیم. جاب‌ها می‌توانند در بک‌گراند سیستم از سر گرفته شده یا در Foreground سیستم از سر گرفته شوند. نکته‌ای که باید به آن توجه داشته باشیم این است که اگر جاب در فور‌گراند از سر گرفته شود، کنترل شل را تا زمان پایان کار خود از دست ما می‌گیرد.

برای از سرگیری یک جاب و قرار دادن آن در بک‌گراند، از دستور bg به علاوه‌ی شماره‌ی جاب مورد نظر استفاده می‌کنیم:

```
[root@localhost ~]# jobs
[1] - Stopped                ./script23.sh
[2]+ Stopped                ./script23.sh
[root@localhost ~]# bg 2
[2]+ ./script23.sh &
[root@localhost ~]# Loop #2
[root@localhost ~]# jobs
[1]+ Stopped                ./script23.sh
[2]- Running                ./script23.sh &
[root@localhost ~]# Loop #3
Loop #4
Loop #5
Loop #6
Loop #7
Loop #8
Loop #9
Loop #10

[2]- Done                ./script23.sh
```

همانطور که می‌بینید، در ابتدا دو جاب موجود در ترمینال کنونی، استاپ شده‌اند. ما با وارد کردن دستور bg و مشخص کردن شماره‌ی جاب ۲، به ترمینال گفتیم که جاب دارای شماره‌ی جاب ۲ را زنده کند. به محض وارد کردن این دستور، جاب دارای شماره‌ی ۲ زنده شد و در وضعیت Running قرار گرفت و به ادامه‌ی کار خود پرداخت تا در نهایت کارش تمام شد و در وضعیت Done قرار گرفت. از آنجایی که ما این جاب را در بک‌گراند زنده کردیم، در حین اجرای این جاب، توانستیم از ترمینال استفاده کنیم و دستورهای دیگر را در سیستم وارد کنیم.

برای زنده کردن یک جاب در فور‌گراند سیستم، از دستور fg به علاوه‌ی شماره‌ی جاب مورد نظر استفاده می‌کنیم:

```
[root@localhost ~]# jobs
[1]+ Stopped                ./script23.sh
[root@localhost ~]# fg 1
./script23.sh
```

Loop #2
Loop #3

همانطور که می‌بینید، به محض وارد دستور 1 fg، جاب دارای شماره‌ی جاب ۱ شروع به ادامه‌ی کار خود در فورگراند می‌کند و به همین دلیل، کنترل ترمینال را از ما گرفته و ما نمی‌توانیم تا به پایان رسیدن این جاب، از این ترمینال استفاده کنیم.

اجرای اتوماتیک اسکریپت‌ها در زمان‌های مشخص

بسیاری از اوقات، نیاز داریم که یک اسکریپت در یک زمان خاص، بدون این که ما پشت سیستم باشیم، اجرا شود. ما می‌توانیم با استفاده از ابزارهای زمان‌بندی عملیات در لینوکس، این نیازمندی را برآورده کنیم. اگر به خاطر داشته باشید، ما در جلسه‌ی قبل به طور خیلی مفصل در مورد چگونگی زمان‌بندی عملیات در لینوکس صحبت کردیم. در این بخش، می‌خواهیم به صورت کاربردی‌تر با این ابزارها آشنا شویم. ما به چگونگی زمان‌بندی اسکریپت‌ها با دو ابزار at و cron، نگاه خواهیم کرد.

زمان‌بندی اسکریپت با استفاده از دستور at

دستور at به ما امکان می‌دهد که به لینوکس بگوییم که یک اسکریپت یا جاب را در چه زمانی اجرا کند. این دستور، هر جاب را داخل یک صف قرار می‌دهد. جاب‌های موجود در صف، شامل اطلاعاتی مربوط به زمان اجرا شدن توسط شیل می‌باشند. دستور دیگری به نام atd که به صورت دائم در پشت صحنه در حال اجراست دائماً صف جاب‌ها را بررسی کرده و اگر زمان اجرای یک جاب فرا رسیده باشد، آن جاب را اجرا می‌کند. دستور atd، این کار را با بررسی یک دایرکتوری، که معمولاً /var/spool/at می‌باشد، انجام می‌دهد. این دایرکتوری، کلیه‌ی جاب‌هایی که کاربر از طریق دستور at زمان‌بندی کرده را درون خود دارد. به صورت پیش‌فرض، atd هر ۶۰ ثانیه یک بار محتویات این دایرکتوری را بررسی می‌کند. اگر جابی در این دایرکتوری وجود داشته باشد، atd زمانی که برای اجرای آن جاب مشخص شده را بررسی کرده و اگر آن زمان برابر با زمان کنونی باشد، آن جاب را اجرا می‌کند. در بخش بعدی با چگونگی زمان‌بندی جاب با at آشنا می‌شویم.

استفاده از دستور at

به طور کلی، دستور at از فرمت زیر پیروی می‌کند (موارد نوشته شده داخل کروشه اختیاری می‌باشند):
at [-f filename] time

همانطور که در جلسه‌ی قبل دیدیم، در حالت پیش‌فرض، دستور at یک زمان برابر با time از ما دریافت کرده، سپس یک پرامپت در اختیار ما قرار می‌دهد تا در آن پرامپت، دستورهایی می‌خواهیم در زمان مشخص شده اجرا شوند را وارد کنیم.

به جای این کار، ما می‌توانیم با استفاده از آپشن -f، یک فایل اسکریپت به دستور at معرفی کرده تا at، فایل اسکریپت مورد نظر ما را داخل صف قرار دهد و آن را در زمان برابر با time، اجرا کند.

ما می‌توانیم زمان time را با فرمت‌های متفاوتی به دستور at بدهیم. از انواع فرمت‌های مشخص کردن زمان اجرا، می‌توان به موارد زیر اشاره کرد:

- مشخص کردن ساعت و دقیقه به صورت استاندارد، مثلاً: ۲۳:۱۵
- استفاده از AM/PM، مثلاً: ۱۱:۱۵PM

- مشخص کردن نام یک زمان، مثل noon، now، midnight یا teatime (ساعت ۴ بعد از ظهر)
- از فرمت‌های قابل استفاده برای مشخص کردن تاریخ اجرا می‌توان به موارد زیر اشاره کرد:
- استفاده از فرمت‌های استاندارد تاریخ، مثل MMDDYY، MM/DD/YY یا MM.DD.YY
- مشخص کردن روز و نام ماه، با یا بدون سال، مثلا 4 Jul یا 25 Dec 2021.
- استفاده از فرمت‌هایی نظیر:
 - Now + 25 minutes
 - 10:15PM tomorrow
 - 10:15 + 7 days

صف‌های at

هنگام استفاده از دستور at، جاب مورد نظر داخل یک صف جاب، یا Job Queue قرار می‌گیرد. دستور at، ۲۶ صف جاب با اولویت‌های متفاوت دارد. صف‌های جاب، با استفاده از حروف کوچک انگلیسی a تا z مشخص می‌شوند. به صورت پیش‌فرض، همه‌ی جاب‌ها، در صف جاب دارای بالاترین اولویت (a) قرار می‌گیرند. اگر بخواهیم یک جاب در صف دارای اولویت پایین‌تر قرار گیرد، کافی است نام صف را با آپشن -q به دستور at بدهیم.

دسترسی به خروجی جاب‌ها

زمانی که اجرای یک جاب را به دستور at می‌سپاریم، خروجی اجرای آن جاب روی ترمینال ما نمایش داده نمی‌شود، بلکه خروجی جاب به ایمیل (معمولا لو کال) یوزر ایجاد کننده‌ی جاب ارسال می‌شود. برای مثال:

```
[root@localhost ~]# cat script24.sh
#!/bin/bash
# Testing the at command

echo "It's $(date)! Wahooo!"

[root@localhost ~]# chmod u+x script24.sh
[root@localhost ~]# at -f script24.sh now + 1 min
job 6 at Tue Mar 9 10:08:00 2021
[root@localhost ~]# cat /var/mail/root
[...]
```

From root@localhost.localdomain Tue Mar 9 10:08:00 2021
 Return-Path: <root@localhost.localdomain>
 X-Original-To: root
 Delivered-To: root@localhost.localdomain
 Received: by localhost.localdomain (Postfix, from userid 0)
 id 8DB7315C0A6; Tue, 9 Mar 2021 10:08:00 +0330 (+0330)
 Subject: Output from your job 6
 To: root@localhost.localdomain
 Message-Id: <20210309063800.8DB7315C0A6@localhost.localdomain>
 Date: Tue, 9 Mar 2021 10:08:00 +0330 (+0330)
 From: root@localhost.localdomain (root)

It's Tue Mar 9 10:08:00 +0330 2021! Wahooo!

همانطور که می‌بینید، ما یک اسکریپت ساده که تاریخ را echo می‌کرد ایجاد کردیم و سپس آن اسکریپت را به دستور at دادیم و از at خواستیم که پس از یک دقیقه از الان، این اسکریپت را اجرا کند. پس از گذشت

یک دقیقه، `at` این اسکریپت را اجرا کرد و خروجی که این اسکریپت قرار بود نمایش دهد (چه `STDOUT` و چه `STDERR`) را به آدرس ایمیل لوکال کاربری که جاب را ایجاد کرده ارسال کرد. اگر اسکریپت یا دستور ما از خود خروجی تولید نکند، دستور `at` به صورت پیش فرض چیزی به کاربر ایمیل نمی‌کند. اگر `at` را با آپشن `-m` اجرا کنیم، حتی اگر اسکریپت یا دستور داده شده به `at` خروجی نداشته باشد، `at` یک ایمیل مبنی بر تکمیل کردن جابی که به او سپرده‌ایم به ما ارسال می‌کند.

نکته: ایمیل‌های ارسال شده به کاربران در توزیع CentOS، در دایرکتوری `/var/mail` قرار می‌گیرد. در این دایرکتوری، یک فایل به نام هر کاربر موجود در سیستم وجود دارد که داخل آن، همه‌ی ایمیل‌های ارسال شده به کاربر ذخیره می‌شود.

مشاهده‌ی جاب‌های موجود در صف `at`

دستور `atq`، جاب‌های موجود در صف برنامه‌ی `at` را به ما نشان می‌دهد:

```
[root@localhost ~]# at -f ./script24.sh 13:00 tomorrow
job 7 at Wed Mar 10 13:00:00 2021
[root@localhost ~]# at -f ./script24.sh teatime
job 8 at Tue Mar 9 16:00:00 2021
[root@localhost ~]# at -f ./script24.sh 11:30PM
job 9 at Tue Mar 9 23:30:00 2021
[root@localhost ~]# atq
7      Wed Mar 10 13:00:00 2021 a root
8      Tue Mar 9 16:00:00 2021 a root
9      Tue Mar 9 23:30:00 2021 a root
```

همانطور که می‌بینید، دستور `atq` شماره‌ی جاب، روز، تاریخ و ساعت اجرای هر کدام را به ما نشان می‌دهد. همانطور که می‌بینید، پس از نشان دادن سال (۲۰۲۱)، حرف `a` را مشاهده می‌کنیم. `a`، نشان دهنده‌ی صفی می‌باشد که این جاب درون آن ذخیره شده است.

حذف یک جاب از صف `at`

ما می‌توانیم با استفاده از دستور `atrm` و شماره‌ی جاب در صف `at`، آن جاب را حذف کنیم:

```
[root@localhost ~]# atrm 8
[root@localhost ~]# atq
7      Wed Mar 10 13:00:00 2021 a root
9      Tue Mar 9 23:30:00 2021 a root
```

همانطور که می‌بینید، با وارد کردن دستور `atrm` و مشخص کردن شماره‌ی جابی که می‌خواهیم حذف شود، توانستیم جاب را حذف کنیم.

اجرای اسکریپت‌ها در بازه‌های زمانی مشخص

دستور `at`، یک اسکریپت را فقط یک بار در یک زمان مشخص شده اجرا می‌کند. این امر می‌تواند بسیار کاربردی باشد، اما اگر بخواهیم یک دستور در یک زمان مشخص به صورت روزانه، هفتگی و... اجرا شود باید چه کنیم؟ در چنین حالتی، باید به سراغ سرویس `cron` برویم. ما در جلسه‌ی قبل در مورد سرویس `cron` در لینوکس صحبت کردیم و با چگونگی ایجاد کرون‌جاب‌ها آشنا شدیم. در این قسمت، بار دیگر به توضیح چگونگی عملکرد کرون می‌پردازیم.



اگر به خاطر داشته باشید، گفتیم که کرون دائماً در پشت صحنه‌ی سیستم در حال اجرا می‌باشد و به صورت مداوم، جدول‌های ویژه‌ای که به آن جدول کرون یا crontab می‌گویند را بررسی می‌کند. هر کاربر در سیستم، یک جدول کرون مخصوص به خود دارد و هر کاربر می‌تواند با وارد کردن دستور زیر به آن دسترسی پیدا کند:

```
[root@localhost ~]# crontab -e
```

در این جدول، بازه‌ی اجرایی و همچنین برنامه یا اسکریپتی که باید اجرا شود را مشخص می‌کنیم. می‌دانیم که موارد نوشته شده در جدول کرون، دارای چندین فیلد به شرح زیر می‌باشند:

```
min hour dayofmonth month dayofweek command
```

جدول کرون به ما امکان می‌دهد که در هر فیلد، از یک مقدار مشخص، محدوده‌ای از مقادیر (مثلاً ۴-۲) یا از وایلد کاردها (مثل *) استفاده کنیم. مثلاً اگر بخواهیم یک اسکریپت هر روز در ساعت ۱۰:۱۵ صبح اجرا شود، به صورت زیر عمل می‌کنیم:

```
15 10 * * * /home/root/script1.sh
```

وایلد کارد * در فیلدهای سوم، چهارم و پنجم، به این معنا هست که کرون، اسکریپت مشخص شده را هر روز هر ماه در ساعت ۱۰:۱۵ دقیقه اجرا می‌کند.

اگر بخواهیم یک اسکریپت در روز دوشنبه‌ی هر هفته‌ی هر ماه در ساعت ۴:۱۵ عصر اجرا شود، به صورت زیر عمل می‌کنیم:

```
15 16 * * 1 /home/root/script1.sh
```

همانطور که می‌بینید، ما در فیلد پنجم، روزی از هفته که می‌خواهیم اسکریپت در آن اجرا شود را مشخص کردیم. البته ما می‌توانیم در این فیلد به جای شماره‌ی روز، از سه حرف اول نام روز، نظیر tue, mon ... استفاده کنیم.

اگر بخواهیم یک اسکریپت در ساعت ۱۲ ظهر روز اول هر ماه اجرا شود، به صورت زیر عمل می‌کنیم:

```
00 12 1 * * /home/root/script1.sh
```

همانطور که می‌بینید، ما در فیلد سوم، روزی از ماه که می‌خواهیم اسکریپت در آن اجرا شود را مشخص کردیم. این فیلد می‌تواند مقداری بین ۱ تا ۳۱ داشته باشد.

نکته: دقت کنید که برای مشخص کردن اسکریپت‌هایی که می‌خواهیم توسط کرون اجرا شوند، بهتر است از آدرس کامل اسکریپت‌ها استفاده کنیم.

کرون هنگام اجرای یک اسکریپت، خروجی آن را روی صفحه به ما نشان نمی‌دهد، اما در برخی از توزیع‌ها، خروجی اسکریپت را به ما ایمیل می‌کند و علاوه بر آن، ما می‌توانیم هنگام تعریف کرون‌جاب، خروجی اسکریپت را ریدایرکت کنیم. برای مثال:

```
15 10 * * * /home/root/script1.sh > script1.sh
```

برنامه‌ی کرون، اسکریپت‌ها را به عنوان کاربری که کرون‌جاب را ایجاد کرده اجرا می‌کند، پس هنگام ایجاد کرون‌جاب‌ها، باید حواسمان به مجوزهای دسترسی باشد.