



دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

دستور کار جلسه دوم

پاییز ۱۴۰۲



فهرست مطالب

2	پوسته (Shell)	1
8	اسکرپت نویسی (Script)	2
8	زبان اسکرپت نویسی	3
8	انتخاب پوسته برای اجرای اسکرپت	4
12	ابزارهای برنامه نویسی	5
15	اجرای دستورات خط فرمان در برنامه	6
15	بکارگیری ابزار Make در فرآیند برنامه نویسی	7



1 پوسته (Shell)

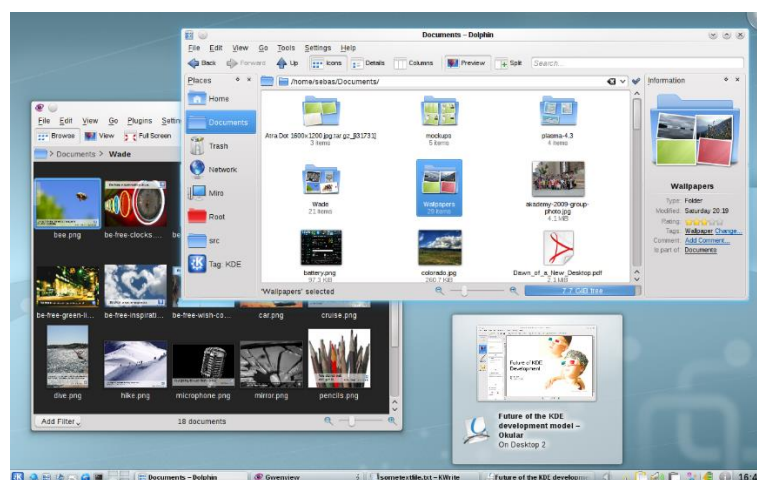
پوسته محیطی است که کاربر اطلاعات خود را در آن وارد می کند، وظیفه پوسته ترجمه ی این دستورات با دستورات سیستمی و در نهایت ارسال آنها به هسته است. پوسته ها به دو دسته تقسیم می شوند، که در ادامه معرفی می شوند.

1.1 Graphical User Interface (GUI)

در این نوع رابطی گرافیکی وجود دارد که اطلاعات دریافتی از ورودی های مختلف به دستوری قابل فهم برای هسته تبدیل می شود و اجرا می شود. شکل ۱ و ۲ تصاویری از مطرح ترین رابط های گرافیکی موجود را نمایش می دهد.



شکل ۱ . 3.1 Gnome



شکل ۲ . KDE Plasma Desktop



1.2 Command Line Interface (CLI)

در این محیط کاربر تمامی دستورات را با صفحه کلید وارد می کند، مشکل این روش، به خاطر سپردن تعداد زیادی از دستورات و گاهی خطاهایی است که در هنگام وارد کردن دستورات رخ می دهد. اما مزیت بزرگ این روش استفاده از پوسته برای خودکارسازی فرآیندهای تکراریست. چهار پوسته مشهور عبارتند از :

- Bourne Shell (sh)
- C Shell (csh)
- Bourne Again Shell (bash)
- Korn Shell (ksh).
- Z Shell (zsh)

1.3 متغیرهای محیطی (Environmental Variables)

هر پوسته قبل از اجرا شدن تعدادی متغیر محیطی را از فایل های پیکربندی می خواند، این متغیرها برای تمامی پروسس های ساخته شده در آن پوسته و دستوراتی که در آن اجرا می شود قابل دسترسی می باشند (مقادیر آنها به ارث می رسد). در این حالت تغییر دادن مقدار متغیر در یک پروسس، مقدار آن را در پوسته تغییر نخواهد داد.

در محیط پوسته می توان متغیرهای موجود را مقدار دهی کرد. همچنین می توان متغیرهای محیطی جدید و متغیرهای پوسته جدید را تعریف کرد، متغیرهای پوسته همانند متغیرهای محیطی هستند با این تفاوت که فقط در همان پوسته قابل دسترس هستند و پروسس هایی که در پوسته ساخته می شوند به متغیرهای پوسته دسترسی ندارند.

1.3.1 تعریف متغیر پوسته

برای تعریف متغیر در پوسته نیازی به تعریف نوع آن (رشته، صحیح، اعشاری و...) نیست:

```
variable_name="value"
```

نکته : در هر دو طرف = نباید فاصله ای وجود داشته باشد.



نکته : اگر مقدار متغیر یک قسمت داشته باشد لزومی به استفاده از " " در دو طرف آن نیست ولی برای مقادیری که بین آنها جداکننده‌ای وجود دارد، قرار دادن " " الزامی است. برای مثال اگر متغیر device را داشته باشیم و بخواهیم مقدار pc را در آن ذخیره کنیم می‌توانیم به شکل زیر بنویسیم (با استفاده از echo می‌توان مقدار متغیر را مشاهده کرد، خط آخر خروجی دستور echo است):

```
device="laptop"  
device=$device" pc"  
echo $device  
laptop pc
```

نکته : پوسته همه متغیرها را به عنوان رشته (string) در نظر می‌گیرد ولی خود قابلیت تفکیک اعداد و مقادیر حسابی از رشته‌ها را داراست و در مواقع لزوم می‌توان عملیات حسابی را بر روی متغیرها اعمال کرد.

```
device=pc یا device="pc"
```

ولی برای ذخیره مقدار laptop pc در آن باید به این شکل نوشته شود :

```
device="laptop pc"
```

Export 1.3.2

محدوده متغیرهای محیطی که در حالت قبل تعریف می‌شوند در یک پوسته است و پوسته‌های اجرا شده در پوسته کنونی از مقدار آنها بی‌اطلاعند. اگر بخواهیم متغیری محیطی تعریف کنیم که در پوسته‌هایی که از این پس اجرا می‌شوند نیز قابل دسترسی باشند:

```
export variable_name=value
```

با اضافه کردن export متغیر به عنوان متغیری محیطی شناخته می‌شود و به پروسس‌هایی که در این پوسته ساخته می‌شوند نیز به ارث می‌رسد.

Echo 1.3.3

برای نمایش مقدار یک متغیر به کار می‌رود :

```
device="laptop"  
echo $device
```



laptop

نکته : در صورتی که رشته در یک خط قابل نمایش باشد نیازی به استفاده از " " در دو طرف آن نیست ولی اگر لازم باشد رشته در بیشتر از یک خط نشان داده شود باید از " " در ابتدا و انتهای رشته استفاده کرد. مثال :

```
echo this is example یا echo "this is example"
this is example

echo this
is
example
error
echo "this
is
example"
this
is
example
```

Set 1.3.4

با استفاده از این دستور می توان همه متغیرهای تعریف شده در پوسته را مشاهده کرد.

Alias 1.3.5

گاه دستوراتی استفاده می کنیم که بسیار طولانی بوده و خود از چندین دستور دیگر تشکیل می شوند، در این صورت هربار تکرار این دستور طولانی و پیچیده احتمال خطا را بالا برده و همچنین وقت بسیاری را تلف می کند. راه حلی که پوسته در اختیار قرار می دهد به این شکل است که یک دستور طولانی را می توان در قالب یک متغیر محلی ذخیره کرد و هر بار به جای اجرای دستور طولانی، معادل کوتاه شده آن را به کار برد. دستور معادل کوتاه شده را alias (نام مستعار) می نامند و به صورت زیر تعریف می کنند:

```
alias name="command sequence"
```

نکته : در هر دو طرف علامت = نباید هیچ فاصله ای وجود داشته باشد، همچنین وجود " " و یا ' ' در سمت راست تساوی الزامی ست.



مثال : دستور زیر لیست فایل‌های دایرکتوری جاری را با جزییات آنها گرفته و با استفاده از خط لوله به دستور less منتقل می‌کند تا در صفحه‌ای جداگانه نشان داده شود :

```
alias list="ls -l | less"
```

حال این سوال مطرح است که اگر بخواهیم متغیرهای محیطی یا دستورات مستعار را برای همه پوسته‌ها تعریف کنیم تا هر پوسته پس از راه‌اندازی سیستم از این مقادیر آگاهی داشته باشد چگونه و در کجا این مقادیر را تعریف کنیم؟ پاسخ این سوال در بخش‌های بعدی آورده شده است.

نکته : برای اضافه کردن مقادیر جدید به یک متغیر و همچنین حفظ مقادیر قبلی آن باید به شکل زیر عمل کرد .

برای مثال متغیر device تعریف شده و مقدار "pc" در آن ذخیره شده، اگر بخواهیم مقدار "laptop" را نیز به انتهای آن اضافه کنیم :

```
device=$device" laptop"  
echo $device  
pc laptop
```

Unset 1.3.6

در صورتی که بخواهیم یک متغیر تعریف شده مقدارش را از دست داده و از این پس تعریف شده نباشد دستور unset را اجرا می‌کنیم :

```
device="laptop"  
unset device
```

1.4 متغیرهای محیطی تعریف شده

HOME	مسیر دایرکتوری خانه برای کاربر
SHELL	نام پوسته در حال اجرا
IFS	تعیین کننده Internal Field Separator (کاراکتری که به عنوان جدا کننده کلمات در پوسته به کار می‌رود)



LD_LIBRARY_PATH	اولین مسیر جستجوی objectها برای Dynamic Linking* حین اجرای پروسسها
PATH	مسیر جستجوی برنامه ها و دستورات برای اجرا هر مسیر با : از مسیر دیگر تفکیک داده می شود.
PWD	مسیر کنونی (دایرکتوری کنونی)
RANDOM	مقداری تصادفی بین 0 تا 32767 ایجاد می کند.
SHLVL	هر بار که یک پوسته جدید درون پوسته کنونی اجرا شود به مقدار این متغیر یکی اضافه می شود در حالت عادی پس از وارد شدن به سیستم (login) اولین پوسته اجرا شده و مقدار آن 1 است.
TZ	منطقه‌ی زمانی سیستم
UID	شناسه عددی کاربر کنونی

اگر لازم باشد متغیرهای محلی جدید تعریف کنیم و مقدار آنها به صورت خودکار برای هر کاربر تعیین شود، باید متغیر در یکی از فایل های زیر نوشته شود :

- /etc/profile

اسکرپیت نوشته شده در این فایل برای همه کاربران سیستم اجرا می شود، متغیرهای مشترک برای همه کاربران در این فایل تعریف می شوند.

- ~/.profile

پس از /etc/profile این اسکرپیت برای هر کاربر اجرا می شود، مقادیر ویژه‌ی هر کاربر باید در این فایل تعریف شود.

نکته : برای متغیر محیطی که در هر دو فایل تعریف شده و مقدار گرفته باشد، مقدار تعیین شده در ~/.profile در نظر گرفته می شود.



2 اسکریپت نویسی (Script)

به مجموعه ای از دستورات خط فرمان که در یک فایل نوشته شده باشند، اسکریپت گفته می شود. با وجود داشتن پوسته و خط فرمان چه نیازی به اسکریپت نویسی داریم ؟ پاسخ این است که گاه لازم است یک فعالیت تکراری که شامل تعداد زیادی دستور خط فرمان است را برای ورودی های مختلف و بر روی ماشین های مختلف اجرا کنیم.

اسکریپت نویسی نه تنها زمان بسیار کمتری می گیرد بلکه در صورتی که اسکریپت به خوبی نوشته شده باشد کار را با دقتی بسیار بالاتر به انجام می رساند.

3 زبان اسکریپت نویسی

همه پوسته های موجود در Unix به عنوان یک زبان اسکریپت نویسی قابل استفاده هستند. هنگام نوشتن یک اسکریپت می توان از تمام دستورات و امکانات CLI استفاده کرد. همچنین در حاضر اغلب زبان های Python و Perl برای نوشتن اسکریپت به کار می روند. در اینجا روش نوشتن اسکریپت در پوسته توضیح داده می شود. برای جزییات بیشتر به آدرسهای زیر مراجعه کنید.

<http://tldp.org/LDP/abs/html/refcards.html>

<http://www.gnu.org/software/bash/manual/bashref.html>

4 انتخاب پوسته برای اجرای اسکریپت

اسکریپت نوشته شده بایستی یک پوسته از پوسته های نصب شده در سیستم را انتخاب کرده و در آن محیط اجرا شود، در صورتی که این پوسته در اسکریپت ذکر نشده باشد اسکریپت در پوسته ی جاری شروع به کار می کند (پوسته ای که در زمان اجرای اسکریپت فعال باشد).

اولین خط در اسکریپت تعیین کننده پوسته انتخابی است :

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

در این حالت #! اعلام کننده مسیر پوسته مورد نظر برای اجراست، در اینجا پوسته ی bash انتخاب شده که در آدرس /bin/bash/ قرار دارد.



نکته : در برخی موارد لازم است که اسکریپت از هر جای سیستم قابل دسترسی باشد، به این منظور باید آن را در یکی از مسیرهای تعیین شده در PATH اضافه کرد و یا مسیر کنونی اسکریپت را به PATH افزود.

4.1 متغیرها در پوسته (variables)

تعریف و مقداردهی متغیر همانند تعریف متغیر در پوسته است.

4.2 آرگومان (arguments)

همانند توابع برای هر اسکریپت نیز می‌توان از آرگومان‌های ورودی آن استفاده کرد. آرگومان‌ها مقادیری هستند که در رشته فراخوانی اسکریپت آورده می‌شوند، ترتیب دسترسی به آنها نیز به ترتیب وارد شدن آنها است.

آرگومان 0 نام اسکریپت فراخوانی شده است و با مقدار \$0 قابل دسترسی است. بعد از آن به ترتیب شماره‌های بعدی، آرگومان‌های اسکریپت را مشخص می‌کند. برای مثال در اجرای اسکریپت script.sh به صورت زیر، برای استفاده از هر یک از آرگومان‌ها در متن اسکریپت می‌توان از \$i استفاده کرد که در آن i یک عدد ثابت که نماینده شماره آرگومان مورد نظر است می‌باشد. برای مثال \$2 در این دستور برابر world است:

```
./script.sh hello world with arguments
```

متغیرهایی با مقادیر ویژه در پوسته وجود دارند، توضیحات مربوطه در جدول زیر آورده شده اند.

مقدار	متغیر
نام اسکریپت اجرا شده	\$0
مقدار آرگومان اول	\$1
مقدار 10 مین آرگومان در اسکریپت کنونی	\${10}
تعداد کل آرگومان های ورودی	\$#
تعداد کل آرگومان های ورودی	\${#*}
تعداد کل آرگومان های ورودی	\${#@}



"\$*"	تمامی آرگومان های ورودی در یک رشته
"\$@"	
(\$@)	آرایه ای از تمامی آرگومان های ورودی در اسکرپیت کنونی، نام
(\$*)	اسکرپیت در این آرایه قرار ندارد و آرگومان شماره 0 در واقع اولین آرگومان ورودی است.
\$?	مقدار بازگشتی آخرین دستور اجرا شده، اغلب در صورت موفقیت در اجرای دستور این مقدار برابر 0 است
\$\$	شماره پروسس (PID=Process Identifier) اسکرپیت کنونی

مثال : خواندن آرگومان های با شماره فرد در یک اسکرپیت

```
arg_num=$#
arg_value=($*)
for ((i=0;i<arg_num;i+=2));
do
    echo ${arg_value[$i]}
done
```

4.3 عبارت شرطی (if)

▪ if ... then

```
if [ "foo" == "foo" ]; then
    echo expression evaluated as true
fi
```

▪ if ... then ... else

```
#!/bin/bash
if [ "foo" == "foo" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

```
#!/bin/bash
T1="foo"
```



```
T2="bar"
if [ "$T1" == "$T2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

موارد ذکر شده در جدول زیر می توانند به عنوان شرط if قرار بگیرند :

[-e FILE]	True if FILE or DIRECTORY exists.
[-f FILE]	True if FILE exists and is a regular file.
[STRING1 == STRING2]	True if the strings are equal.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if STRING1 sorts before STRING2 lexicographically in the current locale.
[STRING1 > STRING2]	True if "STRING1" sorts after "STRING2" lexicographically in the current locale.

4.4 حلقه

```
for (condition);
do
works to do
done
```

مثال : نمایش نتیجه اجرای دستور ls و چاپ کردن خط به خط آن

```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

مثال : نمایش مقدار همه آرگومان ها

```
for arg in "$@";
do
    echo $arg
done
```



مثال :

```
#!/bin/bash
for i in `seq 1 10`; do
    echo $i
done
```

مثال :

```
C-like for
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
    echo "Welcome $c times"
done
```

```
While :
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

```
Until :
#!/bin/bash
COUNTER=20
until [ $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

5 ابزارهای برنامه نویسی

5.1 کتابخانه Glibc

<http://www.gnu.org/software/libc/index.html>



هر سیستم عامل مشابه Unix (Unix-like) نیاز به کتابخانه ای به زبان C دارد چرا که ساختار های اصلی Unix به زبان C نوشته شده اند. از جمله آنها فراخوانی های سیستمی (System Call) که در ادامه دستور کار به تفصیل از آنها استفاده شده است. برای ایجاد اینترفیس یکسان برای همه یا بیشتر ماشین هایی که دارای یک سیستم عامل مبتنی بر یونیکس هستند، اینترفیس استاندارد با نام POSIX (Portable Operating System Interface) تعریف شده است. POSIX باعث می شود بتوان کدی را بدون تغییرات اساسی در سیستم های مختلف مبتنی بر یونیکس استفاده کرد.

Gnu C Library یا glibc کتابخانه ای استاندارد به زبان C است که توسط بنیاد GNU نگهداری می شود، این کتابخانه با استانداردهای C11 و POSIX.1-2008 سازگاری کامل دارد. در نوشتن کد برنامه های دستور کار تماما از کتابخانه glibc استفاده شده است.

5.2 کامپایلر GCC

کد خود را به زبان C در فایلی نوشته و ذخیره می کنیم، در این مثال نام فایل program.c انتخاب شده است، سپس با دستور زیر برنامه کامپایل خواهد شد.

```
gcc program.c
```

نتیجه فایل اجرایی a.out است، در صورتی که بخواهیم نام فایل اجرایی را خود انتخاب کنیم باید مطابق دستور زیر عمل کنیم.

```
gcc program.c -o app  
app
```

در اینجا نام app برای فایل اجرایی انتخاب شده است.

مراحل کامپایل کردن در gcc به شرح زیر است.

- Preprocessing: در این مرحله، همه کامنت ها حذف شده، ماکروها (دستوراتی که با # شروع می شوند) اجرا شده و محتوای فایل های include شده کپی می شوند

```
gcc -E main.c -o main.i
```

- Compile: در این مرحله، کد سی تبدیل به کد اسمبلی می شود.



```
gcc -S main.c -o main.s
```

- **Assemble:** در این مرحله، کد اسمبلی تبدیل به کد ماشین (فایل object) می‌شود، در نظر داشته باشید که تا این مرحله، کد تولید شده آدرس توابع استفاده شده که در فایل‌های c دیگر هستند را ندارد، امضاها و موجود در فایل‌های h. صرفاً مشخص کننده مکان این آدرس‌ها هستند که در مرحله لینک اضافه خواهند شد

```
gcc -c main.c -o main.o
```

- **Linking:** در این مرحله، فایل‌های آبجکت ساخته شده به هم لینک شده و آدرس‌های مورد نیاز مشخص می‌شوند.

```
gcc main.o -o main.out
```

<code>gcc -c code.c</code>	فایل <code>code.c</code> را به عنوان ورودی گرفته، فایل <code>object</code> با نام <code>code.o</code> را خواهد ساخت
<code>gcc code.c</code>	فایل <code>code.c</code> را به عنوان ورودی گرفته، فایل اجرایی با نام <code>a.out</code> را خواهد ساخت
<code>gcc code.c -o app_name</code>	فایل <code>code.c</code> را به عنوان ورودی گرفته، فایل اجرایی <code>app_name</code> را خواهد ساخت

```
gcc -std=c99 app.c
```

برای مثال در نسخه‌ی اولیه زبان امکان تعریف متغیر در داخل حلقه وجود نداشت و اگر پیش‌فرض کامپایلر، روی نسخه‌های قدیمی زبان باشد آن را به عنوان خطا در نظر می‌گیرد. اما اگر همانند فوق برنامه را کامپایل کنیم، برنامه به درستی کامپایل خواهد شد. برخی از استانداردهای معتبر برای تعیین نسخه‌ی زبان عبارتند از:

```
c99, c11, c14, c17, c18
```



همچنین gcc دارای flag های دیگری است که برخی از آن ها عبارتند از: fno-hosted، fsyntax-only، Wformat-overflow و Wall (مورد استفاده این flag ها را در manual ببینید و آن ها را برای یک برنامه نمونه امتحان کنید).

6 اجرای دستورات خط فرمان در برنامه

کتابخانه stdlib.h تابعی با نام system(char * str) را ارائه می کند که بوسیله آن می توان دستورات خط فرمان را در برنامه به زبان C اجرا کرد. عیب این روش کند بودن آن و همچنین عدم دسترسی به نتیجه اجرای دستور است.

در مثال زیر دستور ls در پوسته اجرا کننده برنامه app اجرا می شود، عیب بزرگ این روش این است که اگر لازم باشد نتایج دستور ls در همین برنامه استفاده شوند، باید ابتدا این مقادیر را در یک فایل ذخیره کرده و از فایل بازخوانی شوند.

```
//app.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    Char* command="ls";
    system (command);
    return 0;
}
```

7 بکارگیری ابزار Make در فرآیند برنامه نویسی

در پروژه های بزرگ کامپایل کردن کل کدها و ساختن فایل اجرایی نهایی نیازمند زدن دستورات تکراری و زیادی برای کامپایل شدن و لینک شدن هستیم. به همین منظور برای جلوگیری از اتلاف وقت و ساده شدن کد نویسی در پروژه ها روش make در سیستم عامل های لینوکس تعبیه شده است. در این روش با ایجاد یک اسکریپت Makefile به سیستم عامل می فهمانیم که کدهای ما چگونه کامپایل شوند. این فایل معمولاً در کنار فایل های پروژه قرار می گیرد.



یکی از مهمترین مزیت‌های make این است که در صورتی که یک فایل از کد تغییر کند توسط دستور make شناسایی شده و تنها آن فایل دوباره کامپایل می‌شود و در نهایت با فایل‌های کامپایل شده‌ی قبلی ترکیب شده و خروجی جدید را تولید می‌کند. این مزیت باعث می‌شود تغییرات کوچک در پروژه‌های بزرگ زمان بسیار کمی برای کامپایل بگیرد. در حالی که کامپایل شدن پروژه‌ی بزرگ از ابتدا گاهی ساعت‌ها زمان می‌برد. یک نمونه از این پروژه‌های بزرگ کرنل سیستم‌عامل لینوکس است.

برای آشنایی با make لازم است با ساختار Makefile آشنا شوید. در حالت پیش فرض پس از اجرای دستور make ، فایلی با نام Makefile در همان مسیر جستجو شده و دستورات داخل آن توسط make اجرا خواهد شد.

هر Makefile می‌تواند از چند قسمت تشکیل شده باشد. هر قسمت از الگوی زیر پیروی میکند:

```
target: dependencies
      commands
```

هر target که به نوعی نام قسمت محسوب می‌شود، پیشنهادهايش به همراه دستوراتش است. در هنگام اجرای دستور make می‌توان target را مشخص نمود تا دستورات آن بخش از Makefile اجرا شود. دستور زیر

```
make t
```

دستورات هدف t در Makefile را اجرا می‌کند.

به طور پیش‌فرض اگر target خاصی هنگام اجرای دستور make تعیین نشود، فرآیند کامپایل از targetی به نام all شروع می‌شود.

در گام بعد برای اجرای هر قسمت ابتدا تمام وابستگی‌های (dependencies) مورد نظر، بررسی می‌گردند. وابستگی‌ها می‌تواند شامل فایل‌های مورد نیاز یا قسمت‌های دیگر Makefile باشد. به طور مثال هدف t در صورتی که نیازمند هدف r باشد دستور make برای اجرای t ابتدا قسمت r را اجرا می‌کند. به این ترتیب می‌توان بین قسمت‌های مختلف Makefile زنجیره درست کرد. اگر تمامی



وابستگی‌ها موجود بودند آن‌گاه برنامه دستورات مربوط به آن هدف را اجرا می‌کند. در قسمت دستورات هر نوع دستور bash می‌تواند اجرا شود. در این قسمت معمولاً با استفاده از کامپایلری مانند gcc فایل‌ها کامپایل می‌شوند و پیام‌ها با echo چاپ می‌گردند.

مثال: فرض کنید دو فایل mylib.c و mylib.h را در اختیار داریم. می‌خواهیم از توابع تعریف شده در این فایل‌ها در فایل main.c استفاده کنیم:

فایل mylib.c :

```
int m_cube(int a){  
    return a*a*a;  
}
```

در نظر داشته باشید، زمانی که در حال برنامه نویسی چند فایل‌هستید، به ازای هر فایل c. شامل توابع مختلف، نیاز به یک فایل header شامل پروتوتایپ‌های آن توابع نیز هست، تا در فایل‌هایی که از آن توابع استفاده می‌کنند include شوند و در نتیجه امضای توابع برای کامپایلر در 3 مرحله اول تعریف شده باشد.

فایل mylib.h :

```
int m_cube(int a);
```

فایل کد اصلی: (main.c)

```
#include <mylib.h>  
#include <stdio.h>  
  
int main()  
{  
    int m;  
    scanf("%d", &m);  
    printf("cube is %d", m_cube(m));  
    return 0;  
}
```



برای کامپایل این پروژه باید دو فایل کد (c)، کامپایل شده و در نهایت با یکدیگر لینک شوند. به همین دلیل Makefile زیر آماده شده است.

```
all: app_name

app_name: mylib.o main.o
    gcc mylib.o main.o -o app_name
mylib.o: mylib.c
    gcc -c mylib.c
main.o: main.c
    gcc -c main.c
clean:
    rm -rf mylib.o main.o app_name
```

در این فایل اگر دستور make زده شود هدف all به صورت پیش فرض اجرا می شود که منجر به انجام تمام هدفهای دیگر غیر از clean می شود. با انجام این دستور فایل های object از دو فایل کد ساخته شده و در نهایت با یکدیگر در قسمت app_name لینک می شوند.

معمولا در فایل های Makefile یک هدف clean نیز ایجاد می شود تا در صورت نیاز فایل های کامپایل شده کامل پاک شوند و محیط برای کامپایل دوباره از صفر، کاملا تمیز گردد.

نکته: اگر نامی غیر از Makefile برای فایل مورد نظر انتخاب شده باشد باید از دستور `make -f file_name` استفاده کرد که در آن `file_name` مسیر دسترسی به فایل مورد نظر ماست. همچنین می توان متغیرهایی در Makefile تعریف کرد، اینکار مشابه تعریف متغیر پوسته انجام می گیرد، برای دسترسی به مقدار متغیر بایستی متغیر را در `$(var)` به کار برد.

```
CC=gcc
$(CC) code.c -o app
```



برای کامنت گذاشتن کافی است که در ابتدای هر کامنت یک علامت # قرار دهیم و اگر کامنت ما چند خط را شامل شود در انتهای هر خط \ نیز قرار می‌دهیم:

```
#this is a comment\  
in two lines
```

نکته : همواره لازم نیست همه دستورات یک Makefile اجرا شوند، می‌توان برای اجرای هر قسمت از دستور make target استفاده کرد که در آن target نام قسمت مورد نظر ماست، در مثال زیر برای اجرای کد قسمت clean کافیست دستور زیر را اجرا کنیم:

```
make clean.
```

در ادامه برخی flag های مفید در دستور make آمده است.

- -d : اطلاعات کافی برای debug کردن در اختیار قرار می‌دهد و آن‌ها را چاپ می‌کند
- -k : به صورت پیش‌فرض عملیات کامپایل بعد از مشاهده اولین خطا متوقف می‌شود اما با قراردادن این علامت فرآیند کامپایل تا آن جایی که امکان دارد ادامه پیدا می‌کند.
- -s : با قراردادن این علامت، دستورات در حال اجرا دیگر چاپ نمی‌شوند.