



دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

دستور کار جلسه دوم

تابستان 1401



فهرست مطالب

2	1 پوسته (Shell)
Error! Bookmark not defined.	Dynamic Linking 2
7	3 اسکریپت نویسی (Script)
8	4 زبان اسکریپت نویسی
8	5 انتخاب پوسته برای اجرای اسکریپت
12	6 ابزارهای برنامه نویسی
15	7 ساخت و استفاده از static library و dynamic library
17	8 اجرای دستورات خط فرمان در برنامه
18	9 بکارگیری ابزار Make در فرآیند برنامه نویسی



1 پوسته (Shell)

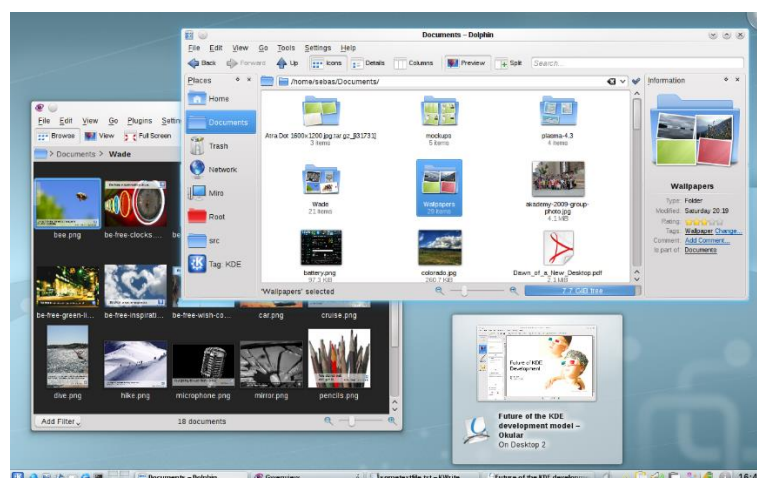
پوسته محیطی است که کاربر اطلاعات خود را در آن وارد می کند، وظیفه پوسته ترجمه ی این دستورات با دستورات سیستمی و در نهایت ارسال آنها به هسته است. پوسته ها به دو دسته تقسیم می شوند، که در ادامه معرفی می شوند.

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 فایل های دایرکتوری جاری را با جزییات آنها گرفته و با استفاده از خط لوله به دستور less منتقل می کند تا در صفحه ای 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
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 برنامه‌ها و دستورات برای اجرا هر مسیر با : از مسیر دیگر تفکیک داده می‌شود.
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.



اسکرپت نویسی نه تنها زمان بسیار کمتری می‌گیرد بلکه در صورتی که اسکرپت به خوبی نوشته شده باشد کار را با دقتی بسیار بالاتر به انجام می‌رساند.

3 زبان اسکرپت نویسی

همه پوسته‌های موجود در Unix به عنوان یک زبان اسکرپت‌نویسی قابل استفاده هستند. هنگام نوشتن یک اسکرپت می‌توان از تمام دستورات و امکانات CLI استفاده کرد. همچنین در حاضر اغلب زبان های Python و Perl برای نوشتن اسکرپت به کار می‌روند. در اینجا روش نوشتن اسکرپت در پوسته توضیح داده می‌شود. 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 اسکرپت

اسکرپت 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 اسکرپت ذکر نشده باشد اسکرپت در پوسته ی جاری شروع به کار می‌کند (پوسته ای که در زمان اجرای اسکرپت فعال باشد).

اولین خط در اسکرپت تعیین کننده پوسته انتخابی است :

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

در این حالت `#!/bin/bash` 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 اسکرپت 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 اسکرپت 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.

متغیر	مقدار
\$0	نام اسکریپت اجرا شده
\$1	مقدار آرگومان اول
\${10}	مقدار 10 مین آرگومان در اسکریپت کنونی
\$#	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 آرگومان‌های ورودی در اسکریپت کنونی، نام اسکریپت 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) اسکریپت کنونی



مثال : خواندن آرگومان‌های با شماره فرد در یک اسکریپت

```
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 :

[-a FILE]	True if FILE exists.
[-e FILE]	True if FILE 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 آرگومان‌ها

```
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
```

تابع 4.5

```
#!/bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

5 ابزارهای برنامه نویسی

5.1 کتابخانه Glibc

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

هر سیستم عامل مشابه (Unix-like) Unix نیاز به کتابخانه ای به زبان C دارد چرا که ساختارهای 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

مراحل کامپایل کردن در gcc به شرح زیر است.

- Parser: بررسی token ها و تحلیل معنایی (Semantic Analysis)
- ایجاد کد میانی و بهینه سازی: کد تولید شده در این مرحله بسیار سطح پایین است ولی همچنان به زبان ماشین ترجمه نشده است)
- Assembler: ایجاد فایل های object با پسوند o
- Linker: ادغام فایل های object در یکدیگر و در نهایت تهیه یک فایل اجرایی (executable) و یا یک Dynamic Library، فایل تولید شده در این مرحله به صورت پیش فرض نام a.out را خواهد داشت.

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

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  
a.out
```

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.

```
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
```

برای ساخت فایل obj از یک فایل c باید به شکل زیر عمل کرد.

```
gcc myapp.c -c
```

با گذاشتن این flag درواقع به کامپایلر دستور می‌دهیم که برنامه ما را به linker تحویل ندهد و عملیات لینک را روی آن انجام ندهد ولی تمام مراحل قبل (شامل pre-processing، compile و assembling) را اجرا کرده و فایل خروجی با پسوند o. ایجاد کند. بدین ترتیب خروجی این دستور، فایل قابل اجرا نخواهد بود.

غیر از خروجی obj که با پسوند o. مشخص می‌شود می‌توان خروجی‌های دیگری هم از کامپایلر گرفت. برای مثال با آپشن -E می‌توان فایل با پسوند i. ساخت که این فایل خروجی preprocessor است. همچنین با استفاده از -S می‌توان کد اسمبلی تولید کرد.

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



6 ساخت و استفاده از static library و dynamic library

همه کتابخانه‌های استاندارد در زبان c دارای پیشوند lib هستند که هنگام استفاده معمولاً به عنوان flag در gcc ذکر نمی‌شوند. اما در موارد دیگر، نیاز است کتابخانه در دستور کامپایل معرفی شود. نحوه ایجاد کتابخانه‌ها یا کامپایل با کتابخانه‌های مشخص در ادامه شرح داده می‌شود.

6.1 Static library

پسوند این دسته از فایل‌ها a. است و تنها در هنگام کامپایل نیاز می‌شوند. این فایل از فایل‌های object ساخته می‌شود. بنابراین باید فایل‌های خود را به فایل obj تبدیل کنیم:

```
gcc -c staticlib.c -o staticlib.o
```

حال کافی است که این فایل‌ها را در یک فایل با پسوند a. ذخیره کنیم. برای این کار از دستور ar استفاده می‌کنیم که معمولاً برای فشرده‌سازی، تصحیح و استخراج فایل‌ها از آن استفاده می‌شود:

```
ar -r libstaticLib.a libstaticLib.o
```

حال فرض کنید کدی به اسم myapp.c داریم که از توابع موجود در این static library استفاده کرده است. برای اینکه این کتابخانه را به صورت استاتیک به کد خود لینک کنیم به صورت زیر کدمان را کامپایل می‌کنیم:

```
gcc myapp.c -L./ -lstaticLib -o app.out
```

دقت کنید که flagهای -l و -L باید بعد از اسم برنامه نوشته شوند. زیرا ابتدا باید برنامه شما آماده باشد تا لینکر بتواند کتابخانه‌های مورد نیاز را تشخیص دهد و به آن لینک کند. در ادامه توضیح کوتاهی در مورد هر کدام بیان می‌شود.



-llibrary: قراردادن این عبارت باعث می‌شود که لینکر تمام مسیرهای استاندارد (برای مثال `/usr/lib/`) را برای پیدا کردن کتابخانه‌ای به اسم `liblibrary.a` جستجو کند و سپس برنامه را با آن لینک کند. برای مثال در دستور زیر:

```
gcc myapp -lpthread
```

لینکر به دنبال فایلی با نام `libpthread.a` می‌گردد.

-Lpath: مسیر داده‌شده را به لیست مسیرهایی که `-l` در آن جستجو می‌کند اضافه می‌کند.

6.2 Dynamic library

این دسته از کتابخانه‌ها که عموماً به اسم `shared object` نیز شناخته می‌شوند، دارای پسوند `.so` هستند. این کتابخانه‌ها هم از فایل‌های `obj` ساخته می‌شوند. بنابراین ابتدا باید فایل‌های `obj` موردنیاز خود را تولید کرد:

```
gcc -c -fPIC dynamicLib.c -o dynamicLib.o
```

استفاده از `fPIC` باعث می‌شود که کد حاصل مستقل از آدرس و مکان باشد. یعنی از آنجا که برنامه به برنامه‌های مختلف لینک می‌شود نمی‌توان آدرس دقیق سمبل‌ها را تعیین کرد (اصلی‌ترین تفاوت کتابخانه استاتیک و پویا). به همین دلیل با ذکر این `flag` این محدودیت را نادیده می‌گیریم تا خود سیستم عامل هنگام بارگذاری برنامه اصلی این آدرس‌ها را تعیین کند. حال برای تولید کتابخانه پویا به شکل زیر عمل می‌کنیم:

```
gcc -shared dynamicLib.o -o dynamicLib.so
```

اکنون کتابخانه پویای ما آماده شده‌است. حال فرض کنید که کدی به اسم `myapp.c` داریم که از توابع موجود در `dynamicLib` استفاده کرده است. برای اینکه به صورت پویا این کتابخانه را به کد خود لینک کنیم به شکل زیر عمل می‌کنیم:

```
gcc myapp.c -L./ -ldynamicLib -o app.out
```



با اجرای دستور فوق، فایل اجرایی شما تولید می‌شود. اما شاید هنگام اجرای آن به خطا برخورد کنید و نتوانید آن را اجرا کنید. این خطا احتمالاً به این دلیل است که loader نمی‌تواند کتابخانه پویا را پیدا کند و آن را بارگذاری کند. معمولاً زمانی این مشکل پیش می‌آید که فایل اشتراکی (کتابخانه پویا) در مسیرهای استاندارد قرار ندارد. بنابراین یا کتابخانه پویا را در یک مسیر استاندارد قرار دهید و یا اینکه مسیر موردنظر را به متغیر LD_LIBRARY_PATH اضافه کنید:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH";path_to_shared_object"  
export LD_LIBRARY_PATH
```

7 اجرای دستورات خط فرمان در برنامه

کتابخانه stdlib.h تابعی با نام `system(char * str)` را ارائه می‌کند که بوسیله آن می‌توان 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 می‌شود، 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;  
}
```



8 بکارگیری ابزار Make در فرآیند برنامه نویسی

در پروژه‌های 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 پروژه‌ها روش make در سیستم‌عامل‌های لینوکس تعبیه شده است. در این روش با ایجاد یک اسکریپت Makefile به سیستم‌عامل می‌فهمانیم که کدهای ما چگونه کامپایل شوند. این فایل معمولاً در کنار فایل‌های پروژه قرار می‌گیرد.

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;  
}
```

فایل mylib. :

```
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` 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
```

متغیرهای پیش فرضی نیز در فایل Makefile تعریف شده اند که برخی از آن ها عبارتند از:

\$@	file name of the target of the rule
\$<	name of the first prerequisite
^	names of all the prerequisites, with spaces between them
+	similar to ^ but include duplicates

از [اینجا](#) می توانید اطلاعات بیشتر درباره متغیرهای پیش فرض بدست آورید.



توصیه می‌شود که حتماً تا جایی که امکان دارد از تعریف متغیر استفاده کنید. زیرا در این صورت اگر نیاز به تغییر نام بعضی فایل‌ها باشد یا لازم باشد فرآیند کامپایل را تغییر دهید، به راحتی می‌توان با تغییر دادن مقدار متغیر مربوطه آن تغییر را در کل فایل اعمال کرد.

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 کافیهست دستور زیر را اجرا کنیم:

```
make clean
```

یکی از موارد مهمی که در Makefile ها وجود دارد دستورات و مقادیر implicit در آن است که کار را بسیار ساده می‌کند و این فایل‌ها را خلاصه می‌سازد.

در این فایل‌ها دو پارامتر CC و CFLAGS به صورت پیشفرض مقادیری دارند که در دستورات implicit استفاده می‌شود CC مقدار GCC را دارد. دو تا از مهمترین دستورات implicit در زیر آمده است:

- در صورتی که برای target ابجکتی فایل‌های .o نیاز باشد، ساخته شدن فایل .o به صورت اتوماتیک از فایل .c با کامپایلر CC رخ می‌دهد.
- در صورتی که یک target نهایی (بدون پسوند) نیاز باشد با کامپایلر CC و .o target همانم به صورت خودکار ساخته می‌شود.

برای جزییات بیشتر می‌توان با کلید واژه‌ی Fancy Rules با این قوانین آشنا شوید.

```
objects = mylib.o main.o  
all: main  
main: $(objects)  
  
clean:  
rm -f *.o main
```



در این مثال به صورت خودکار o. فایل ها از فایل های c ساخته می شود و در نهایت main از دو object فایل به صورت خودکار لینک می شود.

در صورتی که بخواهیم متغیر objects را به صراحت تعریف نکنیم و تمام فایل های c. کامپایل شوند و در نهایت با فایل اصلی main.c لینک شوند می توانیم از Makefile زیر بهره برداری کنیم:

```
src = $(wildcard *.c)
objects = $(src:.c=.o)

all: main
main: $(objects)

clean:
rm -f *.o main
```

در این مثال با دستور wildcard تمام فایل های c. انتخاب می شوند و سپس با جایگزین کردن o. به جای c. در متغیر objects ذخیره می شوند و ادامه ی فایل مانند مثال پیش است.

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

- -d : اطلاعات کافی برای debug کردن در اختیار قرار می دهد و آن ها را چاپ می کند
- -k : به صورت پیش فرض عملیات کامپایل بعد از مشاهده اولین خطا متوقف می شود اما با قراردادن این علامت فرآیند کامپایل تا آن جایی که امکان دارد ادامه پیدا می کند.
- -s : با قراردادن این علامت، دستورات در حال اجرا دیگر چاپ نمی شوند.