دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

علی فانیان

زینب زالی

تابستان ۹۸

دستور کار جلسه دوم

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.

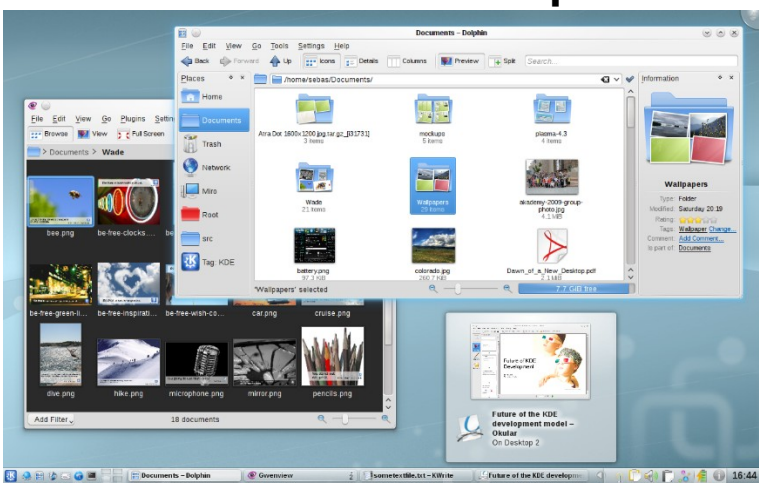
پوسته ها به دو دسته تقسیم می شوند :

- 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 مطرح‌ترین رابط‌های گرافیکی موجود را نمایش می‌دهد.

Gnome 3.1

KDE Plasma Desktop



- # 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.
۴ پوسته مشهور عبارتند از :

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

متغیر های محیطی (Environmental Variables)

هر پوسته قبل از اجرا شدن تعدادی متغیر محیطی را از فایل های پیکربندی می خواند، این متغیرها برای تمامی پروسس‌های ساخته شده در آن پوسته و دستوراتی که در آن اجرا می‌شود قابل دسترسی می‌باشند (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 می‌توان متغیر های 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 پروسس‌هایی که در پوسته ساخته می شوند به متغیرهای پوسته دسترسی ندارند.

- **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 می گیرد ولی خود قابلیت تفکیک اعداد و مقادیر حسابی از رشته ها 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**

محدوده متغیرهای محیطی که در حالت قبل تعریف می شوند در یک پوسته است و پوسته های اجرا شده در پوسته کنونی از مقدار آنها بی اطلاعند. اگر بخواهیم متغیری محیطی تعریف کنیم که در پوسته‌هایی که از این پس اجرا می شوند نیز قابل دسترسی باشند :

```
export variable_name=value
```

با اضافه کردن export متغیر به عنوان متغیری محیطی شناخته می شود و به پروسس‌هایی که در این پوسته ساخته می شوند نیز به ارث می رسد.

- **echo**

برای نمایش مقدار یک متغیر به کار می رود :

```
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**

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

- **alias**

گاه دستوراتی استفاده می کنیم که بسیار طولانی بوده و خود از چندین دستور دیگر تشکیل می شوند، در این‌صورت 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 ومقدار “pc” در آن ذخیره شده، اگر بخواهیم مقدار " laptop" را نیز به انتهای آن اضافه کنیم :

```
device=$device” laptop”
echo $device
pc laptop
```

- **unset**

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

```
deivce=”laptop”
unset device
```

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 می رود)
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 ۰ تا ۳۲۷۶۷ 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 ۱ است.
TZ	منطقه‌ی زمانی سیستم

شناسه عددی کاربر کنونی	**UID**

Dynamic Linking:

اغلب به هنگام استفاده از کتابخانه های بزرگ برای برنامه نویسی، کتابخانه به عنوان قسمتی از کد برنامه استفاده می شود ولی همراه با آن کامپایل نخواهد شد، در چنین حالتی linking در حین اجرای برنامه اتفاق می افتد به این طریق که کتابخانه به صورت مجموعه ای از object ها در کنار برنامه قرار گرفته و مسیر آن در کد اصلی برنامه ذکر می شود. این مسیر اغلب به عنوان متغیری محلی تعریف می شود.

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

اسکریپت 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 اسکریپت برای هر کاربر اجرا می‌شود، 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 می‌شود.

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 ورودی‌های مختلف و بر روی ماشین‌های 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.

زبان اسکریپت نویسی

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

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.

- **متغیر ها در پوسته (variables)**
  تعریف و مقداردهی متغیر همانند تعریف متغیر در پوسته است.
- **آرگومان (arguments)**
  همانند توابع برای هر اسکریپت نیز می‌توان از آرگومان‌های ورودی آن استفاده کرد. آرگومان ها مقادیری هستند که در رشته فراخوانی اسکریپت آورده می شوند، ترتیب دسترسی به آنها نیز به ترتیب وارد شدن آنها می باشد .

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 $۲ 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 ۱۰ 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 ۰ در واقع اولین آرگومان ورودی می باشد
$?	مقدار بازگشتی آخرین دستور اجراشده، اغلب در صورت موفقیت در اجرای دستور این مقدار برابر ۰ است
$$	شماره پروسس (PID=Process IDentifier) اسکریپت کنونی

مثال :
خواندن آرگومان‌های با شماره فرد در یک اسکریپت

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

- **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.

- **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**

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

ابزارهای برنامه نویسی

▪ 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 شده‌اند. 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 ماشین‌هایی که دارای یک سیستم عامل مبتنی بر یونیکس هستند، اینترفیس استانداردی با نام POSIX (Portable Operating System Interface) تعریف شده است. POSIX باعث می‌شود بتوان کدی را بدون تغییرات اساسی در سیستم‌های 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.

▪ gcc

مراحل کامپایل کردن در gcc :

۱. Parser :

بررسی token ها و تحلیل معنایی (Semantic Analysis)

۲. ایجاد کد میانی و بهینه سازی (کد تولید شده در این مرحله بسیار سطح پایین است ولی همچنان به زبان ماشین ترجمه نشده است)

3. Assembler :

ایجاد فایل های object با پسوند o.

۴. 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 نسخه‌ای 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 نسخه ی 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-only، fno-hosted، Wformat-overflow و Wall (مورد استفاده این flag ها را در manual ببینید و آن‌ها را برای یک برنامه نمونه امتحان کنید).

ساخت و استفاده از dynamic library و static 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.

- **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 پیداکردن 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.

- **Dynamic library**

این دسته از کتابخانه‌ها که عموما 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 می کنیم:

```
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
```

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;
}
```

بکارگیری ابزار Make در فرآیند برنامه نویسی

برای کامپایل کردن تعداد محدودی از فایل های object، کد برنامه و تعیین آدرس کتابخانه‌های پویا لازم است هر بار دستورات لازم برای کامپایل را وارد کنیم که این کار احتمال خطای بالا و اتلاف وقت به همراه دارد. برای

اجتناب از این وضعیت می‌توان دستورات لازم را در یک فایل نوشته و از ابزاری به نام make بهره برد. این کار نوعی اسکریپت نویسی است با این تفاوت که نتیجه اجرای دستورات به کامپایل یک برنامه منتهی می شود.

از مزایای make بررسی تغییرات ایجاد شده در فایل هاست، به این معنی که اگر در یکی از فایل هایی که در آن ذکر شده اند تغییری ایجاد شده باشد، نیازی به کامپایل همه منابع نیست و فقط فایل تغییر یافته مجددا کامپایل خواهد شد.

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 از چند قسمت تشکیل شده که به ترتیب تعیین شده، دستورات هر قسمت اجرا می شوند :

```
[target]: dependencies
<tab>commands
```

به طور پیش‌فرض اگر target خاصی هنگام اجرای دستور تعیین نشود، فرآیند کامپایل از targetی به نام all شروع می‌شود: ابتدا تمام وابستگی‌های (dependencies) هدف (target) مورد نظر بررسی می‌گردند. اگر تمامی وابستگی‌ها موجود بودند آن‌گاه برنامه دستورات مربوط به آن هدف را اجرا می‌کند. ولی اگر تعدادی از وابستگی‌ها موجود نباشند (یا تغییر کرده باشند) آنگاه فرآیند کامپایل ابتدا به هدفی همنام با همان وابستگی منتقل می‌شود و پس از ساخت آن دوباره به همین نقطه برمی‌گردد و ادامه می‌دهد.

مثال :

```
vim Makefile
#example:
all: app_name

app_name: code1.o code2.o
        clang   code1.o         code2.o         app_name
code1.o: code1.c
        clang –c         code1.c
code2.o: code2.c
        clang –c code2.c
clean:
        rm –rf *o code1.o code2.o app_name
```

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=clang
$(CC) code.c –o app
```

متغیرهای پیش‌فرضی نیز در فایل Makefile تعریف شده‌اند که برخی از آن‌ها عبارتند از:

```
$@
$*
$<
$^
            spaces, discard duplicates
$+      similar to $^ but include duplicates
```

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 تغییردادن 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، می توان 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 flagهای 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.