جلسه‌ی چهارم — نوشتن و ترجمه‌ی برنامه‌ها

در این جلسه با امکانات محیط یونیکس برای نوشتن برنامه‌ها و ساختن فایل اجرایی از آنها آشنا خواهید شد.

ترجمه‌ی برنامه‌ها در محیط یونیکس

محیط یونیکس ابزارهای زیادی را برای نوشتن، ترجمه و مدیریت کد برنامه‌ها در اختیار برنامه‌نویسان قرار می‌دهد. پس از نوشتن برنامه‌ها، می‌توان با استفاده از یکی از مترجم‌های موجود در محیط یونیکس برنامه‌ها را ترجمه نمود.

```
$ cat test.c
#include <stdio.h>
int main(void)
{
     printf("Hello world!\n");
     return 0;
}
$ cc test.c              # the resulting binary is a.out
$ ./a.out                # executing ./a.out
Hello world!
$ cc -o test test.c      # specifying the output file
$ ./test
Hello world!
```

برای ترجمه‌ی برنامه‌هایی که در زبان C++ نوشته شده‌اند نیز می‌توان از دستور c++ استفاده نمود. در بیشتر توزیع‌های لینوکس، معمولا به صورت پیش‌فرض از مترجم GCC[1] برای ترجمه‌ی برنامه‌ها استفاده می‌شود و معمولا دستور cc معادل دستور gcc و c++ معادل دستور g++ می‌باشد.

در صورتی که تعداد فایل‌های کد برنامه زیاد باشد (یا در صورتی که زبان‌های متفاوتی استفاده شده باشند)، عمل ترجمه و لینک[2] در دو گام مجزا می‌توانند انجام شوند:

```
$ ls
src1.c src2.c src3.c
```

۱ GNU Compiler Collection
۲ Linking

```
$ cc -c src1.c          # create an object file for src1.c
$ cc -c src2.c          # create an object file for src2.c
$ cc -c src3.c          # create an object file for src3.c
# link the objects to create the binary file
$ cc -o out src1.o src2.o src3.o
$ ./out                 # execute the resulting binary
```

ساختن خودکار فایل‌های خروجی

یک راه برای ترجمه‌ی خودکار تعداد زیادی فایل، نوشتن اسکریپتی برای این کار می‌باشد. اما راه بهتر استفاده از ابزار make است که با گرفتن رابطه بین فایل‌ها و دستورهای لازم برای تولید فایل‌های خروجی، فایل‌ها را فقط در صورت نیاز ترجمه می‌کند. برنامه‌ی make با خواندن یک فایل ورودی (که معمولا Makefile یا makefile نامیده می‌شود)، دستورهای لازم برای تولید یک فایل خروجی را یکی پس از دیگری اجرا می‌کند. فایل ورودی make به ازای خروجی‌های میانی و نهایی، پیش‌نیازها و دستورهایی لازم برای تولید آنها را مشخص می‌کند. در مثال زیر، الگوی کلی این فایل نمایش داده شده است: برای ساختن فایل «target» فایل‌هایی که پس از آن مشخص می‌شوند («prereq1» و «prereq2») باید ساخته شوند و برای ساختن این فایل، دستورهایی که در خط‌های بعد مشخص شده‌اند (دستورهای «command 1» و «command 2») اجرا می‌شوند.

```
target: [prereq1] [prereq2] ...
     [command 1]
     [command 2]
     ...
```

با دستور «make target» در پوسته، فایل target تنها وقتی ساخته می‌شود که این فایل وجود نداشته باشد یا موجود باشد و حداقل یکی از پیش‌نیازهای آن جدیدتر از آن باشند. در مثال بخش قبل برای ساخت out، در صورتی که پس از دستورهای گفته شده فایل src3.c تغییر کند، ترجمه‌ی دوباره‌ی src1.c و src2.c لازم نیست. یک Makefile نمونه برای ساختن این فایل در ادامه نشان داده می‌شود:

```
$ cat Makefile
src1.o: src1.c              # src1.o depends on src1.c
     cc -c src1.c           # command to create src1.o
src2.o: src2.c
```

```
	cc -c src1.c
src3.o: src3.c
	cc -c src1.c
out: src1.o src2.o src3.o   # out depends on src[123].o
	cc -o out src1.o src2.o src3.o
```

سپس با استفاده از دستور make می‌توان فایل‌های مشخص شده در Makefile را با استفاده از دستورات معرفی شده به صورت خودکار ساخت.

```
$ make out                  # build <out> target
cc -c src1.c
cc -c src2.c
cc -c src3.c
cc -o out src1.o src2.o src3.o
```

اگر یکی از فایل‌های پیش‌نیاز تغییر کند، ساختن دوباره‌ی همه‌ی فایل‌ها لازم نیست؛ در مثال زیر، فایل src3.c با استفاده از دستور touch (که زمان تغییر یک فایل را به روز می‌رساند) تغییر داده می‌شود.

```
$ touch src3.c              # modify src3.c
$ make out                  # only src.o and out are replaced
cc -c src3.c
cc -o out src1.o src2.o src3.o
```

در Makefile-ها می‌توان متغیر تعریف نمود و دستورات و پیش‌نیازهای فایل‌هایی که به صورت مشابه شاخته می‌شوند را به صورت خلاصه‌تری بیان نمود. برای جزئیات بیشتر، به مستنداتی که با عمق بیشتری به ابزار make می‌پردازند مراجعه شود.

متغیرهای محیطی

به هر پردازه در یونیکس، از جمله پوسته، تعدادی متغیر محیطی[1] اختصاص می‌یابد. این متغیرهای محیطی پس از فراخوانی سیستمی fork() در پردازه‌ی فرزند باقی می‌مانند و از این رو برای انتقال داده‌های رشته‌ای کوتاه به پردازه‌ها استفاده می‌شوند. متغیرهای محیطی پوسته را می‌توان به صورت زیر تعریف کرد یا مقدار آنها را خواند.

```
$ export MYENV="my env"    # define an environment variable
```

۱ Environment Variable

```
$ echo $MYENV              # the value of an environment variable
my env
$ env                      # the list of environment variables
...
MYENV=my env
...
```

یکی از متغیرهای محیطی مهم در یونیکس، متغیر PATH می‌باشد. این متغیر، فهرستی از شاخه-هایی که حاوی فایل‌های اجرایی هستند و با علامت «:» جدا می‌شوند را در خود نگه می‌دارد. برای اجرای فایل‌هایی که در این شاخه‌ها قرار دارند، مشخص کردن آدرس آنها لازم نیست و برخی از توابع کتابخانه‌ای خانواده‌ی exec برای اجرای برنامه‌ها (مثل execlp()) در این شاخه‌ها نام فایل اجرایی داده شده را جستجو می‌کنند.

```
$ echo $PATH
/sbin:/usr/sbin:/bin:/usr/bin
$ ls /bin        # /bin contains the <uname> executable file
...
uname
...
$ uname          # thus <uname> is equivalent to </bin/uname>
Linux
```

با استفاده از تابع getenv() در کتابخانه‌ی استاندارد زبان C می‌توان مقدار یک متغیر محیطی را در زبان C خواند. این تابع که در فایل «stdlib.h» معرفی می‌شود، در صورتی که متغیر محیطی داده شده تعریف نشده باشد، مقدار NULL را بر می‌گرداند؛ برای جزئیات بیشتر به getenv(3) (یعنی دستور «man 3 getenv») مراجعه شود.

تمرین چهارم

در شاخه‌ی ~/ex4 برنامه‌ای به نام procinfo.c بنویسید که پیغامی به صورت زیر چاپ کند:

```
user    me
home    /home/me
pid     111
uid     1001
path    /home/me/ex4
```

در این مثال، عبارت پس از user مقدار متغیر محیطی USER، عبارت بعد از home مقدار متغیر محیطی HOME، مقدار پس از pid شماره‌ی پردازه‌ی ایجاد شده (فراخوانی سیستمی ()getpid)، مقدار پس از uid شماره‌ی کاربری (فراخوانی سیستمی ()getuid) و مقدار پس از عبارت path شاخه‌ی جاری پردازه (فراخوانی سیستمی ()getcwd) هستند. برای استفاده از این فراخوانی‌های سیستمی، صفحه‌ی راهنمای‌های آنها را مشاهده نمایید. همچنین یک Makefile برای ترجمه‌ی این فایل توسط دستور make بنویسید.