دانشگاه صنعتی نوشیروانی بابل

دستور کار پیشنهادی

آزمایشگاه سیستم عامل

زمستان ۱۳۹۵

فهرست مطالب

مقدمه ۱
جلسه‌ی اول — آشنایی با پوسته ۲
جلسه‌ی دوم — ورودی و خروجی در پوسته ۷
جلسه‌ی سوم — اسکریپت‌های پوسته ۱۱
جلسه‌ی چهارم — مروری بر پوسته ۱۷
جلسه‌ی پنجم — نوشتن و ترجمه‌ی برنامه‌ها ۲۱
جلسه‌ی ششم — مدیریت پردازه‌ها ۲۷
جلسه‌ی هفتم — انتقال داده با لوله ۳۰
جلسه‌ی هشتم — کتابخانه‌ی PThreads ۳۵
جلسه‌ی نهم — سیگنال‌ها ۳۹

مقدمه

سیستم عامل یونیکس تأثیر بسیار مهمی بر سیستم‌های عامل پس از آن و رابطهای مطرح شده در استاندارد POSIX[1] گذاشته است. در درس آزمایشگاه سیستم عامل، برخی از این رابطها معرفی می‌گردند تا از یک سو برخی از مفاهیم معرفی شده در درس سیستم عامل به صورت تجربی مشاهده گردند و از سوی دیگر توانایی توسعه‌ی نرم‌افزار و مدیریت در سیستم‌های عاملی که از استاندارد POSIX تبعیت می‌کنند، در دانشجویان ایجاد گردد.

اهداف اصلی این درس موارد زیر هستند:

- استفاده از پوسته[2] برای مدیریت فایل‌ها و دسترسی‌ها
- پردازش متن[3] با استفاده از دستورات یونیکس و ترکیب آنها با لوله[4]
- استفاده از رابط برنامه‌نویسی برای مدیریت پردازه‌ها
- استفاده از کتابخانه‌ی PThreads برای همروندی و مدیریت دسترسی‌های همزمان در آن
- شکستن یک برنامه به چند پردازه یا ریسمان و انتقال اطلاعات بین آنها
- استفاده از سیگنال و لوله برای تعامل بین پردازه‌ها با استفاده از رابط برنامه‌نویسی

محدودیت زمان در این آزمایشگاه موجب شده است که مطالب هر جلسه به محدودترین و ساده‌ترین شکل ممکن ارائه گردند. با این وجود، سعی شده است محتوای جلسه‌ها برای انتقال مفاهیم مورد نظر و یافتن جزئیات بیشتر در مورد مطالب مطرح شده کافی باشد.

ارزشیابی

نمره‌ی اختصاص یافته به هر دانشجو با توجه به گزارش تمرین‌ها، یک امتحان در پایان ترم و در صورت لزوم یک تمرین نهایی خارج از آزمایشگاه می‌تواند تعیین گردد.

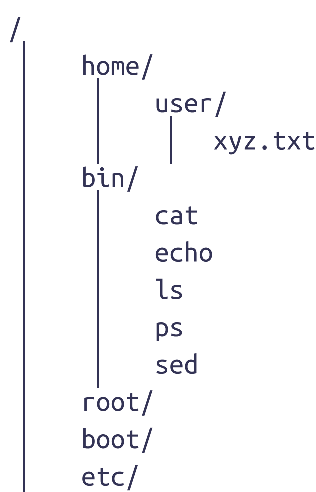
1 The Portable Operating System Interface
2 Shell
3 Text Processing
4 Pipe

جلسه‌ی اول — آشنایی با پوسته

در این جلسه با مقدمات استفاده از رابط پوسته[1] برای مدیریت فایل‌ها در محیط‌های مشابه یونیکس[2] آشنا خواهید شد.

ساختار فایل سیستم

فایل سیستم در یونیکس یک ساختار درخت دارد. برخی از رأس‌های این درخت، از جمله رأس‌های میانی آن شاخه[3] هستند. این درخت از شاخه‌ی ریشه (که با علامت «/» نشان داده می‌شود) شروع می‌شود. شکل زیر ساختار درختی یک فایل سیستم نمونه را نشان می‌دهد.



با پیمودن مسیر شروع شده از ریشه به فایل‌ها و شاخه‌ها در این درخت، آدرس آنها تعیین می‌شود: شاخه‌های این مسیر از ریشه از چپ به راست کنار هم قرار داده می‌شوند و با علامت «/» جدا می‌گردند. برای نمونه، آدرس فایل «ls» در شکل روبرو «/bin/ls» می‌باشد. هر شاخه در فایل سیستم دو زیر شاخه‌ی مجازی دارد: «.» به همان شاخه و «..» به شاخه‌ی بالاتر از آن شاخه اشاره می‌کند. بنابراین دو آدرس «/root/» و «/bin/../root/./» به یک شاخه اشاره می‌کنند.

به هر پردازه (از جمله پوسته) در سیستم عامل شاخه‌ای به نام شاخه‌ی جاری[4] اختصاص داده می‌شود؛ این شاخه، آدرسی که پردازه در آن در حال اجرا است را مشخص می‌کند. با توجه به این شاخه، آدرس فایل‌ها و شاخه‌ها به صورت نسبی بیان می‌گردد؛ آدرس‌های نسبی آدرس‌هایی هستند که با «/» شروع نمی‌شوند. برای یافتن مقصد این آدرس‌ها، آدرس شاخه‌ی جاری به ابتدای آنها اضافه می‌گردد. به عنوان مثال، در صورتی که شاخه‌ی جاری «/home/user/» باشد، آدرس نسبی «../» به شاخه‌ی «/home/» اشاره می‌کند و آدرس نسبی «xyz.txt» به فایل «/home/user/xyz.txt» اشاره می‌کند.

1 Shell
2 Unix-like
3 Directory
4 Current working directory

اجرای دستورات در پوسته

اصلی‌ترین رابط کاربر در یونیکس برنامه‌ای به نام پوسته می‌باشد. پوسته دستورات را یکی پس از دیگری از ورودی می‌خواند و اجرا می‌کند. برخی از دستورات ابتدایی پوسته و محیط یونیکس در ادامه نمایش داده می‌شوند.

```
$ pwd                    # آدرس شاخه‌ی جاری پوسته را چاپ می‌کند
$ cd path                # شاخه‌ی جاری پوسته را به «path» تغییر می‌دهد
$ cd ~                   # شاخه‌ی جاری پوسته را به شاخه‌ی خانه تغییر می‌دهد
$ cd                     # معادل دستور قبلی
$ ls                     # همه‌ی فایل‌ها و شاخه‌های شاخه‌ی جاری را فهرست می‌کند
$ find -name "pat"       # جستجوی همه‌ی فایل‌هایی که نامشان با الگوی «pat» مطابقت می‌کند
$ find path -name "pat"  # مشابه دستور قبل برای شاخه‌ی «path»
$ mkdir XYZ              # شاخه‌ای با نام «XYZ» ایجاد می‌کند
$ rmdir XYZ              # شاخه‌ی «XYZ» را حذف می‌کند؛ شاخه باید خالی باشد
$ rm XYZ                 # فایل «XYZ» را حذف می‌کند
$ rm -r XYZ              # به صورت بازگشتی فایل یا شاخه‌ی «XYZ» را حذف می‌کند
$ cp src dir1/           # فایل مبدأ (پارامتر اول) را به شاخه‌ی مقصد (پارامتر دوم) کپی می‌کند
$ cp -r dir1/ dir2/      # فایل یا شاخه‌ی مبدأ را به صورت بازگشتی کپی می‌کند
$ mv dir1 dir3           # فایل یا شاخه‌ی اول را به شاخه‌ی دوم انتقال می‌دهد
$ echo "Hello!"          # عبارت «Hello!» را چاپ می‌کند
$ cat TEST               # محتویات فایل «TEST» را چاپ می‌کند
$ passwd                 # گذرواژه‌ی کاربر را عوض می‌کند
$ ls --help              # برخی از امکانات دستور «ls» را فهرست می‌کند
$ man ls                 # صفحه‌ی راهنمای دستور داده شده را نشان می‌دهد
$ date                   # زمان و تاریخ را چاپ می‌کند
$ ps aux                 # فهرست همه‌ی پردازه‌ها را نشان می‌دهد
$ pstree                 # درخت پردازه‌ها را نشان می‌دهد
$ sleep 5                # پنج ثانیه متوقف می‌ماند
```

دستورات را در پوسته می‌توان به شکل‌های گوناگونی ترکیب نمود. در ادامه چند مثال برای ترکیب دستورها نشان داده می‌شود.

```
$ cmd1; cmd2       # الف) ابتدا دستور «cmd1» و سپس «cmd2» را اجرا می‌کند.
$ cmd2 `cmd1`      # ب) عبارت «`cmd1`» را با خروجی دستور «cmd1» جایگزین و دستور «cmd2» را اجرا می‌کند.
$ cmd1 && cmd2     # ج) دستور «cmd2» را تنها در صورت موفقیت‌آمیز بودن «cmd1» اجرا می‌کند.
$ cmd1 || cmd2     # د) دستور «cmd2» را تنها در صورت ناموفقیت‌آمیز بودن «cmd1» اجرا می‌کند.
```

در پایان هر دستور در پوسته مانند برخی از زبان‌های برنامه‌نویسی می‌توان علامت «;» را قرار داد. در صورتی

که دو دستور مستقل در یک خط بیان گردند می‌توان آنها را با این علامت جدا ساخت (قسمت «الف» از شکل قبل). همچنین، پوسته قبل از اجرای یک دستور، عبارت‌های داخل دو علامت «`» را اجرا می‌کند و آنها را با خروجی‌شان جایگزین می‌نماید (قسمت «ب»). در قسمت‌های «ج» و «د» از شکل قبل، دو دستور به صورت شرطی با هم ترکیب می‌گردند: موفقیت یک دستور با توجه به کد برگشتی[1] آن دستور تعیین می‌گردد: به صورت قراردادی در صورتی که یک برنامه مقدار صفر را به عنوان کد برگشتی برگرداند موفقیت آمیز بوده است و در غیر این صورت مشکلی در اجرای برنامه بوجود آمده است. برای مثال دستور «ls» در صورتی که به آن یک آدرس غیر موجود به عنوان ورودی داده شود، مقداری غیر صفر بر می‌گرداند.

گسترش‌ها در پوسته

یکی از ویژگی‌های پوسته که مشخص کردن تعداد زیادی فایل را آسان می‌کند، ویژگی گسترش نام فایل[2] در آن می‌باشد. پوسته عبارت‌های شامل علامت‌های «?»، «*» یا «[..]» را به عنوان الگوی فایل‌ها می‌پذیرد و آن عبارت را با فهرست فایل‌هایی که با آنها مطابقت دارند جایگزین می‌کند. در این الگوها، «?» با هر حرفی، «*» با هر رشته‌ای و «[..]» با هر یک از حروف مشخص شده در آن مطابقت می‌کنند. برای مثال «*.[hc]» با نام همه‌ی فایل‌های شاخه‌ی جاری که پسوند «.c» یا «.h» دارند جایگزین می‌گردد. در شکل زیر چند نمونه از گسترش فایل‌ها در پوسته در ادامه نشان داده می‌شوند.

```
$ ls                       # فهرست همه‌ی فایل‌های شاخه‌ی جاری
a1.c a1.h a2.c a3.c b1.c b1.h b2.c b3.c c1.c c1.h
$ ls a*                    # فهرست فایل‌هایی که با «a» شروع می‌شوند
a1.c a1.h a2.c a3.c
$ ls *.h                   # فهرست فایل‌های با پسوند «.h»
a1.h b1.h c1.h
$ ls a1.[hc]               # فهرست فایل‌هایی که با الگوی «a1.[hc]» مطابقت دارند
a1.c a1.h
$ ls c?.?                  # فهرست فایل‌هایی که با الگوی «c?.?» مطابقت دارند
c1.c c1.h
$ ls ?1.c                  # فهرست فایل‌هایی که با الگوی «?1.c» مطابقت دارند
a1.c b1.c c1.c
```

علاوه بر گسترش نام فایل‌ها، پوسته عبارت‌های دیگری را نیز گسترش می‌دهد. نام‌های پس از علامت «$» با مقدار متغیر پوسته یا مقدار متغیر محیطی با آن نام جایگزین می‌گردند.

1 Return code
2 File name expansion

```
$ echo $VAR        # مقدار متغیر محلی یا متغیر پوسته‌ی «VAR» را چاپ می‌کند
$ echo ${VAR}      # مشابه دستور قبل
$ VAR="abc"        # رشته‌ی «abc» را به متغیر «VAR» نسبت می‌دهد
```

صفحه‌ی راهنمای دستورات

بیشتر دستورات در یونیکس با گرفتن پارامتر ورودی «h-» یا «help--» فهرستی از ویژگی‌هایشان و چگونگی فعال‌سازی آنها را چاپ می‌کنند. علاوه بر آن، به همراه بیشتر دستورات در محیط یونیکس یک صفحه‌ی راهنما وجود دارد که در مورد شیوه‌ی استفاده از آن دستور و امکانات آن توضیح می‌دهد. دستور «man» صفحه‌ی راهنمای یک دستور را نمایش می‌دهد.

```
$ man ls           # صفحه‌ی راهنمای دستور «ls»؛ برای خروج دکمه‌ی «q» را فشار دهید.
```

تمرین یک

پس از دریافت فایل فشرده‌ی «git-2.6.0.tar.gz»، محتویات آن را با دستور زیر در شاخه‌ی «/git-2.6.0» در شاخه‌ی خانه‌ی خود باز (Extract) نمایید:

```
$ tar xzvf git-2.6.0.tar.gz
```

سپس شاخه‌ای با نام ex1 در شاخه‌ی خانه‌ی خود بسازید که ساختاری مانند شکل زیر داشته باشند. دقت نمایید که فایل‌های این درخت را باید از فایل‌های باز شده در /git-2.6.0 بگیرید.

```
git/
    include/
        diff.h
        khash.h
        refs.h
        tar.h
        url.h
        utf8.h
    src/
        diff.c
        pager.c
        refs.c
        url.c
        utf8.c
    contacts/
        Makefile
        git-contacts
        git-contacts.txt
    git-log.sh
    git-clean.sh
```

گام‌های پیشنهادی برای انجام این تمرین:

۱ دریافت و باز کردن فایل «git-2.6.0.tar.gz»

۲ ساختن شاخه‌های مورد نیاز در «/ex1»

۳ یافتن فایل‌های مورد نیاز در شاخه‌ی «/~/git-2.6.0»

۴ کپی کردن فایل‌های مورد نیاز از شاخه‌ی «/~/git-2.6.0»

جلسه‌ی دوم — ورودی و خروجی در پوسته

در این جلسه با مدیریت ورودی و خروجی در پوسته و استفاده از لوله برای ترکیب دستورها آشنا خواهید شد.

مدیریت ورودی و خروجی در پوسته

در حالت عادی، برنامه‌هایی که توسط پوسته اجرا می‌شوند منتظر دریافت ورودی از کاربر می‌شوند و خروجی خود را در صفحه چاپ می‌کنند. امکان «Redirection» در پوسته، فرستادن خروجی یک برنامه به یک فایل و خواندن ورودی آن از یک فایل را ممکن می‌سازد. این کار با استفاده از علامت‌های کوچک‌تر و بزرگ‌تر به صورت زیر قابل انجام است:

```
$ cmd >path          # خروجی دستور «cmd» را به فایل مشخص شده با آدرس «path» می‌نویسد
$ cmd 1>path         # معادل دستور قبل
$ cmd >>path         # خروجی دستور «cmd» را به انتهای فایل «path» اضافه می‌کند
$ cmd <path          # ورودی دستور «cmd» را از فایل «path» می‌خواند
$ cmd 2>path         # خروجی خطای دستور «cmd» را به فایل «path» می‌نویسد
$ cmd 1>path 2>&1    # خروجی دستور «cmd» و خروجی خطای آن را به فایل «path» می‌نویسد
```

استفاده از لوله در پوسته

می‌توان خروجی یک برنامه را توسط لوله[1] به برنامه‌ی دیگری فرستاد؛ یک لوله، که با علامت «|» نشان داده می‌شود، دو سر دارد؛ خروجی برنامه‌ای که در سمت چپ لوله قرار گرفته است به عنوان ورودی به برنامه‌ای که در سمت راست آن قرار گرفته داده می‌شود. در مثال زیر خروجی برنامه‌ی cmd1 به عنوان ورودی به برنامه‌ی cmd2 فرستاده می‌شود:

```
$ cmd1 | cmd2        # خروجی دستور «cmd1» را به صورت ورودی به دستور «cmd2» می‌دهد
```

همان طور که در مثال زیر نشان داده شده است، تعداد زیادی دستور می‌توانند به وسیله‌ی لوله به صورت زنجیره‌ای با هم ترکیب شوند تا خروجی هر دستور به عنوان ورودی به دستور بعدی انتقال یابد.

---

1 Pipe

```
$ cmd | grep "error" | sort | uniq | head -n10
```

در محیط یونیکس برنامه‌های زیادی وجود دارند که کار ساده‌ای را انجام می‌دهند. لوله امکان ترکیب این برنامه‌ها را ایجاد می‌کند. برنامه‌های زیادی در یونیکس برای چنین کاربردی طراحی شده‌اند:

```
$ cat          # ورودی دستور را بدون تغییر چاپ می‌کند
$ wc           # تعداد خطها، کلمه‌ها و حرف‌های ورودی را چاپ می‌کند
$ sort         # خطهای ورودی را به صورت مرتب شده چاپ می‌کند
$ uniq         # خطهای ورودی را پس از حذف خطهای تکراری متوالی چاپ می‌کند
$ tac          # خطهای ورودی را از آخر به اول چاپ می‌کند
$ grep kwd     # خطهای ورودی که شامل الگوی «kwd» هستند را چاپ می‌کند
$ head -n X    # «X» خط اول ورودی را چاپ می‌کند
$ tail -n X    # «X» خط آخر ورودی را چاپ می‌کند
$ tee out      # مشابه دستور «cat» با این تفاوت که یک کپی از ورودی را در فایل «out» می‌نویسد
$ rev          # خطهای ورودی را با معکوس کردن ترتیب حرف‌های آنها چاپ می‌کند
$ shuf         # خطهای ورودی را با ترتیب تصادفی چاپ می‌کند
$ seq X        # اعداد یک تا «X» را در خروجی چاپ می‌کند
$ tr X Y       # حرف «X» در خطهای ورودی را با «Y» جابجا می‌کند و آنها را چاپ می‌کند
$ fmt          # پاراگراف‌های ورودی را به خطهای تقریبا هم اندازه می‌شکند
$ cut -f X     # ستون شماره‌ی «X» از خطهای ورودی را چاپ می‌کند
```

برای اطلاعات بیشتر در مورد این دستورات، به صفحه‌ی راهنمای آنها (با دستور «man cmd») مراجعه شود. توجه به این نکته نیز لازم است که بیشتر این دستورات، با گرفتن آدرس تعدادی فایل به عنوان پارامتر، محتوای آن فایل‌ها را به عنوان ورودی در نظر می‌گیرند. برای نمونه دستور «cat xyz.txt» محتویات فایل «xyz.txt» را چاپ می‌کند و «head -n5 xyz.txt» پنج خط اول فایل «xyz.txt» را چاپ می‌کند. بنابراین، سه دستور زیر خروجی یکسانی دارند:

```
$ cat xyz.txt | head -n5     # استفاده از لوله
$ head -n5 <xyz.txt          # استفاده از Redirection
$ head -n5 xyz.txt           # مشخص کردن فایل به عنوان پارامتر
```

یکی از برنامه‌های پرکاربرد محیط یونیکس برای تغییر محتوای فایل‌ها «sed» می‌باشد. این دستور عملیات زیادی را برای تغییر جریان ورودی پشتیبانی می‌کند. یکی از مهم‌ترین کاربردهای این دستور، جایگزینی یک الگوی عبارت منظم در جریان ورودی با رشته‌ی دیگری است. ادامه شیوه‌ی انجام این کار نشان داده شده است (برای کاربردهای بیشتر، به صفحه‌ی راهنمای این دستور مراجعه شود).

```
$ sed 's/pat/rep/g'        # برای خطهای ورودی الگوی «pat» را با عبارت «rep» جایگزین می‌کند
$ sed '/pat/d'             # خطوط ورودی شامل الگوی «pat» را حذف و بقیه را چاپ می‌نماید
```

در این مثال‌ها، الگوی «pat» می‌تواند یک عبارت منظم باشد.

تمرین دوم

مشابه تمرین یک، پس از دریافت فایل فشرده‌ی «git-2.6.0.tar.gz»، آن را در شاخه‌ی «/git-2.6.0» خانه باز نمایید. سپس شاخه‌ی «ex2» را در شاخه‌ی خانه‌ی خود ایجاد نمایید و فایل‌های خواسته شده را در تمرین‌های زیر (بجز قسمت الف)، با توجه به فایل‌های موجود در شاخه‌ی «/git-2.6.0/~» بسازید.

الف) دستورات زیر را اجرا کنید، توضیح دهید چه عملی انجام می‌دهند و در چه شرایطی مفید هستند؛ در صورت نیاز به صفحه‌ی راهنمای آنها مراجعه نمایید.

```
$ grep malloc test.c | wc -l        #    «test.c» یک فایل ورودی نمونه می‌باشد
$ seq 12 | shuf | head -n1          #    عدد ۱۲ را می‌توانید تغییر دهید
$ cat list | sort | uniq | wc -l    #    فایل «list» می‌تواند فهرستی از نام‌ها باشد
```

ب) فایل «~/ex2/query1.out» را بسازید که شامل فهرست فایل‌هایی که شامل عبارت «get_indexed_object» هستند، باشد.

ج) فایل «~/ex2/query2.out» را بسازید که شامل خطهای ۵۹ تا ۱۰۰ فایل «quote.c» باشد.

د) فایل «~/ex2/query3.out» ایجاد کنید که شامل خطهای متمایز شامل عبارت «#include» در فایل‌های با پسوند «.c» باشد.

جلسه‌ی سوم — اسکریپت‌های پوسته

در این جلسه با عبارت‌های شرطی، حلقه‌ها و اسکریپت‌های پوسته آشنا خواهید شد.

اسکریپت‌های پوسته

می‌توان دنباله‌ای از دستورات پوسته را در یک فایل قرار داد تا آنها را در هنگام نیاز اجرا نمود؛ به این فایل‌ها اسکریپت پوسته[1] گفته می‌شود. در مثال زیر، سه دستور در فایل «cmd.sh» قرار داده می‌شوند و سپس دستورات موجود در این فایل توسط پوسته‌ی «sh» اجرا می‌گردند.

```
$ cat >cmd.sh                     # نوشتن سه خط در فایل «cmd.sh»
date
sleep 1
date
^D
$ sh <cmd.sh                      # اجرای اسکریپت «cmd.sh»
Sat Oct 17 14:27:40 IRST 2015
Sat Oct 17 14:27:41 IRST 2015
$ sh cmd.sh                       # معادل دستور قبل
Sat Oct 17 14:27:57 IRST 2015
Sat Oct 17 14:27:58 IRST 2015
```

همان طور که مشاهده می‌شود پوسته‌ی «sh» دستورات ورودی را یکی پس از دیگری اجرا می‌کند و در صورتی که یک فایل به عنوان پارامتر به آن داده شود، به جای خواندن ورودی، آن را اجرا می‌نماید.

متغیرهای پوسته

در پوسته می‌توان متغیر تعریف کرد و از آنها استفاده نمود؛ پوسته عبارت «$name» را با مقدار متغیر «name» جایگزین می‌کند.

```
$ var="abc"                       # تعریف متغیر «var» با مقدار «abc»
$ echo $var                       # مشاهده‌ی مقدار متغیر «var»
abc
```

1 Shell script

```
$ var="`pwd`"                    # انتساب خروجی دستور «pwd» به متغیر «var»
$ echo $var                      # مشاهده‌ی مقدار متغیر «var»
/home/user
```

پارامترهایی که به یک اسکریپت فرستاده می‌شوند نیز به صورت مشابهی قابل دسترسی هستند؛ «$1» با پارامتر اول، «$2» با پارامتر دوم و در حالت کلی «${n}» با پارامتر «n»-ام جایگزین می‌شوند. در مثال زیر، اسکریپت «cmd.sh» سه پارامتر اول خود را چاپ می‌نماید.

```
$ cat >cmd.sh
echo "Arg #1: $1"
echo "Arg #2: $2"
echo "Arg #3: $3"
^D
$ sh cmd.sh abc def ghi
Arg #1: abc
Arg #2: def
Arg #3: ghi
```

موفقیت دستورها در پوسته

بدیهی است که دستوراتی که در پوسته اجرا می‌شوند می‌توانند موفقیت‌آمیز باشند یا ناموفق خاتمه یابند. برای مثال، در صورتی که آدرس شاخه‌ای که وجود ندارد به دستور «cd» داده شود، این دستور نمی‌تواند شاخه‌ی جاری را تغییر دهد و با خطا خاتمه می‌یابد. اجرای موفق یک دستور به صورت قراردادی با کد برگشتی[1] آن مشخص می‌شود (برای مثال، در زبان C کد برگشتی، مقداری است که از تابع «main» برگشت داده می‌شود). اجرای موفق یک دستور به صورت قراردادی با کد برگشتی صفر مشخص می‌گردد. پوسته کد برگشتی آخرین دستور اجرا شده را در متغیری به نام «?» قرار می‌دهد:

```
$ cd                             # نمونه‌ای از یک دستور موفق
$ echo $?                        # چاپ کد برگشتی بعد از یک دستور موفق
0
$ cd xyz                         # نمونه‌ای از یک دستور ناموفق
$ echo $?                        # چاپ کد برگشتی بعد از یک دستور ناموفق
1
```

در پوسته عبارت‌های شرطی با توجه به موفقیت اجرای دستورها اجرا می‌شوند. از این رو، دستوری به نام

1 Return code

«test» وجود داد که موفقیت آن با توجه به برقرار بودن شرطهای مشخص شده، تعیین می‌شود.

```
$ test "abc" == "def"      # موفقیت، در صورتی که رشته‌ی اول با دوم برابر باشد
$ test "abc" != "def"      # موفقیت، در صورتی که رشته‌ی اول با دوم برابر نباشد
$ test -f xyz.txt          # موفقیت، در صورتی که «xyz.txt» یک فایل باشد
$ test -d xyz              # موفقیت، در صورتی که «xyz» یک شاخه باشد
$ test ! -d xyz            # موفقیت، در صورتی که «xyz» یک شاخه نباشد
```

برای اطلاع از سایر شرطهای دستور «test»، به صفحه‌ی راهنمای این دستور مراجعه نمایید. دستور «true» همواره موفق است (به صورت مشابه دستور «false» همواره ناموفق است) و می‌توان از آن برای حلقه‌ی بی‌نهایت استفاده نمود.

عبارت‌های شرطی و حلقه‌ها در پوسته

برای تکرار تعدادی دستور به ازای مقادیر یا فایل‌های مختلف، پوسته حلقه‌های «for» و «while» را ارائه می‌دهد. حلقه‌ی «for» دنباله‌ای از کلمه‌ها را دریافت می‌کند و به ازای هر یک از آنها یک بار اجرا می‌شود. برای نمونه، حلقه‌ی زیر، شاخه‌های «dir1»، «dir2» و «dir3» را می‌سازد.

```
$ for dir in dir1 dir2 dir3      #این حلقه به ازای همه‌ی عبارت‌های پس از «in» تکرار می‌شود
do
        mkdir $dir
done
```

در حلقه‌ی قبل، در هر بار اجرای حلقه، یکی از رشته‌های مشخص شده بعد از کلمه‌ی کلیدی «in» (در این مثال «dir1»، «dir2» و «dir3») به متغیر پوسته‌ی مشخص شده بعد از کلمه‌ی کلیدی «for» (در این مثال «dir») منسوب می‌گردد. در تعیین دنباله‌ی کلمات برای حلقه‌ی «for» می‌توان از ویژگی گسترش نام فایل در پوسته نیز استفاده نمود. در مثال زیر، نام همه‌ی فایل‌ها با پسوند «.h» در شاخه‌ی «/usr/include/» چاپ می‌شود.

```
#          اجرای بدنه‌ی حلقه به ازای همه‌ی فایل‌های با پسوند «.h» در شاخه‌ی «/usr/include/»
$ for f in /usr/include/*.h
do
        echo $f
done
```

با استفاده از «if» در پوسته می‌توان تعدادی دستور را در صورت برقراری شرطی اجرا نمود. بدنه‌ی «if» تنها در صورتی که اجرای دستور مشخص شده بعد از کلمه‌ی کلیدی «if» موفق باشد، اجرا می‌شود. در مثال زیر، اگر شاخه‌ی «xyz» وجود نداشته باشد، ساخته می‌شود:

```
$ if test ! -d xyz       #         اجرای بدنه در صورتی که شاخه‌ی «xyz» موجود نباشد
then
         mkdir xyz
fi
```

حلقه‌ی «while» تا وقتی که دستوری که بعد از کلمه‌ی کلیدی «while» مشخص می‌شود با موفقیت اجرا می‌شود، اجرا می‌گردد. برای مثال، دستور زیر تا ساخته‌شدن شاخه‌ی «xyz» صبر می‌کند.

```
$ while test ! -d xyz    #         تکرار بدنه‌ی حلقه تا وقتی «test» با موفقیت اجرا شود
do
         sleep 1
done
```

یکی از پر استفاده‌ترین کاربردهای حلقه‌های «while» خواندن خطوط ورودی می‌باشد؛ این کار را می‌توان با استفاده از دستور داخلی پوسته «read» به صورت زیر انجام داد:

```
$ find /usr/include -name '*.h' | while read ln
do
         basename "$ln"
done | sort | uniq
```

هر خط ورودی یک بار به متغیر «ln» منسوب می‌شود و بدنه‌ی حلقه یک بار برای آن تکرار می‌گردد. همان طور که در این مثال نشان داده شده است، مشابه دستورات معمولی، حلقه‌ها نیز می‌توانند در دنباله‌ی لوله‌ها استفاده شوند. بنابراین، خروجی دستور «find» به حلقه‌ی «while» فرستاده می‌شود و خروجی این حلقه به دستور «sort» و خروجی این دستور نیز به دستور «uniq» فرستاده می‌شود. مجموعه‌ی این دستورات، نام‌های متمایز همه‌ی فایل‌های با پسوند «.h» در شاخه‌ی «/usr/include/» را چاپ می‌کند. دستور «basename» که در بدنه‌ی حلقه فراخوانی شده است، نام فایل در آدرس داده شده را چاپ می‌کند. چگونگی استفاده از دستور «basename» و «dirname» در ادامه نشان داده می‌شود.

```
$ basename /usr/include/stdio.h          # چاپ نام فایل در یک آدرس
stdio.h
$ basename /usr/include/stdio.h .h       # چاپ نام فایل بدون پسوند «.h»
stdio
$ dirname /usr/include/stdio.h           # چاپ نام شاخه در یک آدرس
/usr/include
```

تمرین سوم

الف) دستورات زیر را اجرا کنید، توضیح دهید چه عملی انجام می‌دهند و در چه شرایطی مفید هستند؛ در صورت نیاز به صفحه‌ی راهنمای دستورات مراجعه نمایید.

```
$ cmp f1 f2 || echo "Files do not match"        # فایل‌های «f1» و «f2» ورودی هستند
$ cmp f1 f2 && echo "Files match"               # فایل‌های «f1» و «f2» ورودی هستند
$ test -d dir && echo "Directory already exists" #        شاخه‌ی «dir» ورودی است
```

ب) همه‌ی فایل‌های شامل رشته‌ی «get_indexed_object» را از شاخه‌ی «git-2.6.0/» به شاخه‌ی «~/ex3» کپی نمایید و رشته‌ی «get_indexed_object» در این فایل‌ها را با «get_iobject» جایگزین کنید.

ج) شاخه‌ی «~/ex3/c» را بسازید و همه‌ی فایل‌های شاخه‌ی «~/ex3» که پسوند «.c» دارند را به این شاخه انتقال دهید و پسوند آنها را به «.txt» تغییر دهید.

د) اسکریپتی برای قسمت ب بنویسید که با گرفتن نام یک شاخه (پارامتر اول) همه‌ی فایل‌های موجود در آن شاخه و زیر شاخه‌های آن که شامل یک عبارت ورودی هستند (پارامتر دوم) را با عبارت دیگری (پارامتر سوم) جایگزین کند و در شاخه‌ی جاری قرار دهد.

جلسه‌ی چهارم — مروری بر پوسته

هدف در این جلسه یادآوری برخی از مطالب مطرح شده در جلسات گذشته در مورد پوسته است. آزمایش این جلسه به صورت تعدادی گام سازماندهی شده است؛ این گام‌ها را انجام و به پرسش‌های مطرح شده دقیـق پاسخ دهید.

۱ شاخه‌ی «ex4» را بسازید و با دستورات زیر چند فایل در آن ایجاد نمایید:

```
$ cat >a1
abc
def
ghi
jkl
mno
^D
$ cat >a2
123
456
789
012
345
^D
$ seq 5 >a3
```

سپس بیان کنید دستورات زیر چه عملی را انجام می‌دهند:

```
$ cat a2 a3 >a4
$ cat a2 | sort >a5
$ cat a1 a2 | cat >a6
```

۲ توضیح دهید دستورات زیر چه تفاوتی با هم دارند. کدام دستورها معادل هستند و کدام دستورها خطوط مرتب شده‌ی فایل «a2» را چاپ می‌کنند.

```
$ echo a2 | sort
```

```
$ cat a2 | sort
$ sort <a2
$ sort a2
$ sort ˜/ex4/a2
$ sort ../ex4/a2
$ echo ../ex4/a2 | sort
```

۴ تفاوت دستورات زیر را بیان کنید و مشخص کنید کدام دستور برای کپی کردن فایل «a2» به «a7» مناسب است.

```
$ cp a2 >a7
$ echo a2 | cp a7
$ cat a2 >a7
$ cp <a7 >a2
$ cat <a2 >a7
```

۵ فایل «a7» را به صورتی تغییر دهید که خطهای آن مرتب شوند (دستورات لازم را بنویسید).

۶ مشخص کنید در هر یک از دستورات زیر، چه پارامترهایی به دستور «cmd» فرستاده می‌شوند؟ توضیح دهید. با جایگزینی دستور «echo» به جای «cmd» ادعای خود را ثابت کنید.

```
$ cmd a
$ cmd *
$ cmd "*"
$ cmd ˜/ex3/a*
$ cmd a[134]
$ var=1
$ cmd $var
$ cmd "$var"
$ cmd ’$var’
$ cmd ls
$ cmd ’ls’
$ cmd ‘ls‘
$ cmd echo abc
$ cmd ’echo abc’
$ cmd "echo abc"
$ cmd ‘echo abc‘
$ cmd cat a1
$ cmd ‘cat a1‘
$ cmd ‘echo a*‘
```

۷ تفاوت دستورات زیر چیست و کدام برای چاپ اعداد یک تا نه مناسب است؟

```
$ echo 9 | seq
$ seq <9
$ seq 9
$ seq "9"
$ seq ‘9‘
$ seq ‘echo 9‘
```

۸ تفاوت دستورات زیر چیست و کدام برای چاپ خطوط *متمایز* فایل‌ها مناسب است؟

```
$ sort uniq a*
$ echo a* | sort | uniq
$ cat a* | sort | uniq
$ sort a* | uniq
$ uniq a* | sort
$ uniq ‘sort a*‘
$ sort ‘cat a* | uniq‘
$ sort ‘ls a*‘ | uniq
$ sort "find ˜/ex4/ -name ’a*’" | uniq
```

۹ دستورات زیر چه عملی انجام می‌دهند؟

```
$ for x in a1 a2 a3 a4 a5; do cat $x; done
$ for x in a*; do cat $x; done
$ for x in a[245]; do cat $x; done
$ for x in ˜/ex4/*; do cat $x; done
```

۱۰ دستورات زیر چه عملی انجام می‌دهند؟

```
$ mkdir txt
$ for x in a*; do
        cp $x txt/${x}.txt
done
```

۱۱ حلقه‌ی زیر چه عملی انجام می‌دهد؟

```
$ for x in a?; do
        cmp -s a2 $x && echo $x
done
```

۱۲ حلقه‌ی زیر چه تفاوتی با حلقه‌ی قبل دارد؟

```
$ sort <a2 >/tmp/t1
$ for x in ˜/ex4/a?; do
        sort <$x >/tmp/t2
        cmp -s /tmp/t1 /tmp/t2 && echo $x
done
```

جلسه‌ی پنجم — نوشتن و ترجمه‌ی برنامه‌ها

در این جلسه با امکاناتی که معمولا در یونیکس برای نوشتن برنامه‌ها و ترجمه‌ی آنها موجود هستند، 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 توزیع‌های لینوکس برنامه‌ها را ترجمه نمود.

```
$ cat >test.c
#include <stdio.h>
int main(void)
{
	printf("Hello world!\n");
	return 0;
}
^D
$ cc test.c              #    ترجمه‌ی فایل «test.c»؛ نام فایل اجرایی حاصل «a.out» می‌باشد
$ ./a.out                #    اجرای فایل «a.out»
Hello world!
$ cc -o test test.c      #    مشخص کردن نام فایل خروجی با پارامتر «-o»
$ ./test
Hello world!
```

همان طور که مشاهده می‌شود، دستور «cc» یک مترجم[1] زبان «C» است که فایلی که آدرس آن به عنوان پارامتر به آن داده می‌شود را ترجمه می‌کند. 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»[2] برای ترجمه‌ی برنامه‌ها استفاده می‌شود و معمولا دستور «cc» معادل دستور «gcc» و «c++» معادل دستور «g++» می‌باشد.

1 Compiler
2 GNU Compiler Collection

شکستن کد به تعدادی فایل

کد برنامه‌های نسبتا بزرگ به تعدادی فایل شکسته می‌شود. در صورتی که تعداد فایل‌های کد برنامه زیاد باشد (یا در صورتی که زبان‌های متفاوتی در آن استفاده شده باشند)، می‌توان تولید فایل اجرایی را در دو گام انجام داد. در گام اول فایل‌های «Object» تولید می‌شوند؛ این فایل‌ها خروجی گام ترجمه‌ی مترجم هستند. در گام دوم این فایل‌ها با هم ترکیب می‌شوند تا یک فایل اجرایی حاصل شود. به عملی که در گام اول انجام می‌شود، ترجمه و به عملی که در گام دوم انجام می‌شود، لینک[1] گفته می‌شود. چگونگی انجام این دو گام در ادامه نشان داده می‌شود (فرض کنید فایل‌های «src1.c»، «src2.c» و «src3.c» شامل کد برنامه هستند):

```
$ cc -c src1.c                # «src1.c» برای فایل «src1.o» با نام «Object» ساختن یک فایل
$ cc -c src2.c                # «src2.c» برای فایل «src2.o» با نام «Object» ساختن یک فایل
$ cc -c src3.c                # «src3.c» برای فایل «src3.o» با نام «Object» ساختن یک فایل
$ ls
src1.c src1.o src2.c src2.o src3.c src3.o
$ cc -o out src1.o src2.o src3.o #  «out» برای ساختن فایل اجرایی «Link» انجام عمل
$ ./out                       # اجرای فایل اجرایی حاصل
```

یکی از مزیت‌های تولید فایل اجرایی در این دو گام، در هنگام تغییر کد است: اگر فقط یکی از فایل‌ها تغییر کند، لازم نیست سایر فایل‌ها دوباره ترجمه شوند و فقط ترجمه‌ی فایل تغییر یافته و لینک کردن فایل‌های «Object» کافی است. در صورتی که سرعت ترجمه اهمیت نداشته باشد، بسیاری از مترجم‌ها این دو گام را با یک دستور انجام می‌دهند:

```
$ cc -o out src1.c src2.c src3.c #  «Object» تولید فایل اجرایی در یک مرحله، بدون تولید فایل‌های
```

متغیرهای محیطی

به هر پردازه در یونیکس، از جمله پوسته، تعدادی متغیر محیطی[2] 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 (متغیرهای محیطی مشابه متغیرهای پوسته خوانده می‌شوند).

1 Linking
2 Environment Variable

```
$ export MYENV="my env"          # تعریف متغیر محیطی «MYENV» با مقدار «my env»
$ echo $MYENV                    # چاپ مقدار متغیر محیطی «MYENV»
my env
$ env                            # چاپ همه‌ی متغیرهای محیطی پوسته و مقدارشان
...
MYENV=my env
...
```

یکی از متغیرهای محیطی مهم در یونیکس، متغیر «PATH» می‌باشد. این متغیر، فهرستی از شاخه‌هایی 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 (برای اجرای سایر فایل‌ها باید آدرس آنها مشخص شود).

```
$ echo $PATH                     # متغیر محیطی «PATH» شامل شاخه‌هایی مثل «/bin»
/sbin:/usr/sbin:/bin:/usr/bin
$ ls /bin                        # شاخه‌ی «/bin» شامل فایل اجرایی «uname» است
...
uname
...
$ uname                          # بنابراین «uname» معادل «/bin/uname» است
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 صفحه‌ی راهنمای تابع «getenv()» مراجعه کنید.

پارامترهای ورودی برنامه

همان طور که در جلسه‌های گذشته مشاهده کرده‌اید، دستوراتی که در پوسته اجرا می‌شوند تعدادی پارامتر می‌پذیرند. در صورتی که برنامه‌ی اجرا شونده در زبان «C» نوشته شده باشد، این پارامترها به تابع «main()» آن برنامه فرستاده می‌شوند. برای دسترسی به پارامترها، شکل تابع «main()» می‌تواند به صورت زیر باشد:

```
int main(int argc, char *argv[])
```

در زمان اجرای برنامه، متغیر «argc» تعداد پارامترها و متغیر «argv» پارامترهای داده شده خواهند بود. به صورت قراردادی پارامتر اول (argv[0]) همواره نـام خود برنامـه‌ی اجرا شونده اسـت. بنابراین در صورتی کـه برنامه‌ای در پوسته به صورت زیر صدا زده شود:

```
$ cmd hello world
```

مقدار «argc» برابر ۳، مقدار argv[0] برابر «cmd»، مقـدار argv[1] برابـر «hello» و مقـدار argv[2] برابـر «world» خواهد بود.

ساختن خودکار فایل‌های خروجی (اختیاری)

برای جلوگیری از تکرار دستورات لازم برای تولید فایل اجرایی یـک برنامه، مـی‌توان سـاخت فایل اجرایـی را از کد برنامه‌ها به صورت خودکار انجام داد. یک راه برای ترجمه‌ی خودکار تعداد زیادی فایـل، نوشتن اسکریپتـی می‌باشد که دستورات لازم برای تولید فایل اجرایی را نگه دارد. 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 فایـل ورودی (کـه معمولا «Makefile» یا «makefile» نامیده مـی-شود)، 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 دستورهای 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 بـرای ساختن آن، دستورهایی کـه در خطهای بـعد 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 فایـل وجود نداشتـه

باشد یا موجود باشد و حداقل یکی از پیش‌نیازهای آن جدیدتر از آن باشد. 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 ولی فایل «out» باید دوباره ساخته شود. یک «Makefile» نمونه برای ساختن این فایل در ادامه نشان داده می‌شود:

```
$ cat Makefile
src1.o: src1.c                        # فایل «src1.o» به فایل «src1.c» احتیاج دارد
	cc -c src1.c                  # دستور لازم برای ساختن «src1.o»
src2.o: src2.c
	cc -c src2.c
src3.o: src3.c
	cc -c src3.c
out: src1.o src2.o src3.o             # فایل «out» به فایل‌های «src[123].o» احتیاج دارد
	cc -o out src1.o src2.o src3.o
```

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

```
$ make out                            # ساختن فایل «out»
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                        # تغییر فایل «src3.c»
$ make out                            # فقط فایل‌های «src3.o» و «out» دوباره ساخته می‌شوند
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 کنید.

تمرین پنجم

در شاخه‌ی «ex5/~» 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/ex5
prog      ./ex6
arguments:
          1        hello
          2        world
```

در این خروجی، عبارت پس از «user» مقدار متغیر محیطی «USER»، عبارت بعد از «home» مقدار متغیر محیطی «HOME»، مقدار پس از «pid» شماره‌ی پردازه‌ی ایجاد شده (تابع «()getpid» را فراخوانی کنید)، مقدار پس از «uid» شماره‌ی کاربری (تابع «()getuid» را فراخوانی کنید)، مقدار پس از عبارت «path» شاخه‌ی جاری پردازه (تابع «()getcwd» را فراخوانی کنید)، عبارت پس از «prog» نام برنامه‌ای که نوشته‌اید هستند. در قسمت پایانی خروجی پارامترهایی که به برنامه فرستاده شده‌اند باید نمایش داده شوند. در نمونه‌ی بالا فرض شده است برنامه به صورت «ex6 hello world/.» فراخوانی شده است. برای اطلاعات بیشتر در مورد توابع مورد نیاز، به صفحه‌های راهنمای آنها مراجعه نمایید.

در این جلسه با توابع مدیریت پردازه‌ها در سیستم عامل‌های مشابه یونیکس آشنا خواهید شد. این بخش به سه بخش تقسیم شده است. بخش اول ساختن یک پردازه‌ی جدید را شرح می‌دهد، بخش دوم به اجرای یک برنامه که در فایل سیستم قرار دارد، می‌پردازد و بخش سوم شیوه‌ی انتظار در یک پردازه برای پردازه‌های فرزند آن را توصیف می‌نماید.

ایجاد یک پردازه

با فراخوانی سیستمی «fork()»، سیستم عامل پردازه‌ی جدیدی ایجاد می‌کند که یک کپی از پردازنده‌ی فراخوانی کننده می‌باشد. بنابراین پس از اجرای این فراخوانی سیستمی، دو پردازه‌ی پدر و فرزند هر دو اجرای خود را با برگشتن از تابع «fork()» ادامه می‌دهند. بنابراین، در مثال زیر عبارت قبل از فراخوانی سیستمی «fork()» یک بار و عبارت پس از آن دو بار (یک بار در پردازه‌ی پدر و یک بار در پردازه‌ی فرزند) اجرا می‌گردد (تابع «getpid()» شماره‌ی پردازه‌ی فراخوانی کننده را بر می‌گرداند).

```
printf("%d: before forking\n", getpid());
fork();
printf("%d: after forking\n", getpid());
```

مقدار برگشت داده شده توسط «fork()» در پردازه‌ی پدر و فرزند متفاوت است و با استفاده از آن می‌توان به راحتی تشخیص داد که عبارت بعد، در کدام پردازه در حال اجرا است: این فراخوانی در پردازه‌ی پدر مقدار «PID» پردازه‌ی فرزند (مقداری بزرگ‌تر از صفر) و در پردازه‌ی فرزند صفر را بر می‌گرداند:

```
printf("Before fork syscall\n");
if (fork())                         /*    ایجاد یک پردازه‌ی جدید    */
      printf("The parent process\n");
else
      printf("The child process\n");
```

در صورتی که در اجرای این فراخوانی سیستمی مشکلی رخ دهد (مثلا به دلیل کمبود حافظه، سیستم عامل نتواند پردازه‌ی جدیدی ایجاد نماید) این تابع مقدار منفی یک را بر می‌گرداند.

اجرای یک برنامه

برای اجرای یک برنامه که در فایل سیستم وجود دارد می‌توان یکی از توابع خانواده‌ی «exec()» را فراخوانی نمود. یکی از این توابع، تابع «execvp()» می‌باشد. ورودی اول این تابع، آدرس برنامه‌ی مورد نظر و ورودی دوم آن آرایه‌ای است که پارامترهایی که به پردازه‌ی ایجاد شده فرستاده می‌شوند (ورودی‌های فرستاده شده به تابع «main()» در یک برنامه)، را مشخص می‌کند. این آرایه باید با یک عنصر «NULL» خاتمه پذیرد. به صورت قراردادی، در درایه‌ی صفرم این آرایه، آدرس برنامه تکرار می‌شود.

```
char *argv[] = {"ls", "/home", NULL};
execvp("ls", argv);                    /*  اجرای دستور «ls /home»  */
```

پس از این فراخوانی، قسمت‌های کد و داده‌ی پردازه از بین می‌روند و با مقدار مناسب برای پردازه‌ی جدید جایگزین می‌گردند. بنابراین در صورت موفقیت‌آمیز بودن این فراخوانی، هیچ یک از عبارت‌های پس از این فراخوانی اجرا نمی‌شوند. در صورت رخداد خطا (برای مثال، موجود نبودن برنامه‌ی مشخص شده)، این فراخوانی مقدار منفی یک را بر می‌گرداند.

انتظار برای اتمام پردازه‌ها

فراخوانی سیستمی «wait()» منتظر خواهد بود تا یکی از پردازه‌های فرزند پردازه‌ی فراخوانی کننده خاتمه یابد. مقدار برگشت داده شده از این تابع، شماره‌ی «PID» پردازه‌ی خاتمه یافته است و اطلاعاتی در مورد خاتمه‌ی این پردازه (از جمله مقدار کد برگشتی آن) در متغیری که آدرس آن به این تابع فرستاده می‌شود قرار می‌گیرد. در مثال زیر، شیوه‌ی استفاده از «wait()» نمایش داده شده است.

```
pid = wait(&status);            /*  انتظار برای خاتمه‌ی یک پردازه‌ی فرزند  */

printf("pid %d exited with return code %d\n",
       pid, WEXITSTATUS(status));
```

همان طور که نشان داده شده است، با استفاده از ماکروی[1] «WEXITSTATUS» می‌توان کد برگشتی یک برنامه را از مقداری که این فراخوانی سیستمی در متغیر «status» قرار می‌دهد، استخراج نمود.

1 Macro

تمرین ششم

برنامه‌ای به نام «ex6.c» در شاخه‌ی «ex6» بنویسید که برنامه‌ای که داده می‌شود را اجرا کند، منتظر پایان این برنامه بماند و کد برگشتی آن را چاپ کند. برنامه‌ای که باید اجرا شود و پارامترهای آن به صورت پارامتر به برنامه‌ی «ex6» فرستاده می‌شوند. در مثال زیر برنامه‌ی «ls» با پارامتر اول «1.txt» اجرا می‌گردد:

```
$ ./ex6 ls 1.txt
```

در صورتی که کد برگشتی برنامه غیر صفر بود، اجرای برنامه باید پس از یک ثانیه تکرار شود. این کار باید تا زمانی ادامه پیدا کند که برنامه مقدار صفر را برگرداند. برای تأخیر، می‌توانید تابع «sleep()» را فراخوانی کنید.

گام‌های پیشنهادی برای انجام این تمرین:

۱ ایجاد فایل «ex6.c» و ترجمه‌ی آن

۲ ایجاد یک پردازه‌ی جدید با فراخوانی «fork()» و بررسی آن با چاپ پیغام‌هایی

۳ انتظار برای اتمام پردازه‌ی فرزند در پردازه‌ی پدر با فراخوانی «wait()» و چاپ کد برگشتی آن

۴ اجرای یک برنامه در پردازه‌ی فرزند با فراخوانی «exec()»

۵ تکرار ایجاد پردازه‌ی فرزند در پردازه‌ی پدر در صورت دریافت کد بازگشتی غیر صفر

جلسه‌ی هفتم — انتقال داده با لوله

در این جلسه با شیوه‌ی مدیریت فایل‌های باز پردازه‌ها در یونیکس و استفاده از لوله برای انتقال اطلاعات بین آنها آشنا خواهید شد.

فایل‌ها در یونیکس

در یونیکس علاوه بر فایل‌های ذخیره شده در دیسک، بسیاری از منابع موجود در سیستم عامل (از جمله اتصالات شبکه، لوله‌ها و بسیاری از «Device»-ها از جمله کارت‌های صوتی، دیسک‌ها، حافظه و حافظه‌ی کارت‌های گرافیکی) نیز توسط فایل قابل دسترسی هستند. استفاده از فایل برای این کاربردها از یک سو موجب سادگی رابط هسته برای مدیریت این منابع و دسترسی به آنها گشته است و از سوی دیگر موجب شده است بسیاری از برنامه‌ها بدون وابستگی به نوع فایل‌ها، برای همه‌ی این انواع فایل قابل استفاده باشند. در این بخش برخی از توابع موجود در یونیکس برای دسترسی به فایل‌ها معرفی می‌گردند؛ این توابع در بیشتر سیستم‌های عامل مبتنی بر یونیکس فراخوانی‌های سیستمی هستند.

شناسه‌های فایل

در یونیکس هر فایل باز[1] پردازه با یک عدد مشخص می‌شود؛ به این عدد شناسه‌ی فایل[2] گفته می‌شود. به صورت قراردادی، فایل شماره‌ی صفر به ورودی استاندارد («stdin» در کتابخانه‌ی استاندارد زبان C)، فایل شماره‌ی یک به خروجی استاندارد («stdout») و فایل شماره‌ی دو به خروجی خطا («stderr») اختصاص می‌یابند. پردازه‌ها می‌توانند با استفاده توابع مناسب، شناسه‌های فایل جدیدی را ایجاد نمایند (برای مثال با فراخوانی تابع ()open یا ()dup) یا آنها را ببندند (با فراخوانی تابع ()close).

خواندن از و نوشتن به فایل‌ها

تابع ()read با گرفتن یک شناسه‌ی فایل، یک آرایه‌ی کارکتری و اندازه‌ی آن، از فایل مشخص شده می‌خواند. در مثال زیر، استفاده از این تابع نشان داده شده است.

---

1 Open file
2 File descriptor

```
#include <unistd.h>

char buf[128];                /*        خروجی تابع read() در این آرایه ریخته می‌شود    */
ssize_t nr = read(0, buf, 128);  /*     یک مثال از فراخوانی تابع read()    */
```

مقدار برگشت داده شده توسط این تابع (متغیر «nr» در مثال قبل) تعداد بایت‌های خوانده شده از شناسه‌ی فایلی که با ورودی اول داده می‌شود را نشان می‌دهد. در صورتی که خطایی در فراخوانی این تابع رخ دهد (مشابه بسیاری از فراخوانی‌های سیستمی دیگر) یک عدد منفی برگشت داده می‌شود و عدد صفر به این معنی است که همه‌ی محتوای فایل خوانده شده است.

تابع write() بایت‌های داده شده را (که توسط یک اشاره‌گر و تعداد بایت‌ها مشخص می‌شود) به یک فایل می‌نویسد. عدد برگردانده شده توسط تابع write() تعداد بایت‌های نوشته شده در شناسه‌ی فایل داده شده را مشخص می‌کند. در صورتی که خطایی رخ دهد، عددی منفی از این تابع برگردانده خواهد شد.

```
#include <unistd.h>

char buf[] = "Hello World!\n";   /*        رشته‌ای که نوشته می‌شود    */
ssize_t nw = write(1, buf, 12);  /*   تعداد بایت‌های نوشته شده در «nw» قرار می‌گیرد    */
```

تابع open() یک فایل در فایل سیستم را باز می‌کند و به آن یک شناسه‌ی فایل آزاد (که در حال استفاده نیست) تخصیص می‌دهد. تابع close() یک شناسه‌ی فایل را می‌بندد و سپس، شناسه‌ی فرستاده شده به این فراخوانی سیستمی آزاد می‌شود. برای جزئیات بیشتر به صفحه‌ی راهنمای این فراخوانی‌ها مراجعه شود.

استفاده از توابع کتابخانه‌ای زبان C برای دسترسی به شناسه‌های فایل

استفاده‌ی مستقیم از توابع read() و write() کمی دشوار است؛ این توابع فقط رشته‌ها را می‌پذیرند (برای مثال، اعداد را نمی‌توان مستقیما توسط این دو تابع چاپ کرد) و همچنین باید خروجی این توابع بررسی شود تا تعداد بایت‌های نوشته شده یا خوانده شده (که می‌تواند کمتر از مقدار درخواست شده باشد) مشخص گردد. برای راحتی بیشتر، می‌توان برای این شناسه‌ها یک داده از نوع «FILE» ایجاد نمود و سپس با استفاده از توابع کتابخانه‌ی استاندارد زبان C مثل fprintf() و fscanf() به آنها به صورت غیر مستقیم دسترسی داشت. برای ساخت یک «FILE» از یک شناسه‌ی فایل می‌توان از تابع fdopen() استفاده نمود. در مثال زیر، فراخوانی این تابع نشان داده شده است.

```
FILE *fp = fdopen(fd, "w");
fprintf(fp, "Hello\n");
fclose(fp);
```

پارامتر دوم تابع ()fdopen (مشابه تابع ()fopen) نوع باز کردن فایل را مشخص می‌کند: مثل «w» برای نوشتن به فایل و «r» برای خواندن از آن.

ایجاد لوله

لوله‌ها[1] (که در جلسه‌های گذشته معرفی شدند) در یونیکس با استفاده از فراخوانی سیستی ()pipe ساخته می‌شوند. لوله یک بافر[2] (یعنی حافظه‌ی محدودی که برای انتقال داده‌ها استفاده می‌گردد) در سیستم عامل است که با دو شناسه‌ی فایل قابل دسترسی می‌باشد: یک شناسه‌ی فایل برای سر نوشتن و دیگری برای سر خواندن. داده‌هایی که به سر نوشتن لوله نوشته می‌شوند، از سر خواندن آن خوانده می‌شوند. با شناسه‌ی نوشتن یک لوله، می‌توان داده‌ها را به لوله انتقال داد (تابع ()write). به صورت مشابه، با استفاده از شناسه‌ی خواندن یک لوله، می‌توان داده‌های نوشته شده به یک لوله را توسط تابع ()read خواند.

تابع ()pipe یک لوله می‌سازد و شناسه‌ی فایل دو سر این فایل را در یک آرایه‌ی با طول دو (که به عنوان ورودی به آن داده می‌شود) می‌نویسد.

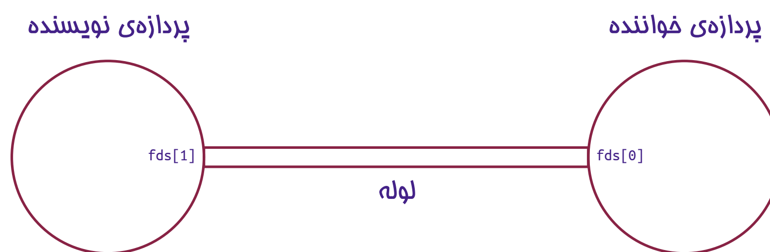
```
int fds[2];        /* شناسه‌های خواندن و نوشتن در این آرایه قرار می‌گیرند */
pipe(fds);         /* fds[0] سر خواندن و fds[1] سر نوشتن هستند */
```

با استفاده از یک پایپ می‌توان داده‌هایی را بین دو پردازه انتقال داد و معمولا پس از این فراخوانی، با تابع ()fork پردازه‌ی جدیدی ساخته می‌شود. سپس یکی از این پردازه‌ها از سر نوشتن لوله داده‌ها را می‌نویسد و پردازه‌ی دیگر از سر خواندن لوله، داده‌ها را می‌خواند:

```
char buf[100];
pipe(fds);
if (fork()) {                          /* پردازه‌ی پدر: نویسنده */
      close(fds[0]);                   /* احتیاجی به سر خواندن در نویسنده نیست */
      write(fds[1], "Hello\n", 6);     /* نوشتن رشته‌ای در لوله */
} else {                               /* پردازه‌ی فرزند: خواننده */
      close(fds[1]);                   /* احتیاجی به سر نوشتن در خواننده نیست */
      read(fds[0], buf, 100);          /* خواندن از لوله */
}
```

1 Pipe
2 Buffer

در شکل زیر، استفاده از لوله بین دو پردازه‌ی مثال قبل نشان داده شده است.



تمرین هفتم

برنامه‌ای را در نظر بگیرید که پردازش را در دو گام انجام می‌دهد: در گام اول، داده‌هایی را تولید می‌کند و در گام دوم این داده‌ها را پردازش می‌نماید. فرض کنید بیش از دو پردازنده در سیستم عامل برای اجرای این برنامه موجود باشند؛ این برنامه فقط می‌تواند از یکی از این پردازنده‌ها به صورت همزمان استفاده نماید. برای استفاده از دو پردازنده، می‌توان برنامه را تغییر داد تا با استفاده از لوله برای انتقال داده‌ها، پردازش را به دو قسمت تقسیم کند که در دو پردازه‌ی مجزا اجرا شوند. این تغییرات را انجام دهید.

پس از دریافت فایل «ex7.c»، آن را تغییر دهید. در این برنامه، قسمت اول پردازش در تابع prod() و قسمت دوم در تابع cons() انجام می‌شود (در حلقه‌ی تابع main()، هر خروجی prod() به تابع cons() فرستاده می‌شود). ابتدا با تابع pipe() یک لوله ایجاد نمایید و سپس با تابع fork() یک پردازه‌ی جدید بسازید. در پردازه‌ی پدر، تابع prod() را صدا بزنید و خروجی آن را به سر نوشتن لوله بنویسید. در پردازه‌ی فرزند، داده‌هایی که توسط پردازه‌ی پدر نوشته می‌شود را از سر خواندن لوله بخوانید و به تابع cons() بفرستید.

گام‌های پیشنهادی برای انجام این تمرین:

۱ دریافت و ترجمه‌ی فایل «ex7.c»

۲ ایجاد یک پردازه‌ی جدید با فراخوانی fork()

۳ ساختن یک لوله قبل از ایجاد پردازه‌ی جدید با فراخوانی pipe()

۴ ایجاد یک «FILE» با fdopen() برای سر نوشتن در پردازه‌ی پدر و برای سر خواندن در فرزند

۵ آزمایش درستی عملکرد لوله برای انتقال یک رشته‌ی آزمایشی

۶ نوشتن اعداد محاسبه شده توسط prod() در پردازه‌ی پدر به لوله و خواندن و فرستادن آنها به تابع cons() در پردازه‌ی فرزند

جلسه‌ی هشتم — کتابخانه‌ی PThreads

یکی از راه‌های اجرای همروند دستورات در سیستم‌های عامـل استفاده از ریسمان[1] مـی‌باشد (اگـر چه ریسـه، ریسمان یا نخ ترجمه‌ی مناسبی برای «Thread» به نظر نمـی‌رسند، بـه علت رواج آنها، در این مستنـد نیز از عبارت ریسمان استفاده می‌شود). در هر پردازه، تعدادی ریسمان می‌توانند به صورت همروند اجرا شوند. چون همه‌ی ریسمان‌های یک پردازه در فضای آدرس پردازه اجرا می‌گردند، می‌توانند با استفاده از حافظه‌ی مشترک (برای مثال متغیرهای سراسری[2]) با هم ارتباط داشته باشند. اما برای جلوگیری از مشکل وضعیت رقابتـی[3]، دسترسی همزمان ریسمان‌ها به متغیرهای مشترک باید مدیریت شود.

استانـدارد «POSIX» رابـط[4] یک کتابخانـه‌ی ریسمـان را معرفـی کرده اسـت که «POSIX Threads» یـا «PThreads» نامیده می‌شود. این کتابخانه در بسیاری از سیستم‌های عامل رایج از جمله لینوکس پیاده‌سازی شده است. در این جلسه با این کتابخانه آشنا می‌شوید. برای استفاده از کتابخانه‌ی PThreads در لینوکس باید پارامتر -lpthread را به Linker فرستاد (در برخی از محیط‌ها باید پارامتر -pthread را هم به Linker و هم به مترجم فرستاد).

آشنایی با کتابخانه‌ی PThreads

با تابع pthread_create() می‌توان یک ریسمان ایـجاد نمود. ورودی اول ایـن تابع یـک اشاره‌گـر به یـک متغیر برای ذخیره‌سازی شناسه‌ی[5] ریسمان جدید است و ورودی سوم این تابع، تابعی است که ریسمان جدید باید از آن اجرای خود را شروع کنـد. برای اطلاع بیشتر در مورد ورودی‌های ایـن تابع، بـه صفحه‌ی راهنمای آن مراجعه نمایید. در صورتی که این تابع با موفقیت اجرا شود، ریسمان جدید اجرای خود را از تابع مشخص شده در ورودی سوم شروع خواهد نمود.

```
#include <pthread.h>

pthread_t tid;
```

1 Thread
2 Global
3 Race condition
4 Interface
5 Identifier

```
pthread_create(&tid, NULL, func, NULL);    /*       ایجاد یک ریسمان از تابع «func»   */
```

تابعی که به ()pthread_create فرستاده می‌شود یک اشاره‌گر دریافت می‌کند (ورودی چهارم ()pthread_create) و یک اشاره‌گر بر می‌گرداند:

```
void *func(void *dat)
{
         printf("Thread started!\n");
         return NULL;
}
```

برای انتظار برای اتمام یک ریسمان می‌توان تابع ()pthread_join را فراخوانی کرد (مشابه تابع ()wait برای پردازه‌ها). مقدار برگشت داده شده توسط تابع اصلی یک ریسمان (ورودی سوم ()pthread_create) در ورودی دوم این تابع (در صورتی که به یک متغیر * void اشاره کند) قرار می‌گیرد.

```
pthread_join(tid, NULL);                    /*      انتظار برای خاتمه‌ی ریسمان «tid»   */
```

مدیریت دسترسی‌های همزمان

کتابخانه‌ی PThreads امکاناتی را برای مدیریت دسترسی‌های همزمان ارائه می‌دهد. یکی از آنها، قفل‌های «Mutex» می‌باشند. برای استفاده از این قفل، باید یک متغیر با نوع «pthread_mutex_t» ساخت و با استفاده از تابع «()pthread_mutex_init» به آن مقدار اولیه داد. با تابع «()pthread_mutex_lock» این قفل بسته و با تابع «()pthread_mutex_unlock» باز می‌شود. در هر لحظه فقط یک ریسمان می‌تواند یک «Mutex» را قفل کند و بقیه‌ی ریسمان-هایی که با فراخوانی «()pthread_mutex_lock» قصد قفل کردن آن را داشته باشند در حالت انتظار[1] قرار می‌گیرند. پس از استفاده از یک قفل، می‌توان منابع اختصاص یافته به آن را با فراخوانی «()pthread_mutex_destroy» آزاد کرد.

```
pthread_mutex_t lock;

pthread_mutex_init(&lock, NULL);   /*              مقداردهی اولیه به یک «Mutex»   */
pthread_mutex_lock(&lock);         /*                         قفل کردن یک «Mutex»   */
pthread_mutex_unlock(&lock);       /*                          باز کردن یک «Mutex»   */
pthread_mutex_destroy(&lock);      /* آزاد کردن منابع اختصاص داده شده به یک «Mutex»   */
```

1 Blocking

کتابخانه‌ی PThreads امکـان استفاده از سمافور[1] را نیز ارائه می‌دهـد. برای یـک سمافور متغیـری با نـوع «sem_t» باید معرفی گردد. فراخوانی «()sem_init» آن را مقدار دهی می‌کند (ورودی سوم، مقـدار اولیه‌ی آن را مشخص می‌کند) و پس از استفاده از آن، فراخوانی «()sem_destroy» منابع اختصاص یافته به آن را آزاد می‌نماید. تابع «()sem_wait» مقدار یک سمافور را کاهش می‌دهد (و در صورت نیاز منتظر می‌ماند) و تابع «()sem_post» مقدار آن را افزایش می‌دهد.

```
sem_t sem;

sem_init(&sem, 0, 1);        /* مقداردهی اولیه به یک سمافور */
sem_post(&sem);              /* افزایش مقدار یک سمافور */
sem_wait(&sem);              /* کاهش مقدار یک سمافور */
sem_destroy(&sem);           /* آزاد کردن منابع یک سمافور */
```

مسئله‌ی تولید کننده و مصرف کننده

در درس سیستم عامل مسئله‌ی تولید کننده و مصرف کننده معرفی شده است. با استفاده از راه‌کارهای ارائه شده در کتابخانه‌ی PThreads می‌توان این مسئله را با در نظر گرفتن دسترسی‌های همزمان حل کرد. هنگامی که اندازه‌ی بافر[2] (که برای انتقال داده‌های تولید شده به مصرف کننده استفاده می‌شود) یک باشد، می‌توان با استفاده از دو سمافور داده‌های تولید شده توسط تولید کننده را به مصرف کننده انتقال داد.

```
sem_t full;                  /* مقدار یک به معنای وجود یک عنصر در بافر است */
sem_t empty;                 /* مقدار یک به معنای خالی بودن بافر است */
sem_init(&full, 0, 0);
sem_init(&empty, 0, 1);
```

در ادامه قسمت مربوط به تولید کننده نشان داده شده است.

```
sem_wait(&empty);            /* انتظار برای خالی شدن بافر */
...                          /* اضافه کردن یک عنصر به بافر */
sem_post(&full);             /* تغییر مقدار سمافور «full» پس از پر شدن بافر */
```

همان طور که در درس مشاهده کرده‌اید، برای اندازه‌ی بافر بزرگ‌تر، لازم است دسترسی‌های همزمان به آن نیز

1 Semaphore
2 Buffer

مدیریت گردد (چگونه؟).

تمرین هشتم

در این تمرین نیز مشابه تمرین هفتم، باید برنامه‌ای را به دو قسمت تقسیم نمایید اما در این تمرین به جای دو پردازه، باید از دو ریسمان استفاده کنید. پس از دریافت فایل «ex7.c»، نام آن را به «ex8.c» تغییر دهید و در شاخه‌ی «~/ex8» قرار دهید. همان طور که در تمرین گذشته دیدید، در این برنامه، قسمت اول پردازش در تابع «prod()» و قسمت دوم در تابع «cons()» انجام می‌شود (در حلقه‌ی تابع «main()»، هر خروجی «prod()» به تابع «cons()» فرستاده می‌شود). در این تمرین باید یک ریسمان بسازید: تابع «prod()» در ریسمان اصلی و تابع «cons()» در ریسمان جدید باید فراخوانی شوند. برای انتقال خروجی تابع «prod()» به ریسمان جدید، باید از تعدادی متغیر مشترک استفاده کنید. با قفل‌های «Mutex» و سمافور از مشکلات مربوط به دسترسی‌های همزمان جلوگیری نمایید.

گام‌های پیشنهادی برای انجام این تمرین:

۱ دریافت و ترجمه‌ی فایل «ex8.c»

۲ ایجاد یک ریسمان جدید

۳ انتظار برای اتمام ریسمان جدید در ریسمان اصلی

۴ انتقال فراخوانی تابع «cons()» به ریسمان جدید

۵ معرفی یک متغیر سراسری برای انتقال داده بین ریسمان‌ها

۶ مشاهده‌ی نیاز به مدیریت دسترسی‌های همزمان

۷ مدیریت دسترسی‌های همزمان توسط دو سمافور

جلسه‌ی نهم — سیگنال‌ها

در این جلسه با سیگنال‌ها در یونیکس و شیوه‌ی دریافت و ایجاد آنها آشنا خواهید شد. همچنین، شیوه‌ی مدیریت دسترسی به فایل‌ها به صورت خلاصه معرفی می‌گردد.

سیگنال‌ها

سیستم عامل می‌تواند با استفاده از سیگنال[1]، پردازه‌ها را از رخدادهای خارجی مطلع سازد. سیگنال‌ها برای اهداف مختلفی استفاده می‌شوند: گاهی برای اطلاع از گذشت زمان مشخص (مثل فراخوانی سیستمی alarm()‎)، گاهی برای گزارش اشکال در اجرای پردازه (مثل اشکال در دسترسی به حافظه)، گاهی برای ارتباط بین پردازه‌ها و گاهی برای اطلاع پردازه از درخواست‌های خارجی (مثل درخواست اتمام پردازه).

سیستم عامل برای هر سیگنال (که با یک عدد مشخص می‌شود) به صورت پیش‌فرض عمل خاصی را در هر پردازه انجام می‌دهد (این عملیات پیش‌فرض در صفحه‌ی راهنمای «signal» شرح داده شده‌اند). هر پردازه می‌تواند عملی که باید بعد از رخداد هر سیگنال (با چند استثنا) انجام شود را تغییر دهد. یکی از راه‌های انجام این کار، استفاده از فراخوانی سیستمی signal()‎ می‌باشد. فراخوانی سیستمی signal()‎ دو ورودی دریافت می‌کند: ورودی اول شماره‌ی سیگنال و ورودی دوم عملی که باید پس از رخداد سیگنال انجام شود را مشخص می‌کنند.

```
signal(SIGINT, SIG_DFL);      /*                  انجام عمل پیش فرض  */
signal(SIGINT, SIG_IGN);      /*  دور انداختن سیگنال (پس از سیگنال عملی انجام نمی‌شود)  */
signal(SIGINT, func);         /*      در صورت بروز سیگنال تابع func صدا زده می‌شود  */

void func(int signo)          /*      تابع «func» باید به این صورت تعریف شده باشد  */
{
        printf("Signal %d\n", signo);
}
```

همان طور که در این مثال مشاهده می‌شود، شماره‌ی سیگنال‌ها توسط ماکروهایی[2] (که در فایل «signal.h» تعریف شده‌اند) مشخص می‌شود. عملی که باید پس از رخداد سیگنال انجام شود با چند ماکرو یا یک تابع

1 Signal
2 Macros

مشخص می‌شود. ماکروی «SIG_DFL» عمل پیش‌فرض را مشخص می‌کند و ماکروی «SIG_IGN» به این مفهوم است که سیگنال باید نادیده گرفته شود. در صورتی که یک تابع به عنوان ورودی دوم به signal() داده شود، در صورت رخداد سیگنال تابع مشخص شده فراخوانی می‌گردد. در مثال قبل این سه حالت نمایش داده شده‌اند.

یکی از سیگنال‌هایی که بسیاری از برنامه‌ها رفتار آن را تغییر می‌دهند سیگنال SIGINT است: در صورتی که کاربر (با استفاده از کلیدهای کنترل و C) درخواست خاتمه‌ی یک برنامه را داشته باشد این سیگنال به برنامه فرستاده می‌شود که به صورت پیش‌فرض موجب خاتمه‌ی آن می‌گردد. سیگنال SIGCHLD پس از اتمام هر یک از فرزندان پردازه به آن ارسال می‌گردد. سیگنال‌های SIGUSR1 و SIGUSR2 برای تعامل بین پردازه‌ها استفاده می‌شود (یک پردازه بتواند پردازه‌ی دیگری را از اتفاقی آگاه کند).

یک پردازه نیز می‌تواند از سیستم عامل درخواست کند تا سیگنالی به پردازه‌ی دیگری فرستاده شود. این کار توسط فراخوانی سیستمی kill() انجام می‌شود. این فراخوانی سیستمی دو ورودی دریافت می‌کند: ورودی اول شماره‌ی پردازه‌ی دریافت کننده‌ی سیگنال و ورودی دوم شماره‌ی سیگنال می‌باشد.

```
kill(pid, SIGUSR1);          /*      ارسال سیگنال «SIGUSR1» به پردازه‌ی با شناسه‌ی «pid»   */
```

مدیریت پردازه‌ها در پوسته

با استفاده از دستور «ps» می‌توان فهرست پردازه‌های در حال اجرا را مشاهده نمود. دستور «ps aux» فهرست همه‌ی پردازه‌های در حال اجرا، صاحب هر پردازه و شماره‌ی آن را نمایش می‌دهد. همچنین دستور «pstree» ساختار درختی پردازه‌ها را به صورت گرافیکی نمایش می‌دهد.

```
$ ps aux
$ pstree -phcU
```

برای فرستادن یک سیگنال از پوسته، می‌توان از دستور «kill» استفاده نمود. در دستور «kill»، می‌توان نام یا شماره‌ی سیگنال را مشخص نمود. نام سیگنال (پس از حذف SIG از شروع آن) باید پس از «-s» قرار گیرد (مثل «-sTERM») و شماره‌ی سیگنال باید بعد از علامت «-» مشخص می‌شود (مثل «-2»).

```
$ kill -sUSR1 pid
```

مدیریت دسترسی‌ها

در سیستم‌های عامل چند کاربره[1] لازم است مکانیزمی برای محافظت از فایل‌ها و به اشتراک گذاشتن آنها وجود داشته باشد. همان طور که در درس سیستم عامل اشاره می‌گردد، برای هر فایل (یا شاخه) در یونیکس یک کاربر به عنوان صاحب آن مشخص می‌گردد. صاحب هر فایل می‌تواند گروه و میزان دسترسی افراد مختلف به آن فایل را مشخص نماید. برای دسترسی به فایل‌ها، افراد به سه دسته تقسیم می‌شوند: صاحب فایل، اعضای گروه فایل و سایر افراد. به ازای هر یک از این دسته‌ها، صاحب فایل می‌تواند مشخص کند که این افراد اجازه‌ی خواندن، نوشتن یا اجرای فایل را دارند یا خیر. معمولا اجازه‌ی دسترسی‌ها به یک فایل را با یک عدد سه رقمی در مبنای هشت (مثل ۶۴۴) نمایش می‌دهند. هر رقم در این نمایش دسترسی یکی از این دسته‌ها را مشخص می‌کند. در «XYZ»، X دسترسی صاحب فایل، Y دسترسی اعضای گروه فایل و Z دسترسی سایر افراد را مشخص می‌نماید. در هر یک از این رقم‌ها، بیت کم‌ارزش توانایی اجرا، بیت بعدی توانایی نوشتن و پر ارزش‌ترین بیت توانایی خواندن را نشان می‌دهد. برای مثال، در صورتی که دسترسی یک فایل «۶۴۰» باشد یعنی صاحب فایل می‌تواند از فایل بخواند و بر روی آن بنویسد، اعضای گروه فایل فقط می‌توانند آن را بخوانند و سایر افراد اجازه‌ی هیچ یک از این عملیات را ندارند. پارامتر l- دستور «ls» صاحب، گروه و دسترسی افراد را برای فایل‌ها نمایش می‌دهد.

```
$ ls -l
-rw-r--r--   1 user     users        143 Dec  4 14:10 Makefile
-rwxr-xr-x   1 user     users       7609 Nov  9 23:57 ex8
-rw-------   1 user     users        412 Nov  9 23:57 ex8.c
-rw-r--r--   1 user     users       2232 Nov  9 23:57 ex8.o
```

صاحب، گروه و دسترسی افراد را می‌توان با دستورات زیر تعیین نمود (راه‌های خلاصه‌تری برای مشخص کردن اجازه‌ی دسترسی افراد برای دستور «chmod» وجود دارد؛ به صفحه‌ی راهنمای آن مراجعه شود).

```
$ chown user path
$ chgrp users path
$ chmod 644 path
```

1 Multi-user

تمرین نهم

برای انجام این تمرین فایل «ex9.c» را دریافت کنید. این برنامه یک ریسمان می‌سازد و در آن مقدار متغیر «found» را به روز می‌رساند. این برنامه را به صورتی تغییر دهید که پس از دریافت سیگنال «SIGUSR1»، آخرین مقدار متغیر «found» را چاپ کند و در صورت دریافت سیگنال «SIGINT»، پس از چاپ مقدار «found» خاتمه یابد.

گام‌های پیشنهادی برای انجام این تمرین:

۱ دریافت و ترجمه‌ی فایل «ex9.c»

۲ نوشتن تابعی برای دریافت سیگنال «SIGUSR1» و فراخوانی «()signal» برای ثبت آن

۳ آزمایش درستی دریافت سیگنال «SIGUSR1» با استفاده از دستور «kill» از پوسته

۴ تکرار مرحله‌ی دو برای سیگنال «SIGINT» و خروج پس از دریافت آن

۵ آزمایش درستی دریافت سیگنال «SIGINT» با فشار دادن دکمه‌های کنترل و «C»